USER'S MANUAL

WFDB-x

The WFDB Software Package in Python, C++, Rust, Julia, R, MATLAB/Octave and Kotlin

Version 1.2.3 | November 2026

Reza Sameni

https://wfdb-x.org

WFDB-x implements version 10.7.0 of the WFDB Software Package. Its library is distributed under the GNU Lesser General Public License, version 2.1 or later, and its command-line tools under the GNU General Public License, version 3 or later (Section 1.9). To cite WFDB-x, see Section 1.8.

Contents

I Concepts 1

1 Introduction 2
- 1.1 What WFDB-x is . . . 2
- 1.2 The seven implementations . . . 3
- 1.3 Three ways to use WFDB-x . . . 4
- 1.4 How the results are checked . . . 5
- 1.5 Versions . . . 6
- 1.6 How this manual is organized . . . 6
- 1.7 Typographic conventions . . . 7
- 1.8 How to cite . . . 7
- 1.9 Authors and license . . . 8

2 WFDB records 9
- 2.1 Records and record names . . . 9
- 2.2 The header file . . . 9
  - 2.2.1 The record line . . . 10
  - 2.2.2 Signal specification lines . . . 10
  - 2.2.3 Info strings and info files . . . 10
  - 2.2.4 An example: the header of record 100s . . . 12
- 2.3 Signal files and storage formats . . . 12
- 2.4 ADC units and physical units . . . 13
- 2.5 Frames, samples per frame and multi-frequency records . . . 14
- 2.6 Multi-segment records . . . 14
- 2.7 Annotation files . . . 15
- 2.8 Times and dates . . . 16
  - 2.8.1 Time strings . . . 16
  - 2.8.2 Formatted times . . . 16
  - 2.8.3 Base time and dates . . . 17
- 2.9 Calibration files . . . 17
- 2.10 EDF and EDF+ records . . . 18

3 Where files are read and written 19
- 3.1 The input and output directories . . . 19
- 3.2 Record names . . . 20
- 3.3 How a name is resolved . . . 20
  - 3.3.1 Files named in a header . . . 21
  - 3.3.2 Files that a tool opens itself . . . 22
  - 3.3.3 Legacy mode . . . 22
- 3.4 Network access . . . 23
- 3.5 The standard input and output . . . 24
- 3.6 Environment variables . . . 24

4 Differences from the original package 26
- 4.1 No built-in sampling frequency, gain, ADC resolution or units . . . 26
- 4.2 Files are found only where they are named . . . 28
- 4.3 Out-of-order annotation files are sorted in the library . . . 29

4.4 An output signal file is never one of the input signal files . . . 29
4.5 Write errors give a nonzero exit status . . . 30
4.6 The record line of a header is found as the header parser finds it . . . 30
4.7 Input on which the C package crashes . . . 30
4.8 Additions . . . 31
4.9 Corrections of version 1.2.0 . . . 33
4.10 Corrections of version 1.2.2 . . . 34
4.11 Corrections of version 1.2.3 . . . 35
4.12 Legacy mode . . . 35
4.13 Errors and warnings . . . 36
4.14 Known C behaviors that WFDB-x reproduces . . . 37

5 Header metadata 39
5.1 Entries . . . 39
5.2 Arguments . . . 40
5.3 Writing . . . 41
5.4 Which file . . . 41
5.5 read_header and write_record . . . 42
5.6 An example . . . 42
5.7 The functions in each language . . . 43

II The seven languages 45

6 Python 46
6.1 Installation and requirements . . . 46
6.1.1 Requirements . . . 46
6.1.2 Installing from PyPI . . . 46
6.1.3 Installing from the repository . . . 47
6.1.4 The sample record and the example notebooks . . . 47
6.2 Quick start . . . 47
6.3 The high-level API . . . 48
6.3.1 Conventions . . . 48
6.3.2 Reading records . . . 50
6.3.3 Writing records . . . 54
6.3.4 Checking records . . . 59
6.3.5 Header metadata . . . 60
6.3.6 The sample record and the example notebooks . . . 64
6.3.7 The result classes . . . 65
6.3.8 Warnings and errors . . . 69
6.4 The WFDB library functions in Python . . . 70
6.4.1 Conventions . . . 70
6.4.2 Module layout . . . 72
6.4.3 The structures . . . 72
6.4.4 The settings functions . . . 73
6.4.5 Examples . . . 74
6.5 The command-line tools in Python . . . 75
6.6 Testing . . . 76

7 C++ 78
7.1 Installation and requirements . . . 78
7.1.1 Requirements . . . 78
7.1.2 Building and installing . . . 78
7.1.3 Using the library from CMake . . . 79
7.1.4 The sample record . . . 79
7.2 Quick start . . . 80
7.3 The high-level API . . . 81
7.3.1 Conventions . . . 81

7.3.2 Reading records . . . 83
7.3.3 Writing records . . . 86
7.3.4 Checking records . . . 91
7.3.5 Header metadata . . . 92
7.3.6 The options . . . 96
7.3.7 The result types . . . 99
7.3.8 Warnings and errors . . . 101
7.4 The WFDB library functions in C++ . . . 103
7.4.1 Conventions . . . 103
7.4.2 The structures . . . 104
7.4.3 The settings functions . . . 105
7.4.4 Examples . . . 106
7.5 The command-line tools in C++ . . . 108
7.6 Testing . . . 108

8 Rust **110**
8.1 Installation and requirements . . . 110
8.1.1 Requirements . . . 110
8.1.2 Using the library in a Cargo project . . . 110
8.1.3 Building and installing the tools . . . 111
8.1.4 The sample record . . . 111
8.2 Quick start . . . 111
8.3 The high-level API . . . 112
8.3.1 Conventions . . . 113
8.3.2 Reading records . . . 115
8.3.3 Writing records . . . 119
8.3.4 Checking records . . . 125
8.3.5 Header metadata . . . 127
8.3.6 The options . . . 134
8.3.7 The result types . . . 137
8.3.8 Warnings and errors . . . 140
8.4 The WFDB library functions in Rust . . . 142
8.4.1 Conventions . . . 142
8.4.2 The structures . . . 144
8.4.3 The settings functions . . . 144
8.4.4 Examples . . . 146
8.5 The command-line tools in Rust . . . 147
8.6 Testing . . . 148

9 Julia **150**
9.1 Installation and requirements . . . 150
9.1.1 Requirements . . . 150
9.1.2 Installing the package . . . 150
9.1.3 The command-line tools . . . 151
9.1.4 The sample record . . . 151
9.2 Quick start . . . 151
9.3 The high-level API . . . 152
9.3.1 Conventions . . . 152
9.3.2 Reading records . . . 154
9.3.3 Writing records . . . 159
9.3.4 Checking records . . . 164
9.3.5 Header metadata . . . 166
9.3.6 The result types . . . 170
9.3.7 Warnings and errors . . . 173
9.4 The WFDB library functions in Julia . . . 174
9.4.1 Conventions . . . 174
9.4.2 The structures . . . 175

9.4.3 The settings functions . . . 176
9.4.4 Examples . . . 177
9.5 The command-line tools in Julia . . . 178
9.6 Testing . . . 180

10 R **181**
10.1 Installation and requirements . . . 181
10.1.1 Requirements . . . 181
10.1.2 Installing the package . . . 181
10.1.3 The command-line tools . . . 182
10.1.4 The sample record . . . 182
10.2 Quick start . . . 182
10.3 The high-level API . . . 183
10.3.1 Conventions . . . 183
10.3.2 Reading records . . . 184
10.3.3 Writing records . . . 189
10.3.4 Checking records . . . 194
10.3.5 Header metadata . . . 195
10.3.6 The results . . . 200
10.3.7 Warnings and errors . . . 202
10.4 The WFDB library functions in R . . . 204
10.4.1 Conventions . . . 204
10.4.2 The structures . . . 205
10.4.3 The settings functions . . . 206
10.4.4 Examples . . . 207
10.5 The command-line tools in R . . . 208
10.6 Testing . . . 210

11 MATLAB and Octave **212**
11.1 Installation and requirements . . . 212
11.1.1 Requirements . . . 212
11.1.2 Installing . . . 213
11.1.3 The sample record . . . 213
11.2 Quick start . . . 213
11.3 The high-level API . . . 214
11.3.1 Conventions . . . 214
11.3.2 Reading records . . . 215
11.3.3 Writing records . . . 220
11.3.4 Checking records . . . 224
11.3.5 Header metadata . . . 226
11.3.6 The result structures . . . 230
11.3.7 Warnings and errors . . . 233
11.4 The WFDB library functions in MATLAB and Octave . . . 235
11.4.1 Conventions . . . 235
11.4.2 The structures . . . 236
11.4.3 The settings functions . . . 237
11.4.4 Examples . . . 238
11.5 The command-line tools in MATLAB and Octave . . . 239
11.6 Testing . . . 240

12 Kotlin **242**
12.1 Installation and requirements . . . 242
12.1.1 Requirements . . . 242
12.1.2 Building and installing the library . . . 243
12.1.3 The command-line tools . . . 243
12.1.4 Licenses . . . 243
12.1.5 The sample record . . . 243
12.1.6 Android and iOS . . . 244

12.2 Quick start . . . 244
12.3 The high-level API . . . 245
12.3.1 Conventions . . . 245
12.3.2 Reading records . . . 246
12.3.3 Writing records . . . 251
12.3.4 Checking records . . . 257
12.3.5 Header metadata . . . 259
12.3.6 Settings . . . 264
12.3.7 The result classes . . . 266
12.3.8 Warnings and errors . . . 269
12.4 The WFDB library functions in Kotlin . . . 271
12.4.1 Conventions . . . 271
12.4.2 Package layout . . . 272
12.4.3 The structures . . . 272
12.4.4 Examples . . . 273
12.5 The command-line tools in Kotlin . . . 275
12.6 Testing . . . 276

III Reference 277

13 The WFDB library functions 278
13.1 Calling the library functions from each language . . . 278
13.1.1 Python . . . 280
13.1.2 C++ . . . 280
13.1.3 Rust . . . 281
13.1.4 Julia . . . 282
13.1.5 R . . . 282
13.1.6 MATLAB/Octave . . . 283
13.1.7 Kotlin . . . 284
13.2 Opening and closing records . . . 285
13.3 Reading and writing signals . . . 289
13.4 Time and date conversion . . . 294
13.5 Amplitude conversion . . . 298
13.6 Annotations . . . 300
13.6.1 Classification and mapping of annotation codes . . . 305
13.7 Calibration . . . 311
13.8 Header and info strings . . . 313
13.9 Error handling and settings . . . 318
13.10 File and path functions . . . 320
13.11 WFDB-x additions . . . 321
13.12 Constants and structures . . . 327

14 The command-line tools 330
14.1 Running the tools . . . 330
14.2 Conventions . . . 331
14.2.1 Records and files . . . 331
14.2.2 Time arguments . . . 332
14.2.3 Help, options and arguments . . . 332
14.2.4 Legacy mode . . . 332
14.2.5 Standard input and output . . . 332
14.2.6 Exit status . . . 333
14.3 Summary of the tools . . . 333
14.4 Signal tools . . . 334
14.5 Annotation tools . . . 344
14.6 Format converters . . . 348

15 Storage formats and annotation codes 359
15.1 Signal storage formats . . . 359
15.1.1 Format 0: null signals . . . 360
15.1.2 Format 8: first differences . . . 360
15.1.3 Formats 16 and 61: 16-bit amplitudes . . . 360
15.1.4 Formats 80 and 160: offset binary . . . 360
15.1.5 Format 212: 12-bit amplitudes . . . 360
15.1.6 Format 310: 10-bit amplitudes in 16-bit words . . . 361
15.1.7 Format 311: 10-bit amplitudes in 32-bit words . . . 361
15.1.8 Formats 24 and 32: 24- and 32-bit amplitudes . . . 361
15.2 FLAC formats . . . 361
15.3 Annotation files . . . 362
15.3.1 MIT format . . . 362
15.3.2 AHA format . . . 363
15.4 Annotation codes . . . 364
15.4.1 Mapping tables . . . 365

PART I

Concepts

CHAPTER 1

Introduction

WFDB-x is a reimplementation of the WFDB Software Package, version 10.7.0: the C library and the command-line tools with which physiologic signal and annotation files in the WFDB format, such as those of the PhysioNet databases, are read, written and converted. WFDB-x reimplements the package in seven programming languages: Python, C++, Rust, Julia, R, MATLAB/Octave and Kotlin. In each of them it reads and writes the same files as the original C library, provides the same functions under their C names, and provides the same command-line tools, with the same options, output, messages and exit status. The seven implementations are tested to give identical results, byte for byte, to the original C programs and to each other.

WFDB-x is an independent project. It is not part of the WFDB project (`https://wfdb.io/`) or of PhysioNet (`https://physionet.org/`), and is not endorsed by them. It follows the WFDB format and the behavior of the original C package, which serves as its specification: the expected results of all its tests were produced by the C programs.

1.1 What WFDB-x is

Each of the seven implementations

- reads and writes exactly the same files as the original C library: headers (`.hea`), signal files in every WFDB storage format (8, 16, 61, 80, 160, 212, 310, 311, 24, 32, and the FLAC formats 508, 516 and 524), annotation files (in MIT and AHA format), EDF and EDF+ files, calibration files, info files, and multi-segment records (see Chapter 2);
- provides the same 33 command-line tools as the C package (listed in Section 1.3), with the same options;
- provides the WFDB library functions under their C names (`isigopen`, `getvec`, `getann`, `putann`, `strtim`, ...), with the behavior of the C functions, and a high-level API in the idiom of the language;
- is written natively in its language: none of the implementations calls the C library or another language. It needs nothing beyond the standard library of its language, except NumPy for Python. FLAC files are read and written without libFLAC. Records are read from URLs with the system `curl` program in C++ and Rust (and in R and MATLAB/Octave when `curl` is installed), and with the standard facilities of the language otherwise (see Section 3.4).

WFDB-x behaves exactly as the original package, with a few deliberate exceptions, which are the same in all seven languages and are tested like everything else: it never assumes a sampling frequency, gain, ADC resolution or units that a record does not give; it reads and writes files only where they are named, and opens URLs only when network access is turned on; it sorts annotation files itself, without losing annotations; a tool never writes a signal file over a signal file it is reading; a tool that cannot write its output exits with status 1; and `getinfo` finds the record line of a header as the header parser does. Chapter 4 describes each of them, and *legacy mode*, which restores the original behavior exactly.

The interactive viewer of the C package (WAVE), its plotting programs, and the analysis programs of its `psd/` directory are outside the scope of WFDB-x.

The speed and memory use of the seven implementations, compared with those of the C library and tools, are measured in `docs/BENCHMARKS.md`. In short, Rust, Julia and C++ read an hour of a two-signal ECG in 4 to 6 ms, 4 to 7 times faster than the C library, which reads one frame per `getvec` call; the high-level functions of every language read whole blocks of samples, and are the fastest way to read a record in that language.

1.2 The seven implementations

The website of WFDB-x is `https://wfdb-x.org`, and its source code is in the repository `https://github.com/alphanumericslab/wfdb-x`. All seven implement the complete library and all 33 tools. Each has a directory of the repository, with a `README.md` that gives its requirements, installation instructions and examples, and a chapter of Part II. Table 1.1 shows how each is installed, from a copy of the repository (or from PyPI for Python), and how a tool is run in it. In code, the packages are named `wfdbx` (Python, MATLAB/Octave, Rust, R, Kotlin, and the C++ namespace) or `WFDBX` (Julia).

Table 1.1 The seven implementations: installation, and how a tool is run.

Language	Directory	Installation	Running a tool
Python	`python/`	`pip install wfdb-x` (PyPI), or `pip install ./python`	`wfdbx rdsamp -r 100s`
C++	`cpp/`	`cmake -S cpp -B cpp/build && cmake --build cpp/build`	`rdsamp -r 100s`
Rust	`rust/`	`cargo install --path rust`	`rdsamp -r 100s`
Julia	`julia/`	`using Pkg; Pkg.develop(path="/path/to/wfdb-x/julia")`	`wfdbx rdsamp -r 100s`
R	`r/`	`R CMD INSTALL r`	`wfdbx rdsamp -r 100s`
MATLAB / GNU Octave	`matlab/`	`addpath('/path/to/wfdb-x/matlab')` (tested with GNU Octave; MATLAB untested)	`wfdbx rdsamp -r 100s`
Kotlin (Android, iOS, JVM)	`kotlin/`	`cd kotlin && ./gradlew publishToMavenLocal`, then `implementation("io.github.alphanumericslab:wfdbx:1.2.3")` (Gradle, with `mavenLocal()`)	`kotlin/bin/wfdbx rdsamp -r 100s`

In Julia, R and MATLAB/Octave, `wfdbx` is a short alias for the language's own launcher of the tools; the chapter of each language in Part II shows how to set it up. The Python package is distributed on PyPI as `wfdb-x` and installs the `wfdbx` command; `wfdbx --list` lists the tools, `wfdbx --version` prints the version, and `wfdbx --examples` copies the example notebooks of the package.

The Kotlin implementation runs on the JVM, on Android, and on iOS through Kotlin Multiplatform; it is tested on the JVM. The MATLAB/Octave implementation is written for both MATLAB and GNU Octave, and is tested with GNU Octave.

The sample record. Record `100s`, the first minute of record 100 of the MIT-BIH Arrhythmia Database (two ECG signals sampled at 360 Hz, and its beat annotations), is in the directory `tests/data` of the repository, and the Python package includes a copy of it (in the directory that `wfdbx.sample_data_dir()` returns). The examples of this manual use it. To run them, run the tools in that directory, or set the input directory to it (see Chapter 3), for example:

```
cd /path/to/wfdb-x/tests/data
```

```
wfdbx rdsamp -r 100s -t s3
```

which prints the first three samples of each signal, in ADC units:

```
               0     995    1011
               1     995    1011
               2     995    1011
```

1.3 Three ways to use WFDB-x

Every language offers the same three ways in, summarized in Table 1.2. All three are written natively in the language.

Table 1.2 The three ways to use WFDB-x.

	What it is	Use it for
1. The high-level API (recommended)	A few functions in the idiom of the language: `read_record`, `read_header`, `read_annotations`, `write_record`, `write_annotations`, `check_record`, and the header metadata functions `read_metadata`, `get_metadata`, `set_metadata`, `remove_metadata` and `update_metadata` (`readRecord`, `...` in Kotlin). One call reads a whole record or annotation file into arrays, with named options, errors reported as exceptions or error values, and fast block reads. It has no counterpart in the C package.	New code, analysis, notebooks, machine learning: anything that needs a record as arrays.
2. The WFDB library functions	The C library API under its C names (`isigopen`, `getvec`, `getann`, `putann`, `strtim`, ...), with the C behavior, frame by frame and annotation by annotation.	Porting a program written for the C library, or processing a record as a stream with the exact C semantics.
3. The command-line tools	The 33 tools (`rdsamp`, `rdann`, `wrsamp`, `xform`, ...) with the options, output, messages and exit status of the C tools, implemented in each language.	Shell scripts and pipelines, and reproducing work done with the original tools, without the C package.

The high-level API is built on the library functions, and returns exactly the samples and annotations that they, and the C library, give; the tools are built on the library functions too. The three can be mixed in one program, with one caution: the high-level readers and writers call `wfdbquit`, which closes the files opened with the library functions.

In Python, for example, the first three samples of record `100s` are read in physical units with the high-level API as follows:

```
import wfdbx
rec = wfdbx.read_record("100s", sampto=3, physical=True)
print(rec.fs, rec.names, rec.signals.shape)   # 360.0 ['MLII', 'V5'] (3, 2)
print(rec.signals[0])                        # [-0.145 -0.065]
```

and the first sample of each signal is read in ADC units, and converted, with the library functions:

```
import wfdbx
```

```
si = []
n = wfdbx.isigopen("100s", si, 2)     # 2 signals, described in si
v = [0] * n
wfdbx.getvec(v)                       # v is [995, 1011]
print(wfdbx.aduphys(0, v[0]))         # -0.145
wfdbx.wfdbquit()
```

The 33 tools are those of the `app/` and `convert/` directories of the C package:

- signals and annotations: `rdsamp`, `wrsamp`, `rdann`, `wrann`, `sortann`, `ann2rr`, `rr2ann`, `wfdbdesc`, `wfdbtime`, `time2sec`, `sampfreq`, `signame`, `signum`, `sigamp`, `snip`, `xform`, `wfdbcat`, `wfdbwhich`;
- format converters: `edf2mit`, `mit2edf`, `rdedfann`, `wav2mit`, `mit2wav`, `wfdb2mat`, `a2m`, `m2a`, `ad2m`, `md2a`, `ahaecg2mit`, `makeid`, `readid`, `revise`, `parsescp`.

Part III describes each of them, and each library function.

1.4 How the results are checked

For the same input, a WFDB-x tool produces byte for byte the same standard output, standard error (including its error messages), exit status and output files as the original C tool, and the library functions return the same values as their C counterparts. The exceptions are the deliberate differences of Chapter 4; the reference results are produced by the C code with these changes applied, so that they are tested in the same way. This is verified by tests whose expected results were produced by the original C code, not written by hand:

- **The conformance suite** (`tests/conformance/`, described in `docs/conformance.md`): 1155 command-line test cases, run by a harness that does not depend on the language of the implementation. Their expected results were produced by the C programs, built from the original sources (`third_party/wfdb-main.zip`) with FLAC support and with the patch `tests/reference/divergences.patch`, which defines the deliberate differences, applied. The exceptions are 20 cases whose input makes the C programs crash; their expected results are the behavior that WFDB-x defines for that input (Section 4.7). A second suite, for legacy mode, has 886 cases: the cases of the original package, run with `--legacy` and compared with the results of the unmodified C package. Every implementation passes all the cases of both suites on the reference platform, x86-64 Linux with glibc.
- **Test records** (`tests/data/`), written by the C library itself (`tests/reference/gen_records.c`). They cover every storage format, multi-frequency records, multi-segment records of fixed and of variable layout, skewed signals, byte offsets, block-structured files, EDF, and annotation files that use every feature of the annotation format.
- **Unit tests** of the library and of the high-level functions, in each language directory, with expected values printed by the C library (`tests/reference/api_expect.c`).
- **Metadata tests** (`tests/metadata/`): 56 cases of the header metadata functions, which every implementation must pass with byte-identical headers, and a check that the original C programs read the changed headers as before (Chapter 5).

Because all seven implementations must pass the same cases, each is also checked against the other six. The MATLAB/Octave code is tested with GNU Octave, also through the MATLAB driver `matlab/bin/wfdbx_matlab.sh`; MATLAB itself has not been available to run the tests.

A few behaviors of the C code are undefined (crashes, reads of uninitialized memory), and a few depend on the platform. They are listed, with the behavior that WFDB-x chose, in `docs/KNOWN_C_BEHAVIORS.md` and summarized in Section 4.14.

1.5 Versions

WFDB-x has its own version numbers, shared by the seven implementations, which follow semantic versioning (`https://semver.org`): a new patch number (1.1.x) for corrections only, a new minor number (1.x.0) for new features that do not change existing behavior, and a new major number (x.0.0) for changes that can break existing code. This manual describes version 1.2.3. The file `CHANGELOG.md` lists the changes in each version:

- 1.0.0 (16 October 2026): the WFDB library, its 33 command-line tools and a high-level API in Python, C++, MATLAB/Octave, Rust, Julia and R;
- 1.0.1 (27 October 2026): the Kotlin Multiplatform implementation, the seventh language;
- 1.1.0 (28 October 2026): faster reading and writing of records, signal names without surrounding spaces in the high-level readers, and write errors and frame checks in the high-level writers;
- 1.2.0: the header metadata functions (Chapter 5), this manual, and the reference to cite (Section 1.8). No existing function or tool changes behavior, except for three corrections: the files that a tool reads or creates itself follow the input and output directories, `osigopen(record, NULL, 0)` returns 0 in every language, and the high-level functions leave the annotation time resolution and the getvec mode as they were (Section 4.9);
- 1.2.1: the library under the GNU Lesser General Public License, version 2.1 or later, with the command-line tools still under the GNU General Public License, version 3 or later (Section 1.9); and the C++ library builds on Windows again;
- 1.2.2: corrections in all seven languages, without new functions (Section 4.10): the high-level readers report a signal file that ends early as a read error even when a checksum error is found in its last frame, and never return an incomplete frame; the high-level writers check every argument before they touch any file, refuse the annotation codes outside 0 to 58 (59 to 63 are reserved) and the times -2^{63} and $2^{63}-1$, and write through temporary files that replace the existing files only when all of them are complete; `getinfo` finds the record line of a header as the header parser does (except in legacy mode); a remote file whose download failed is requested again the next time it is opened, and a name with a URL scheme other than `http://` and `https://` is refused instead of being read as a local file name (Section 3.4); and the file `NOTICE` (the licenses, and the notice of libFLAC) is included in every package;
- 1.2.3: corrections reported by users (Section 4.11): `write_record` refuses samples that the storage format cannot hold (they were silently wrapped around) and more than 8 signals in a FLAC format, and warns about samples that will be read back as invalid; and, in Python and R, a tool run in the same process writes to a captured standard output.

The version numbers of WFDB-x are independent of the version of the WFDB Software Package that it implements, 10.7.0, in every version. The library reports that version as the C library does: `wfdbversion()` returns `"10.7.0"`, and `wfdberror()`, before any error, returns `"WFDB library version 10.7.0 (Jun 10 2022).\n"`.

1.6 How this manual is organized

- **Part I, Concepts**, describes what the seven implementations share: this introduction; the WFDB records and their files (Chapter 2); where files are read and written, network access and the environment variables (Chapter 3); the differences from the original package, legacy mode, and the reporting of errors (Chapter 4); and the header metadata (Chapter 5).

- **Part II, The seven languages**, has one chapter for the users of each language: installation, the high-level API with every function, its arguments, results, errors and examples, and how the library functions and the tools are used in that language (Chapters 6 to 12).
- **Part III, Reference**, describes the WFDB library functions and the command-line tools, which are the same in every language, and the storage formats and annotation codes (Chapter 15).

1.7 Typographic conventions

- `read_record`, `WFDB_INVALID_SAMPLE`, `"0:10"`: identifiers, values and code, in a typewriter font.
- `rdsamp -r 100s`: a command line; `-r`: an option of a tool; `tests/data/100s.hea`: a file or a directory.
- The WFDB library functions and the tools are named by their C names, which are their names in every language.
- Listings of code and of command lines are set on a shaded background. A listing on a white background, with a bold blue rule, is the call syntax of a function, copied from the code. The Python examples marked as examples are run, as part of the checks of this manual, in a directory that holds a copy of `tests/data`, which is the current directory and the input directory; their results are given in comments.

Three kinds of boxes set off remarks:

Note A note gives details that matter in some cases, such as a behavior of the C package that WFDB-x reproduces.

Tip A tip gives a way to do something faster or more simply.

Patient data A box of this kind concerns data that can identify patients, such as names, dates or ages stored in a header.

1.8 How to cite

To refer to WFDB-x in a publication, cite the preprint that describes it:

> Reza Sameni. *WFDB-x: Conformance-Tested Implementations of the WFDB Software Package in Seven Programming Languages.* arXiv:2611.01358, 2026.
> DOI: 10.48550/ARXIV.2611.01358.
> https://arxiv.org/abs/2611.01358

As a BibTeX entry:

```
@misc{wfdb-x,
  doi = {10.48550/ARXIV.2611.01358},
  url = {https://arxiv.org/abs/2611.01358},
  author = {Sameni,  Reza},
  title = {{WFDB-x: Conformance-Tested Implementations of the WFDB Software Package in Seven
    ↪ Programming Languages}},
  publisher = {arXiv},
  year = {2026},
  copyright = {Creative Commons Attribution Share Alike 4.0 International}
}
```

The same reference is in the file `CITATION.cff` of the repository. Work that uses the WFDB format or PhysioNet data should also cite the WFDB Software Package and PhysioNet, as PhysioNet asks.

1.9 Authors and license

The WFDB Software Package was written by George B. Moody, Benjamin Moody and the WFDB contributors; it is Copyright © 1983–2022 George B. Moody and the WFDB contributors. WFDB was originally developed at MIT by George B. Moody, and builds on decades of work by him, Roger G. Mark and the broader research community. The authors of the C code that each part of WFDB-x translates are credited, file by file, in each WFDB-x source file and in `docs/AUTHORS.md`.

WFDB-x was developed by Reza Sameni (2026). It is an independent reimplementation of the WFDB Software Package: its code in each language is a translation of the package's C code, written for WFDB-x, not code by the original WFDB developers.

The WFDB-x library is distributed under the GNU Lesser General Public License, version 2.1 or (at the user's option) any later version (LGPL-2.1-or-later; see the file `LICENSE`): programs under any license, proprietary ones included, may use it on the LGPL's terms. The command-line tools (each language's `apps` and the `wfdbx` command), translations of the tools of the original package, are distributed under the GNU General Public License, version 3 or (at the user's option) any later version (GPL-3.0-or-later; see the file `LICENSE-GPL`). Each source file states its license. The original C sources, kept unmodified in `third_party/`, keep their own licenses: the GNU Library General Public License, version 2 or later, for the library, and the GNU General Public License, version 2 or later, for the tools; their "or later" terms allow those of WFDB-x.

The file `NOTICE`, at the root of the repository, says which parts of WFDB-x are under which license. It also gives the copyright notice and the license (BSD-3-Clause) of libFLAC 1.4.3, by Josh Coalson and the Xiph.Org Foundation: the FLAC decoder of each implementation is a translation of the stream decoder of libFLAC, and these files carry libFLAC's notice (libFLAC itself is not included in WFDB-x, which needs no external library at run time). `NOTICE` is shipped with the licenses (`LICENSE` and `LICENSE-GPL`; in the R package, `COPYING.LESSER` and `COPYING`, with a file `LICENSE`, named by its `License: file LICENSE`, that says which files are under which license) in the package of every language: the Python wheel and source distribution, the R package (`inst/NOTICE`), the files installed by the C++ build (in `share/doc/wfdbx`), the Rust crate, the Julia package, the Kotlin jars (`META-INF`) and the directory `matlab/`.

CHAPTER 2

WFDB records

A WFDB *record* is a recording of one or more signals, such as the leads of an ECG, and the files that describe and annotate it: a *header file*, which describes the record and its signals; one or more *signal files*, which hold the samples; and any number of *annotation files*, which label events of the signals, such as beats. A record may also have an *info file* of comments. This chapter describes these files as WFDB-x reads and writes them, which is as the WFDB library 10.7.0 does, except where Chapter 4 says otherwise. The exact layout of each storage format, and the annotation codes, are in Chapter 15.

2.1 Records and record names

A record is designated by its *record name*, such as `100s`. The files of a record are found from it:

- the header is the file *record*`.hea` (`100s.hea`);
- the signal files are named in the header (`100s.dat`);
- an annotation file is the file *record*.*annotator*, where the *annotator name* says which set of annotations it holds (`100s.atr`, the reference annotations of record `100s`, annotator `atr`);
- the info file is *record*`.info`.

A record name may include a directory (`mitdb/100`, `/data/mitdb/100`), and is then found as Chapter 3 describes. A trailing `.hea` is removed from a record name by the library functions that open a record, so `100s.hea` designates record `100s`. A name that ends with `/` stands for a record of the same name in that directory: `abc/123/` is `abc/123/123`. A record name that ends with another extension, such as `100s.edf`, names an EDF file (Section 2.10). The record name `-` stands for the standard input or output (Section 3.5).

When a header or an annotation file is created, the record name, and the annotator name, may contain only letters, digits, `_`, `-`, `~` and `/`; any other character is an error (for example, `init: illegal character 46 in record name`). In a record name written under a path, such as `out.d/rec`, only the last component, the record's own name, is checked (the C library checked the whole name; see Section 3.3).

2.2 The header file

A header is a text file of lines separated by line feeds; a carriage return may precede each line feed. Its fields are separated by spaces or tabs. The lines are of four kinds:

- the *record line*, the first line that holds something other than spaces, tabs and carriage returns, and whose first such character is not `#` (Section 2.2.3);
- one *signal specification line* per signal, for an ordinary record, or one *segment specification line* per segment, for a multi-segment record (Section 2.6);
- *comment lines*, whose first character is `#`; those that follow the record line are the *info strings* of the record;

- empty lines, which are ignored.

Comment lines and empty lines may appear anywhere. If the first line of a header begins with `#wfdb` followed by a version number, such as `#wfdb 10.7.1`, a library older than that version refuses the header (`init: reading record REC requires WFDB library version 10.7.1 or later`); WFDB-x has the version of the C library that it implements, 10.7.0.

2.2.1 The record line

The record line gives the fields of Table 2.1, in this order, separated by spaces or tabs. Only the record name and the number of signals are required; each optional field may be given only if the fields before it are.

Table 2.1 The fields of the record line.

Field	Meaning
record name	The name of the record. It must match the last component of the name under which the header is opened, or the header is refused (`init: record name in record REC header is incorrect`).
/segments	(Optional, attached to the record name.) The number of segments of a multi-segment record, as in `multi/3`.
number of signals	An integer; 0 is allowed. Several signals can share one signal file.
sampling frequency	(Optional.) The number of frames per second, a floating-point number greater than 0 (`360`, `360.0` and `3.6e2` are equivalent). WFDB-x assumes none: if it is missing, the frequency is unspecified (0), unless one was set with `setsampfreq` before the record was opened (the C library assumed 250 Hz; see Section 4.1).
/counter frequency	(Optional, attached to the sampling frequency.) The frequency of a counter, such as the page numbers of a chart recording, used by the time strings that begin with `c` (Section 2.8). If it is absent or not positive, it is the sampling frequency.
(*base counter*)	(Optional, attached to the counter frequency.) The counter value of sample 0; 0 if absent.
number of samples	(Optional.) The number of samples per signal. 0 or a missing value means that the length is unspecified, and then the checksums of the signals are not verified.
base time	(Optional.) The time of day of sample 0, as `hh:mm:ss`, with optional milliseconds (`12:34:56.789`).
base date	(Optional, after the base time.) The date of sample 0, as `dd/mm/yyyy`.

2.2.2 Signal specification lines

Each signal specification line gives the fields of Table 2.2, in this order; again each optional field may be given only if the fields before it are. The lines of the signals that share a signal file must be consecutive; such signals form a *signal group*, and must have the same format and block size (otherwise `init: error in specification of signal N or M`).

2.2.3 Info strings and info files

The *info strings* of a record are the comment lines of its header that follow the record line, without their initial `#`, as the library function `getinfo` reads them: every line after the record line whose first character is `#` is an info string, even one between two signal lines. A line that begins with a space before its `#` is not an info string. The comment lines before the record line are not info strings, and a header without a record line has none.

Table 2.2 The fields of a signal specification line, and the values that WFDB-x uses when a field is missing.

Field	Meaning, and value when missing
file name	The signal file, relative to the directory of the header (Section 3.3.1); - is the standard input.
format	The storage format (Section 2.3), an integer such as `212`. Required.
x*spf*	(Attached to the format.) The number of samples per frame (Section 2.5); 1 if missing (or less than 1).
:*skew*	(Attached.) The number of samples of the signal that precede sample 0, to correct a skew between signals; 0 if missing.
+*offset*	(Attached.) The number of bytes at the beginning of the signal file that precede the samples (a prolog); 0 if missing.
gain	The number of ADC units per physical unit, a floating-point number. 0, or a missing field, means that the signal is *uncalibrated* (Section 2.4).
(*baseline*)	(Attached to the gain.) The ADC value that corresponds to 0 physical units; the ADC zero if missing.
/*units*	(Attached to the gain or baseline.) The physical units, such as `mV`, without spaces (at most 50 characters). If missing, the units are unspecified; WFDB-x never takes them to be millivolts, and its tools show them as `nd`.
ADC resolution	The resolution of the analog-to-digital converter, in bits. If missing, it is unspecified (0); the C library inferred it from the format or assumed 12 bits.
ADC zero	The ADC value at the middle of the ADC's input range; 0 if missing.
initial value	The value of sample 0, used by the difference format 8; the ADC zero if missing.
checksum	The sum of all the samples of the signal, as a 16-bit signed integer, verified when the end of the signal is read (if the record line gives the number of samples); 0 if missing.
block size	For signal files that must be read in blocks of a given size (character special files); 0 (none) if missing.
description	The rest of the line, which may contain spaces (at most 100 characters), such as `MLII`. If missing: `record REC, signal N`.

The record line is found as the header parser of the library (`readheader`) finds it: the first line whose first token (separated by spaces, tabs, carriage returns and line feeds) does not begin with `#`. Empty and blank lines before it are therefore skipped, as are comment lines, even those that begin with spaces. The C function `getinfo` has another rule, the first line that does not begin with `#`: for a header whose first line is empty or blank, it takes that line for the record line, and also returns the comments before the real record line as info strings. Since version 1.2.2, WFDB-x's `getinfo` follows the header parser, except in legacy mode, which keeps the C rule; the `comments` of the high-level readers, the notes of `wfdbdesc` and the metadata functions (in every mode) follow the same rule (Section 4.6 gives an example).

After the header, `getinfo` returns the comment lines of the record's info file, *record*`.info`, if there is one. `putinfo` writes an info string, as `#` followed by the string and a CR LF line ending, into the header being written, or into the info file opened by `setinfo`. Chapter 5 describes the info strings of the form `Key: value`, which hold the metadata of a record.

2.2.4 An example: the header of record 100s

The header of the sample record `100s`, `tests/data/100s.hea`, is:

```
100s 2 360 21600
100s.dat 212 200 11 1024 995 21537 0 MLII
100s.dat 212 200 11 1024 1011 -3962 0 V5
# 69 M 1085 1629 x1
# Aldomet, Inderal
#Produced by xform from record 100, beginning at 0:0
```

- The record line says that record `100s` has 2 signals, sampled at 360 frames per second, of 21600 samples each (60 seconds). It gives no base time or date.
- Each signal line says that the signal is in the file `100s.dat` (both signals share it: they form signal group 0), in format 212, with a gain of 200 ADC units per physical unit, no units, an ADC resolution of 11 bits, an ADC zero of 1024, an initial value of 995 (signal 0) or 1011 (signal 1), a checksum of 21537 or -3962, a block size of 0, and the description `MLII` or `V5`. The baseline is not given, so it is the ADC zero, 1024. A sample of 995 ADC units in signal 0 is thus $(995 - 1024)/200 = -0.145$ physical units.
- The last three lines are the info strings `" 69 M 1085 1629 x1"`, `" Aldomet, Inderal"` and `"Produced by xform from record 100, beginning at 0:0"`. The first two are free text; the third contains a colon, and is therefore read as a metadata entry (Chapter 5).

2.3 Signal files and storage formats

A signal file holds the samples of the signals of one signal group, as a sequence of frames (Section 2.5): the samples of each frame are stored one signal after the other. The *storage format* of the group says how each sample is encoded. Table 2.3 summarizes the formats; Chapter 15 gives their exact layout.

The last column is the value that marks an invalid (missing) sample in the file: the most negative value of the format. The library returns an invalid sample, whatever its format, as `WFDB_INVALID_SAMPLE` (-32768), and the high-level readers give it as NaN in physical units. A file in format 212, 310 or 311 may end with a partial group of samples. FLAC files are read and written by WFDB-x without libFLAC: the files it writes decode to the same samples as those of the C library, but their compressed bytes differ.

Three fields of the signal line change how a signal file is read: the *byte offset* skips a prolog at the beginning of the file; the *skew* discards the first samples of a signal, to align it with the others

Table 2.3 The storage formats (summary).

Format	Encoding	Invalid sample
0	no signal file: a null signal, whose samples are all invalid	(all)
8	8-bit first differences, from the initial value of the header	—
16	16-bit two's complement, least significant byte first	-32768
24	24-bit two's complement, least significant byte first	-2^{23}
32	32-bit two's complement, least significant byte first	-2^{31}
61	16-bit two's complement, most significant byte first	-32768
80	8-bit offset binary (128 is subtracted from each byte)	-128
160	16-bit offset binary, least significant byte first	-32768
212	12-bit two's complement, two samples packed in 3 bytes	-2048
310	10-bit two's complement, three samples packed in 4 bytes (two 16-bit words)	-512
311	10-bit two's complement, three samples packed in a 32-bit word	-512
508, 516, 524	FLAC-compressed, with 8, 16 or 24 bits per sample (at most 8 signals per file, all with the same number of samples per frame)	-128, -32768, -2^{23}

(the skipped samples are counted in the checksum); and the *block size* makes the file be read in blocks of that size.

2.4 ADC units and physical units

A signal file stores samples as integers, in *ADC units* (adu), the output of the analog-to-digital converter. The header gives, for each signal, the values that relate them to *physical units*, such as millivolts:

gain g, the number of ADC units per physical unit (0: the signal is uncalibrated);

baseline b, the ADC value that corresponds to 0 physical units;

units the physical units, such as `mV` (unspecified if the header does not give them);

ADC resolution r, in bits, and **ADC zero** z, the value at the middle of the ADC's range. Together they give the range of the ADC: from $z - 2^{r-1}$ to $z + 2^{r-1} - 1$.

The conversions are those of the C library, and of the high-level functions:

- From ADC units to physical units (`aduphys(s, a)`, and `read_record` with physical units): $v = (a - b)/g$, as a floating-point number. An invalid sample becomes NaN in the high-level readers.
- From physical units to ADC units (`physadu(s, v)`, and `write_record` with physical units): $a = \mathrm{round}(vg) + b$, where $\mathrm{round}(x)$ is $\lfloor x + 0.5 \rfloor$ for $x \geq 0$ and $\lceil x - 0.5 \rceil$ for $x < 0$ (halves are rounded away from zero). In `write_record`, NaN becomes an invalid sample.
- `adumuv(s, a)` converts a ADC units to microvolts, as $\mathrm{round}(1000\,a/g)$, and `muvadu(s, v)` converts v microvolts to ADC units, as $\mathrm{round}(g\,v/1000)$. Neither uses the baseline; both take the physical units to be millivolts.

In these functions, g and b are those of input signal `s` of the open record. WFDB-x assumes no gain: for an uncalibrated signal (gain 0) or a signal number that is not that of an input signal, the four functions report an error (`aduphys: signal 0 is uncalibrated`, `aduphys: invalid signal number 5`) and return NaN (`aduphys`) or `WFDB_INVALID_SAMPLE` (the other three); the C library assumed a gain of 200 (Section 4.1).

2.5 Frames, samples per frame and multi-frequency records

A *frame* is the set of samples of all the signals that belong to one sampling interval of the record. The sampling frequency of the record line is the frequency of the frames. Usually each signal has one sample per frame. A signal sampled at an integer multiple n of that frequency has n samples per frame (the x*n* suffix of its format), and its own sampling frequency is n times that of the frames. A record with such signals is a *multi-frequency* record. For example, the test record `tests/data/mfreq.hea`:

```
mfreq 3 100 600
mfreq.dat 16 200/mV 16 0 138 -16869 0 slow
mfreq.dat 16x2 200/mV 16 0 218 5410 0 medium
mfreq.dat 16x4 200/mV 16 0 219 -15996 0 fast
```

has 600 frames at 100 frames per second; the signals are sampled at 100, 200 and 400 Hz, and each frame of its signal file holds $1 + 2 + 4 = 7$ samples.

The library reads such a record in one of two modes, set by `setgvmode`:

- **Low resolution** (`WFDB_LOWRES`, the default): `getvec` returns one sample per signal per frame. The samples of a signal that has more than one per frame are averaged: their mean is rounded to the nearest integer, halves upward, and is invalid if one of them is invalid. `sampfreq` returns the frame frequency.
- **High resolution** (`WFDB_HIGHRES`): `getvec` returns as many sample vectors per frame as the largest number of samples per frame (4 for `mfreq`). Each holds the next sample of each signal that has that many samples per frame; the samples of the other signals are repeated. `sampfreq` returns the frame frequency times that number (400 Hz for `mfreq`), and times (Section 2.8) count these sample vectors.

`WFDB_GVPAD`, which can be combined with either mode, replaces each invalid sample by the previous valid sample of the same signal. `getframe` returns all the samples of a frame. In the high-level API, `read_record` reads in low resolution by default, and in high resolution with its `highres` option; its `pad` option is `WFDB_GVPAD`. For `mfreq`, the first frame is read as `138 1459 499` in low resolution (1459 is the mean of 218 and 2699, rounded up), and as these four rows in high resolution:

```
138   218    219
138   218  -1316
138  2699   1120
138  2699   1973
```

The library can also resample all the signals of a record to another frequency, by linear interpolation (`setifreq`).

2.6 Multi-segment records

A multi-segment record is a sequence of ordinary records, its *segments*, read as if their signals were continuous. Its header (the *master header*) has a record line whose record name is followed by `/` and the number of segments, and then one segment specification line per segment: the record name of the segment and its length in samples. The segments' own headers and signal files are found in the directory of the master header (Section 3.3.1). A segment cannot itself be a multi-segment record (`init: record REC cannot be nested in another multi-segment record`), and the sampling frequency must be the same in all segments. If the record line gives a number

of samples that differs from the sum of the segment lengths, the library warns (`warning (init): in record REC, stated record length (N) ...`).

In a *fixed-layout* record, every segment has the same signals, in the same order, with the same gains and baselines (their storage formats can differ). The test record `tests/data/multi.hea` is one: two copies of record `100s` with a null record between them.

```
multi/3 2 360 45000
100s 21600
null 1800
100s 21600
```

In a *variable-layout* record, the signals may differ from segment to segment. Segment 0, of length 0, is then the *layout segment*: its header, whose signal file names are ~, gives the signals of the record, with their gains, baselines and units. When the record is read, the signals of each segment are rearranged, shifted and rescaled to the gains and baselines of the layout (the results rounded to the nearest integer, halves upward); a signal absent from a segment reads as invalid samples. If a segment's gain is larger than that of the layout, the library warns (`sigmap_init: loss of precision in signal N in segment SEG`). A segment whose record name is ~ is a *null segment*: it has no files, and all its samples are invalid. The test record `tests/data/vlay.hea` and its layout segment `vlay_0.hea`:

```
vlay/4 3 250 3201
vlay_0 0
fmt16 1500
~ 200
fmt212 1501
```

```
vlay_0 3 250 0
~ 0 400(10)/mV 12 0 0 0 0 sig0 fmt16
~ 0 400/uV 12 0 0 0 0 sig1 fmt212
~ 0 100/mmHg 16 0 0 0 0 sig2 fmt16
```

2.7 Annotation files

An annotation file holds a sequence of *annotations*, each of which labels a point of the record, such as a beat. It is the file *record*.*annotator*. Each annotation has the fields of Table 2.4, those of the C structure `WFDB_Annotation`.

Table 2.4 The fields of an annotation.

Field	Meaning
`time`	The time of the annotation, in sample intervals from the beginning of the record (a 64-bit integer).
`anntyp`	The *annotation code*, such as `NORMAL` (1) or `PVC` (5). The codes 1 to `ACMAX` (49) are defined; 59 to 63 are the control codes of the MIT format (below), not annotation codes. Each code has a *mnemonic*, such as `N` or `V`, and a description (`annstr`, `anndesc`); Chapter 15 lists them.
`subtyp`	The subtype, a signed 8-bit integer (-128 to 127).
`chan`	The channel (signal) number, an unsigned 8-bit integer (0 to 255).
`num`	The annotator number, a signed 8-bit integer.
`aux`	Optional auxiliary data, up to 255 bytes, such as the rhythm (`N` of a `RHYTHM` annotation, or the text of a `NOTE`. In the library functions, its first byte is its length, as in C.

The high-level `read_annotations` returns these fields as arrays: the times (`samples`), the codes, their mnemonics (`symbols`), the subtypes, the channels, the annotator numbers and the aux strings.

Annotation files are in one of two binary formats, which the library tells apart when it opens a file (with a warning, `file appears to be in AHA format`, if the caller asked for the other):

- **MIT format**, the usual one: each annotation is a 16-bit word, least significant byte first, whose 6 high bits are the code and 10 low bits the interval from the previous annotation, followed by optional words for the fields that change (pseudo-codes `SKIP` for a long interval, `NUM`, `SUB`, `CHN` and `AUX`); a word of 0 ends the file.
- **AHA format**: 16 bytes per annotation, as in the distribution files of the AHA database.

An annotation file is in *canonical order* when its annotations are sorted by time, then annotator number, then channel. An output file that is not is sorted when it is closed (Section 4.3). When a program has set a time resolution with `setafreq` (as `wrann` does), the library writes it at the beginning of the file, as a `NOTE` annotation whose aux string is `## time resolution: F`. Annotation files can also begin with *modification labels*, `NOTE` annotations at time 0 that redefine the mnemonics and descriptions of codes; the library reads them when it opens the file, so `getann` does not return them.

2.8 Times and dates

Times are counted in *sample intervals* from the beginning of the record (in high resolution, in intervals of the fastest signal; Section 2.5). The library function `strtim` converts a time string to a number of sample intervals, and `timstr` and `mstimstr` convert back; the tools take their time arguments (`-f`, `-t`, ...) as `strtim` strings, and the high-level readers accept them for `sampfrom` and `sampto`.

2.8.1 Time strings

`strtim` first skips spaces, tabs, line feeds and carriage returns, and then reads the string according to its first character, as in Table 2.5. In the table, f is the sampling frequency (the one set by `setifreq`, if any), and a number is rounded to an integer by adding 0.5 and truncating toward zero.

A string that cannot be read gives 0, which is also a valid result (`strtim("0")`). The high-level readers make the value of a time of day positive, so that it is the number of the sample at that time. For record `100s` (360 Hz), `"0:10"` and `"10"` are 3600, `"s123"` is 123, `"1:02:03.5"` is 1340460, `"e"` is 21600, and `"[00:00:10]"` is -3600.

WFDB-x assumes no sampling frequency: when none is known (no record is open, or its header gives none, and none was set with `setsampfreq`), `strtim` reports `strtim: sampling frequency is not specified` and returns 0 for every string that needs one, that is, for all the strings of Table 2.5 except those that begin with `s`, `e`, `i` or `o`, and those that begin with `c` when there is no counter frequency. The function `wfdbx_time_needs_freq(string)` returns 1 if `strtim` needs the frequency for a string, and 0 otherwise. The C library used 1 sample per second instead (Section 4.1).

2.8.2 Formatted times

`timstr(t)` converts a time t to text:

- if $t > 0$, or if the record has no base time or date, as an elapsed time $|t|/f$, truncated to whole seconds: `h:mm:ss`, or, under one hour, `mm:ss`, right-aligned in 8 characters (`" 0:10"` for 10 seconds);

Table 2.5 The time strings that `strtim` accepts.

String	Value (sample intervals)
s*n*, as `s123`	Sample number n.
`e`	The end of the record: its number of samples (times the number of samples per frame of the fastest signal, in high resolution); 0 if it is not known.
`i`	The time of the next sample to be read from the input signals.
`o`	The time of the next sample to be written to the output signals.
c*x*, as `c350.5`	Counter value x: $(x - x_0)\, f/f_c$, with the counter frequency f_c and base counter x_0 of the header. With no counter frequency, x is read as a sample number.
f*n*, as `f100`	Frame number n: nf/f_0, where f_0 is the frame frequency (in high resolution, the number of the first sample of that frame).
[*time*], as `[13:06:00]`	A time of day: the value is the *negated* number of sample intervals from the base time of the record to that time, so that it is 0 or negative.
[*time d*], as `[8:00:00 1]`	A time of day d days after the base date.
[*time date*], as `[12:00:00 1/3/1992]`	A time of day on a date (`dd/mm/yyyy`).
s, as `143` or `14.875`	An elapsed time of s seconds: sf.
m:*s*, as `2:14.875`	m minutes and s seconds.
h:*m*:*s*, as `4:02:01`	h hours, m minutes and s seconds.

- if $t \leq 0$ and the record has a base time or date, as the time of day at $-t$ sample intervals after the base time, in brackets: `[hh:mm:ss dd/mm/yyyy]`, with the date if the record has a base date (or the number of days after the base time if it has none).

`mstimstr(t)` does the same with milliseconds (`" 0:10.003"`, `[12:34:56.789 31/12/1999]`). Without a sampling frequency, both report `timstr: sampling frequency is not specified` (or `mstimstr: . . .`) and return the time as a sample number, `s` followed by $|t|$, which `strtim` reads back.

2.8.3 Base time and dates

The base time and date of a record come from its header, or from `setbasetime`, which takes a string `hh:mm:ss`, with optional milliseconds and an optional date `dd/mm/yyyy` after a space (an empty string sets the current local date and time). `strdat` and `datstr` convert between dates `dd/mm/yyyy` and day numbers. The high-level readers give the base time as text (`base_time`, as `timstr(0)` gives it, to the second), as a number of milliseconds after midnight (`base_time_ms`), and the base date as year, month and day (`base_date`).

2.9 Calibration files

A calibration file lists, for types of signals, the calibration pulses with which they were recorded and their customary plotting scale. Each line that is not a comment (`#`) has six fields:

```
DESC<tab>LOW HIGH TYPE SCALE UNITS
```

`DESC` is a signal type, matched against the beginning of a signal's description (`*` matches any signal); `LOW` and `HIGH` are the levels of the calibration pulse in physical units (`-` for `LOW`: the signal is AC-coupled; `-` for `HIGH`: not defined); `TYPE` is `sine`, `square`, `sawtooth` or `undefined`; `SCALE` is in physical units per centimeter; and `UNITS` are the physical units. For example:

```
ECG - 1 sine 1 mV
```

```
BP  0 100 square 100 mmHg
```

`calopen(name)` reads a calibration file into the library's calibration list (emptied first, unless the name begins with `+`); without a name, it reads the file named by the environment variable `WFDBCAL`, or `wfdbcal`. The file is found as any other file (Chapter 3). `getcal` finds the first entry for a signal's description and units, `putcal` adds one, and `newcal` writes the list to a file. The file `tests/data/wfdbcal` is the default calibration file of the C package.

2.10 EDF and EDF+ records

The library reads EDF and EDF+ files (and BDF files, with 24-bit samples) directly, as records. A record name with an extension other than `.hea`, such as `100s.edf`, is opened as an EDF file: its header is read from the EDF header, and its samples from the data records that follow. Each EDF data record is one frame, so a signal has as many samples per frame as it has samples per data record, and the frame frequency is the inverse of the duration of a data record. The signals are in format 16 (24 for BDF). The gain of a signal is $(d_{\max} - d_{\min})/(p_{\max} - p_{\min})$, from its digital and physical ranges (0 if the physical range is empty), its baseline is $d_{\max} - p_{\max} \times \text{gain}$ rounded to the nearest integer, and its ADC zero is the middle of the digital range. The record's base time and date are the start time and date of the EDF file.

Because frames are data records, the library opens an EDF file in high resolution (Section 2.5): the sample intervals are those of the fastest signal, and `sampfreq` gives its frequency. A program that sets another mode reads one sample per data record per signal; the high-level `read_record`, which sets the mode, returns every sample of an EDF file only with its `highres` option.

The command-line tools `edf2mit` and `mit2edf` convert EDF files to WFDB records and back, and `rdedfann` prints the annotations of an EDF+ file in the format of `rdann`.

CHAPTER 3

Where files are read and written

WFDB-x reads and writes a file exactly where its name says, and nowhere else. A full path, a ~ path or a URL is used as given; any other name is read from the *input directory* and written to the *output directory*, which are both the current directory unless they are set. Nothing is searched, and URLs are opened only when network access has been turned on. This differs from the C library, which searched a list of places, the WFDB path, for every file it read (see Section 4.2 for the reasons); legacy mode restores the search (Section 4.12).

3.1 The input and output directories

Table 3.1 shows the three settings that decide where files are read and written. Each can be set by a library function, which has the same name in every language, and by an environment variable; a call of the function overrides the environment variable.

Table 3.1 The settings that locate files.

Setting	Functions	Environment variable	Default
input directory (a directory, a ~ path, or a URL)	`wfdbx_set_input_dir`, `wfdbx_get_input_dir`	`WFDBX_INPUT_DIR`	the current directory
output directory (a local directory or a ~ path)	`wfdbx_set_output_dir`, `wfdbx_get_output_dir`	`WFDBX_OUTPUT_DIR`	the current directory
network access	`wfdbx_set_network`, `wfdbx_get_network`	`WFDBX_NETWORK`	off

- `wfdbx_set_input_dir(d)` sets the input directory, and returns the previous one (or none, if none was set). `d` is a local directory, a ~ path, or the URL of a directory, such as `https://physionet.org/files/mitdb/1.0.0/`; reading from a URL also needs network access. None (null, `NULL`, ..., depending on the language) or an empty string restores the default: the value of `WFDBX_INPUT_DIR` if it is set, and otherwise the current directory. `wfdbx_get_input_dir()` returns the input directory, or none.
- `wfdbx_set_output_dir(d)` sets the output directory in the same way, and returns the previous one; the default is the value of `WFDBX_OUTPUT_DIR`, or the current directory. The output directory must be local: a name that begins with a URL scheme, such as `https://` or `ftp://` (given to the function, or in the environment variable), is refused with the message `wfdbx_set_output_dir: the output directory must be a local directory`, and the setting does not change.
- `wfdbx_set_network(on)` turns network access on (1, or true) or off (0, or false), and returns the previous setting (1 or 0); none or a negative number restores the default: on if `WFDBX_NETWORK` is set to a nonzero number (such as `1`), off otherwise. `wfdbx_get_network()` returns 1 or 0.

In Kotlin, these functions are also available under camelCase names (`wfdbxSetInputDir`, ...). The settings apply to the whole library: to the high-level functions, the library functions and the tools, which use the environment variables, also for the files that they open themselves (Section 3.3.2). `wfdbquit` does not reset them.

The high-level functions also take these settings as options, which apply for the duration of one call, after which the previous settings are restored: the readers take an input directory (`input_dir`) and a network permission (`network`), and the writers an output directory (`dir`). Each of these options is optional; `input_dir` applies to relative record names only, and `dir` to plain record names (Section 3.2). In Kotlin the options are `inputDir`, `network` and `dir`.

The C functions setwfdb and getwfdb. In WFDB-x, `setwfdb` and `getwfdb` set and return the input directory. `setwfdb` splits its argument as the C library splits a WFDB path (at spaces, semicolons, and colons that are not part of a URL), and accepts a single directory (`setwfdb(".")`); a path of several directories is an error (`setwfdb: WFDB-x reads from a single input directory; use wfdbx_set_input_dir (or legacy mode for the WFDB search path)`), and nothing changes. A directory whose name contains a space must therefore be set with `wfdbx_set_input_dir`, which takes the name as it is. `getwfdb` returns `.` when no input directory is set. `resetwfdb` restores the directory set by the last `setwfdb`.

3.2 Record names

A record name is given without the extension `.hea` of its header (`100s`, not `100s.hea`; a trailing `.hea` is removed by the functions that read a record and by `write_record`). It is one of:

- a full path, absolute (`/data/mitdb/100`) or in the home directory (`~/Documents/100s`), used as given;
- a relative path (`100s`, `data/100s`), read from the input directory if one is set, otherwise from the current directory, and written to the output directory if one is set, otherwise to the current directory (missing directories are created);
- a URL (`https://physionet.org/files/mitdb/1.0.0/100`), read only if network access is on (Section 3.4); a URL cannot be written. Only `http://` and `https://` URLs can be read: a name with another URL scheme, such as `ftp://` or `sftp://`, is an error, and is never read as a local file name.

A writer given a record name with a path (`out/rec`) creates the files in that directory, and the header names the record and its signal file without the path (`rec`, `rec.dat`), so that the record opens under any path that leads to it (Section 3.3). With the `dir` option of a writer, the record name must be a plain name (`rec`), without a directory: the files are created in `dir`, which must exist, and the record opens from that directory under its plain name. The characters allowed in record names are given in Section 2.1.

3.3 How a name is resolved

The name of a file is the record name followed by a dot and the file type (`100s.hea`, `100s.atr`), or a file name given by itself (a signal file named in a header, a calibration file). It designates exactly one location, as Table 3.2 shows.

- A leading `~` or `~/` is replaced by the home directory (the environment variable `HOME`, or the home directory of the current user if it is not set; on MS-Windows, `USERPROFILE`), and, on POSIX systems, `~user` by the home directory of that user. The input and output directories may themselves be `~` paths.

Table 3.2 Where a name is read from and written to.

The name is	Read from	Written to
an absolute path: `/data/100`; on MS-Windows also `C:\data\100`, `C:/data/100` or `\\server\share\100`	that path	that path
a home path: `~/data/100`, `~user/data/100`	the home directory (expanded by WFDB-x)	the same
a URL: `http://...` or `https://...`	that URL, if network access is on	(not possible)
a name with another URL scheme: `ftp://...`, `sftp://...`, `file://...`	(not possible: an error)	(not possible)
`-` (as a record name)	the standard input	the standard output
anything else: `100`, `mitdb/100`, `../other/100`	the input directory followed by the name	the output directory followed by the name

- The input or output directory and the name are joined with one directory separator (`/` in a URL). `..` means one level up; the C library refused every name that contains `...`
- Nothing else is tried: a file that is not at that location is not found, and the operation fails (for example, `init: can't open header for record 100`). The only other name tried at the same location is the one whose file type is cut to its first three characters, as in C.
- When an output file cannot be created because a directory of its path does not exist, the missing directories are created, as in C.

A record written under a path, such as `out/rec`, gets its files in `out/` (under the output directory, if it is relative). Only the record's own name, the last component, must be made of the characters of record names (Section 2.1); the directories may contain any character. The header names the record, its signal files and its segments relative to its own directory (`rec`, `rec.dat`), which is where they are read from; the C library wrote the names as given, with their path (`out/rec` and `out/rec.dat` in `out/rec.hea`), as legacy mode does, so that such a record could not be read back by finding its files relative to its header.

3.3.1 Files named in a header

The files that a header names, the signal files of a record and the headers of the segments of a multi-segment record, are found relative to the directory of the header, as they are written in it (`100s.dat`, or `data/100s.dat`): on disk, or at the URL from which the header was read. A file missing there is an error, not a search elsewhere. Such a name must stay inside the header's directory: a header that names a file with a `..` component, an absolute path, a `~` path or a URL (or, on MS-Windows, a drive letter) is rejected, with the message

```
init: file NAME named in the header of record REC is not in the header's directory
```

The signal file name `-` (the standard input) is accepted, and the null segment `~` of a multi-segment record keeps its meaning. For a header read from the standard input, which has no directory, the files it names are resolved as any relative name, under the input directory. This rule keeps a downloaded or crafted header from making the library read other local files or contact another server.

3.3.2 Files that a tool opens itself

A few tools open a file themselves, by the name given on the command line, instead of through the library. Such a file is found, or created, exactly where the library finds, or creates, a file of that name: a relative name under the input directory (a file that the tool reads) or the output directory (a file that it creates, whose missing directories under the output directory are created), a ~ name expanded, and an absolute name as given; -, where the tool takes it for the standard input or output, keeps that meaning. A URL is not accepted for these files, also when the input directory is a URL: the tool reports that it cannot read, or create, the file, as for a file that does not exist. Messages name the file as it was given. Table 3.3 lists these files.

Table 3.3 The files that the tools open themselves.

Tool	Reads	Creates
`wrsamp`, `edf2mit`, `wav2mit`, `a2m`, `ad2m`	the `-i` file	
`ahaecg2mit`	the input files (`.ecg`, `.txt`)	
`xform`	the script of `-S`	
`wfdbdesc`	the first signal file, to compute the length of a record whose header gives none: in the header's directory, where the library reads it	
`mit2wav`		the WAV file (`-o`), completed where the library wrote its samples
`mit2edf`		the EDF file (`-o`, or *RECORD*`.edf`)
`snip`		the master header of a multi-segment excerpt (`-m`)
`parsescp`		the `.des`, `.ecg`, `.key` and `.txt` files

The library functions `wfdbx_input_path` and `wfdbx_output_path` (Section 13.11) return these locations, for any program that opens a file itself. The C tools opened these files relative to the current directory, so that, with an input or output directory set, a tool read its records from one place and its own files from another (`mit2wav` left the WAV header in the current directory and the samples, without a header, in the output directory). Without an input or output directory, nothing changes, except that a ~ name is expanded and that `wfdbdesc` looks for the signal file of a record in another directory in that directory.

3.3.3 Legacy mode

In legacy mode (Section 4.12), files are found as by the original reference build of the C library, which has no network support: an input file is searched for in each entry of the WFDB path in turn, given by the `WFDB` environment variable or `setwfdb` (by default `. /usr/local/database http://physionet.org/physiobank/database`), with the `%r` substitution of the record name and `@file` indirect paths; absolute names are joined to the entries of the path (an empty entry, as in `WFDB=";"`, makes names be used as given); names that contain `..` are refused, and URLs are not opened. The input directory and `WFDBX_NETWORK` are not used. The output directory is used, as a plain prefix of every output name, absolute names included. The tools open the files that they read themselves (Section 3.3.2) as named, relative to the current directory, as the original tools do, and the output directory, if one is set, is a plain prefix of the names of the files that they create themselves, as of every other output file.

3.4 Network access

Network access is off by default. A URL is opened only when it has been turned on, by `wfdbx_set_network(1)`, by `WFDBX_NETWORK=1`, or by the `network` option of a high-level reader. Otherwise the library says once, the first time it is asked to open a URL:

```
WFDB-x: network access is disabled; set WFDBX_NETWORK=1 or call wfdbx_set_network(1) to open URL
```

and the file is not found. The files of PhysioNet are reached in the same way as any other remote files: by setting the input directory to the URL of the database, `https://physionet.org/files/`*database*`/`*version*`/`, and turning network access on. No URL is built in: WFDB-x never contacts a server it was not told to contact. For example, to print the first 10 seconds of record 100 of the MIT-BIH Arrhythmia Database:

```
export WFDBX_INPUT_DIR=https://physionet.org/files/mitdb/1.0.0/
export WFDBX_NETWORK=1
rdsamp -r 100 -t 10
```

or, in Python:

```
import wfdbx
wfdbx.wfdbx_set_network(1)
wfdbx.wfdbx_set_input_dir("https://physionet.org/files/mitdb/1.0.0/")
rec = wfdbx.read_record("100", sampto="0:10")
```

or, for one call only:

```
rec = wfdbx.read_record("100", input_dir="https://physionet.org/files/mitdb/1.0.0/",
                        network=True)
```

Remote files can only be read. Each is downloaded in full the first time it is opened, and kept for the rest of the process; a download that fails is not kept, so the next time the file is opened it is requested again (a server that was unavailable may have recovered). Redirects are followed, and an HTTP status of 400 or more is a failure. A download gives up if connecting takes 30 seconds, or if no data arrive for 30 seconds; there is no limit on its total time. Each language downloads with its own means:

- Python: its standard `urllib`;
- C++ and Rust: the system `curl` program, which they run without a shell (a missing `curl` is reported when a URL is opened);
- Julia: its standard `Downloads`;
- R and MATLAB/Octave: `curl` when it is installed, and otherwise R's `url()` or MATLAB's `webread` and `websave`;
- Kotlin: `java.net.HttpURLConnection` on the JVM and Android, and `NSURLSession` on iOS.

Only `http://` and `https://` URLs can be read. A name that begins with another URL scheme (`ftp://`, `ftps://`, `sftp://`, `file://`, ...) is refused, with or without network access, with a message the first time:

```
WFDB-x: can't open ftp://example.org/db/100.hea: only http:// and https:// URLs
can be read (mount the server as a local folder, or download the files)
```

(on one line). The same holds for an input directory with such a scheme, a file named in a header, and the name of an output file; the files that a tool reads or creates itself are refused as URLs are (Section 3.3.2), and an output directory with a URL scheme is refused (Section 3.1). Such a

name is never read as a local file name (the directory `ftp:` is never looked in). A scheme is a letter followed by at least one letter, digit, `+`, `-` or `.`, then `://`, so a one-letter prefix, as in `c://x` (like a drive letter), is not one. The records of a server reached by FTP or SFTP can be read once the server is mounted as a local folder, by their local paths, or after the files are downloaded:

```
sshfs user@example.org:/data/db ~/mnt/db       # SFTP (or: rclone mount)
rdsamp -r ~/mnt/db/100 -t 10
```

In legacy mode, every name containing `://` is refused, without a message, as by the original library built without network support.

A header read from a URL can be read but not changed by the metadata functions, except into a local directory (Section 5.4).

3.5 The standard input and output

The record name `-` stands for the standard input, when a file is read, and for the standard output, when one is written, whatever the type of the file: a header, an annotation file or a signal file. A signal file named `-` in a header is the standard input (or output). The tools follow the C tools in this, so that, for example, `wrsamp` reads its text input from the standard input unless `-i` is given. The metadata functions refuse the record name `-`.

3.6 Environment variables

Table 3.4 lists every environment variable that WFDB-x reads. The library reads each one when it needs it.

Table 3.4 The environment variables used by WFDB-x.

Variable	Meaning
`WFDBX_INPUT_DIR`	The input directory (Section 3.1): a local directory, a `~` path or a URL. Not used in legacy mode.
`WFDBX_OUTPUT_DIR`	The output directory: a local directory or a `~` path.
`WFDBX_NETWORK`	Network access is on if this is a nonzero number (`WFDBX_NETWORK=1`). Not used in legacy mode.
`WFDB`	The WFDB path, used only in legacy mode. Outside legacy mode it is ignored, and if it is set the library says so once: `WFDB-x: the WFDB environment variable is ignored; set WFDBX_INPUT_DIR instead (or use legacy mode)`. In legacy mode, `setwfdb` sets it (and sets `WFDBCAL`, `WFDBANNSORT` and `WFDBGVMODE` to their defaults if they are not set), as the C library does, so that the programs it starts inherit them.
`WFDBANNSORT`	If it is `0`, an output annotation file that is not in canonical order is not sorted when it is closed; the library then prints the advice `Use the command: sortann ...` (Section 4.3). The `sort` option of the high-level `write_annotations` decides for its call, whatever this variable says. (`WFDBNOSORT`, named by an old manual page of `sortann`, is not used, as in C.)
`WFDBCAL`	The name of the calibration file that `calopen` reads when it is given no name; `wfdbcal` if it is not set (Section 2.9).
`WFDBGVMODE`	The `getvec` mode used when `setgvmode` is given a negative mode, and by `sigamp` when its options do not set the mode: `0` for low resolution, `1` for high resolution, plus `2` to replace invalid samples by the previous valid ones (Section 2.5).

Variable	Meaning
WFDB_FLAC_CHECK_MD5	If it is set to a nonempty value, the MD5 signature of each FLAC signal file that is read is verified when the file is closed; a mismatch gives `isigclose: warning: incorrect MD5 hash in FLAC input`.
HOME, USERPROFILE	The home directory, for names that begin with ~ (USERPROFILE on MS-Windows).
PATH	Where the system `curl` program is found (C++, Rust, and R and MATLAB/Octave), and, in legacy mode, the `sortann` program that the library runs to sort an annotation file (C++ and Rust look first in the directory of the running program).
TZ	On POSIX systems, the time zone of the local time, which the library uses where the C library does: for the current date and time (`setbasetime` with an empty string, and `mit2edf` for a record without a start time).
WFDBX_DEBUG	(R and Kotlin tools.) If it is set, a tool that stops on an internal error (reported as `wfdbx: error: . . .`) also prints the call stack.
WFDBX_STDIN	(R and MATLAB/Octave.) The name of a file that is read as the standard input: in R, for the tests; in MATLAB/Octave, also set by the MATLAB driver `matlab/bin/wfdbx_matlab.sh`, since MATLAB cannot read its standard input.
WFDBX_STDOUT, WFDBX_STDERR	(MATLAB/Octave.) Files to which the standard output and error are written, set by the MATLAB driver, which copies them to the real standard output and error when MATLAB exits.

No environment variable turns legacy mode on: it is turned on for a tool by its `--legacy` option, and in a program by `wfdbx_set_legacy(1)` or the `legacy` option of a high-level function (Section 4.12). The other variables that the C library reads are not used by WFDB-x: those of its libcurl code (WFDB_PAGESIZE, WFDBPASSWORD, CURL_CA_BUNDLE, WFDB_NET_DEBUG) and those that tune libFLAC's encoder (WFDB_FLAC_COMPRESSION_LEVEL, WFDB_FLAC_BLOCK_SIZE, and the other WFDB_FLAC_. . . variables).

The conformance suite runs every case with LC_ALL=C and TZ=UTC, and without the variables WFDB, WFDBCAL, WFDBANNSORT, WFDBGVMODE, WFDBX_INPUT_DIR, WFDBX_OUTPUT_DIR and WFDBX_NETWORK, so that its results do not depend on them.

CHAPTER 4

Differences from the original package

WFDB-x reproduces the WFDB Software Package 10.7.0 exactly, with the deliberate exceptions described in this chapter. It has no built-in signal parameters; it finds files only where they are named, and opens URLs only when network access is on; it sorts out-of-order annotation files itself, without losing annotations; it never writes a signal file over a signal file it is reading; its tools report write errors in their exit status; its `getinfo` finds the record line of a header as the header parser does; and it gives a defined result for input on which the C package crashes. It also adds features that the C package does not have.

These exceptions are the same in all seven implementations, and are tested like everything else: the reference results of the conformance suite come from the C package with the patch `tests/reference/divergences.patch` applied, which is the precise definition of the differences. The original behavior remains available in every case, on request, in *legacy mode* (Section 4.12). The file `docs/DIVERGENCES.md` gives the reasons for each difference in more detail, and Sections 4.9 to 4.11 list the corrections of versions 1.2.0, 1.2.2 and 1.2.3.

4.1 No built-in sampling frequency, gain, ADC resolution or units

The C package falls back on built-in values for four parameters of a signal whenever a record or the user does not give them (Table 4.1). These values describe the equipment with which the first PhysioNet databases were recorded, and do not describe modern recordings. A silently assumed value corrupts every time, interval, resampled signal and physical value computed from it, and nothing in the output says that it was assumed. WFDB-x therefore never assumes any of these four parameters: it uses the values given in the record, or values given explicitly by the user, and reports an error when an operation needs a value that is not available. Where an output must still show missing units, it shows `nd` (“not defined”, a notation that the C package already used for empty units) rather than `mV`.

Table 4.1 The values that the C package assumes.

Parameter	Built-in value in C
sampling frequency	250 Hz (`WFDB_DEFFREQ`); 500 Hz in `parsecp`
gain	200 adu/mV (`WFDB_DEFGAIN`); 400 adu/mV for AHA files; 12.5 or 6400 adu/mV for WAV files
ADC resolution	12 bits (`WFDB_DEFRES`), or the storage width of the format
units	millivolts

Library.

- A header that does not give a sampling frequency leaves it unspecified: `sampfreq` returns 0 (C: 250), unless a frequency was set with `setsampfreq` before the record was opened (as in C).

- A header that does not give an ADC resolution leaves it unspecified (0); C inferred it from the format or assumed 12 bits. Missing units remain unspecified (`NULL`), as in C, but are never taken to be millivolts.
- The time conversions that need the frequency fail when it is unspecified: `strtim` reports `strtim: sampling frequency is not specified` and returns 0 for the time strings other than sample numbers (`s...`), `e`, `i` and `o` (and counter values when there is no counter frequency); `timstr` and `mstimstr` report the same error and return the time as a sample number (`s` followed by the number). C used 1 sample per second. The new function `wfdbx_time_needs_freq(string)` tells whether `strtim` needs the frequency for a string (Section 2.8).
- `setifreq` fails if the input frequency is unspecified (`setifreq: sampling frequency is not specified`).
- `aduphys`, `physadu`, `adumuv` and `muvadu` fail for an uncalibrated signal (gain 0) or an invalid signal number: they report the error and return NaN (`aduphys`) or `WFDB_INVALID_SAMPLE` (the others). C assumed a gain of 200, and a baseline of 0 for invalid signal numbers.
- The constants `WFDB_DEFFREQ`, `WFDB_DEFGAIN` and `WFDB_DEFRES` remain defined, for compatibility, but the library does not use them (except in legacy mode).

Tools. A tool that writes these parameters into a record requires them; a tool that needs a parameter missing from its input stops with an error only when the requested operation needs it:

- `wrsamp` requires `-F`, `-G` and the new option `-R BITS` when it writes a record (`-o`), and the new option `-U UNITS` for calibrated signals unless the input gives the units; `-G 0` declares the signals uncalibrated. `-G`, `-R` and `-U` take one value per signal (the last applies to the rest).
- `wav2mit` has the new options `-G GAIN` (required) and `-U UNITS` (required unless the gain is 0); `ad2m` and `ahaecg2mit` the new options `-F FREQ`, `-G GAIN`, `-R BITS` and `-U UNITS`, all required (`-U` unless the gain is 0).
- `a2m`, `rr2ann`, `rdann`, `ann2rr`, `sortann` and `m2a` take `-F FREQ` (new except in `rr2ann`), used when the record has no header or its header has no frequency, and required then when times are converted.
- `rdsamp` needs the frequency for time arguments, time columns and XML output, and the gain of each selected signal for physical units (`-p`); `sigamp`, `snip`, `wfdb2mat`, `mit2wav`, `wfdbtime`, `xform` and `mit2edf` need the frequency (and `mit2edf` the gain and ADC resolution of every signal), and so does `time2sec` with `-r` (since 1.2.2; C converted the time at 250 Hz, and WFDB-x until 1.2.1 printed a meaningless number, `-9223372036854775808`, with the exit status 0).
- `wfdbdesc` prints `not specified` for an unspecified frequency and ADC resolution, `uncalibrated` for a gain of 0 (C: `uncalibrated; assume 200`), and `adu/nd` for unspecified units; `sampfreq` prints 0 for a record whose header gives no frequency (C: 250); `parsecp` writes the frequency given in the SCP file (C always wrote 500 Hz).

The messages are `TOOL: sampling frequency of record RECORD is not specified` (followed by `(use -F)` for the tools that have that option), `TOOL: signal N of record RECORD is uncalibrated`, `TOOL: ADC resolution of signal N of record RECORD is not specified`, and `TOOL: the ... must be specified (-X)` for a missing required option. Part III gives the options of each tool.

High-level API. `write_record` requires the sampling frequency (greater than 0), the gains, the ADC resolutions, and the units of the calibrated signals; a gain of 0 declares a signal uncalibrated, which is allowed only for samples in ADC units. `read_record` reports an unspecified frequency as 0, and fails for time strings that need it; with physical units, it fails if a selected signal is

uncalibrated. `read_annotations` takes an optional sampling frequency, used, like the `-F` option of the tools, when the record has no header or its header gives none.

Reproducing the results of the original package. The results of the original tools are obtained by giving the values they assumed, explicitly. For example, `wrsamp` writes a record as the original did with `-F 250 -G 200 -U mV` and `-R` with the storage width of the output format (`-R 12` for format 212, `-R 16` for formats 16, 61, 160 and 516, `-R 8` for 8, 80 and 508, `-R 10` for 310 and 311, `-R 24` for 24 and 524, `-R 32` for 32); `wav2mit` with `-G 12.5 -U mV` for 8-bit files and `-G 6400 -U mV` for 16-bit files; `ad2m` and `ahaecg2mit` with `-F 250 -G 400 -R 12 -U mV`; the annotation tools with `-F 250` for a record without a frequency; and a library program calls `setsampfreq(250)` before it opens a record whose header gives no frequency. One difference remains by construction: a header written with `-U mV` says `200/mV` where the C tools wrote `200`.

What does not change. Values defined by a data format, not by an assumption about the signal (the record lengths and time offsets of the AHA database formats, and `md2a`'s default length); values read from the input (the sampling frequency and resolution of a WAV file, the parameters of an EDF file); and analysis settings that are documented options, such as the measurement windows of `sigamp`. `parsescp` writes a gain of 200 adu/mV because it rescales the SCP samples to units of 5 μV, so that value is exact, not assumed.

4.2 Files are found only where they are named

The C library found a file by searching the WFDB path, a list of places tried in order until the file was found, by default the current directory, `/usr/local/database` and, in builds with network support, PhysioNet. That was convenient, but:

- a search opens the first copy it finds, so a stale or partial copy in a directory earlier on the path is read instead of the intended one, and a signal file missing next to its header is replaced by a file of the same name found anywhere else on the path;
- whether a record is read from disk or downloaded, and from where, depends on the environment, not on the program, and a missing local file turns into a network request instead of an error;
- for every record it read, the library also asked the last entry of the path, PhysioNet, for an optional `.info` file when there was no local one, with the full local path of an absolute record name in the request;
- a header could make the library read any local file (`../../.ssh/id_rsa`) or contact a third-party server, through the names of its signal files;
- the default URL is no longer where PhysioNet serves its files.

WFDB-x reads and writes every file at the one location that its name designates, as Chapter 3 describes: under the input or output directory for a relative name, as given for an absolute path, a ~ path or a URL, and relative to the header for the files that a header names, which must stay in the header's directory. URLs are opened only when network access is on, and only `http://` and `https://` URLs can be read. A name containing `..` is accepted. `setwfdb` and `getwfdb` set and return the input directory, and the `WFDB` environment variable is used only in legacy mode. Reading PhysioNet records must therefore be asked for explicitly (Section 3.4); in return, every program reads exactly the files it names, the same on every machine.

The files that a tool opens itself, by the name given on its command line, rather than through the library (the `-i` file of `wrsamp`, the WAV file of `mit2wav`, and the others of Table 3.3), follow the same rules: the C tools opened them relative to the current directory, so that, with an input or output directory set, a tool read its records from one place and its own files from another. In

WFDB-x such a file is found, or created, where the library finds, or creates, a file of that name (Section 3.3.2); a URL is not accepted for these files. In legacy mode, the files that a tool reads itself are opened as named, and the output directory, if one is set, is a plain prefix of the names of the files that it creates itself.

4.3 Out-of-order annotation files are sorted in the library

When an annotation file written by the library is not in canonical order (Section 2.7), the C library sorted it when the file was closed by running the `sortann` program through the shell, unless the environment variable `WFDBANNSORT` was 0. The result then depended on which `sortann` was found on the `PATH`, and a record or annotator name with a space or a shell character broke the command.

WFDB-x sorts the file itself, in the same process. The file it writes is byte for byte the one that `sortann` writes, in the same order (by time, then annotation number, then channel; a later annotation with the same time, number and channel replaces the earlier one), with the same treatment of `NOTQRS` annotations (they are dropped) and of the time resolution, and the same encoding. The messages and status of the library are the same: `Rearranging annotations for output annotator ANN ...done!`, or the advice `Use the command: sortann -r REC -a ANN` when sorting is turned off or fails; only the messages that the separate `sortann` printed itself are gone. `WFDBANNSORT=0` turns sorting off, and the `sort` option of the high-level `write_annotations` decides for its call alone (sort, or leave the file as written), whatever `WFDBANNSORT` says.

Sorting does not lose annotations. The original `sortann` silently loses annotations: when an annotation at time 0, with annotation number 0 and channel 0, arrives after later ones, it is dropped, and so are the annotations later inserted directly behind it (input sample times 5, 0, 3 give a file that contains only 5), because the search for the insertion point mistakes the empty head entry of its list for an annotation. WFDB-x puts such an annotation at the beginning of the list, where it belongs (5, 0, 3 give 0, 3, 5). Nothing else about the sorting changes. This applies to the `sortann` tool and to the sorting done by the library.

> **Note** In legacy mode, the library runs `sortann` as the original does (with its own `sortann` tool, started with `--legacy`, without a shell), and both lose these annotations as the original does. Legacy mode should not be used to sort annotation files unless reproducing the original results, losses included, is the point: nothing in the output says that annotations were dropped.

4.4 An output signal file is never one of the input signal files

The files of records can be shared: the segments of a multi-segment record are other records, so that `xform -i vlay -o fmt16` reads `fmt16.dat` (a segment of `vlay`) and writes `fmt16.dat` (the output record `fmt16`). The original opens the output file, which truncates it, while still reading it: the input is destroyed, and what is read depends on how much of it the C library had read ahead, so that the result differs between platforms.

WFDB-x refuses to create an output signal file that is the same file as a signal file of the record open for reading (the same file, not the same name: links and different spellings of a path are recognized), with the message `osigfopen: output file NAME is also an input file` (`osigopen:` when the file is opened by `osigopen`); nothing is written. The check is in the library, so every tool and every program that uses the library functions is protected. The high-level `write_record` reads nothing: it closes any open record first, and simply replaces existing files of the same name. The check covers signal files only: for example, `snip -i 100s -n 100s` still writes the header of its

output record over that of its input record, in C and in WFDB-x alike; the output record needs a name of its own.

4.5 Write errors give a nonzero exit status

The original tools report a file that cannot be written (a full disk, for example) only through the library's messages, such as `osigclose: write error in signal group 0` or `oannclose: write error on annotation file atr`, and then exit with status 0; they say nothing at all when the standard output cannot be written, or when a file that a tool writes itself fails. The tools of WFDB-x follow the convention of the standard Unix tools: when an output file or the standard output cannot be written, the exit status is 1. The library's messages are unchanged; a standard output that cannot be written is reported as `PROGRAM: write error on standard output`, and a file that `snip`, `mit2edf` or `parsecp` writes itself as `PROGRAM: write error on FILE`. A tool that exits with another nonzero status for another reason keeps that status, and the tools keep running after a write error, as the original ones do.

The library counts the output files that it could not write, flush or close; `wfdbx_write_errors()` returns that count, and `wfdbx_note_write_error()` adds one, for a program that writes a file itself.

4.6 The record line of a header is found as the header parser finds it

The info strings of a header are its lines that begin with `#` after the record line (Section 2.2.3). The C package has two rules for finding the record line. The header parser (`readheader`) takes the first line whose first token (separated by spaces, tabs, carriage returns and line feeds) does not begin with `#`, so it skips empty and blank lines as well as comment lines; but `getinfo` takes the first line that does not begin with `#`. For a header whose first line is empty or blank, such as

```
(an empty line)
# Age: 40
rec 1 250 10
rec.dat 16 200 12 0 0 0 0 ECG
# Age: 63
```

the original `getinfo` takes the empty line for the record line, and returns `Age: 40`, a comment before the record line, as an info string, as well as `Age: 63`: `wfdbdesc` prints both as notes, and the `comments` of the record hold both.

Since version 1.2.2, the `getinfo` of WFDB-x finds the record line as `readheader` does, so that the info strings are the lines that the header parser leaves after the record line: `Age: 63` only. A header with no record line has no info strings. The info strings after the record line are found as before: every line whose first character is `#` (a comment between two signal lines included). This applies to `getinfo` itself, to the `comments` of `read_header` and `read_record`, and to the notes of `wfdbdesc`. The metadata functions (`read_metadata` and the others, and the `metadata` of `read_header`) follow the same rule in every mode, legacy mode included (Chapter 5). In legacy mode, `getinfo` follows the original rule. The conformance cases `edge-hdr-leading-*` check both (in the legacy suite, against the unmodified package).

4.7 Input on which the C package crashes

Some malformed or unusual input makes the C package crash, run out of memory, loop forever, or give a result that depends on the machine (reads of uninitialized memory). A crash cannot be reproduced, so WFDB-x gives each such input one defined, safe result, the same in all seven implementations, and legacy mode gives the same result. For example:

- a header with a negative or impossibly large number of signals (`n -1 250 10`), for which C tries to allocate memory for 4,294,967,295 signals, gives `init: unexpected EOF in header file for record n`, as C does for a header with fewer signal lines than it declares;
- an EDF file whose signals have 0 samples per data record, on which C crashes with a floating-point exception, reads as a record with no samples;
- a FLAC signal file with 3 or more samples per frame whose FLAC block size is not a multiple of them, for which C writes past the end of a buffer, is read correctly;
- `wfdb2mat` with a one-character record name, for which C overwrites the header of its input record, names its output record as for longer names (`mm` for `m`);
- `edf2mit` with a file shorter than the 256-byte fixed part of the EDF header prints `edf2mit: incomplete EDF header in FILE` and exits with status 2, creating no files;
- `signame -s N`, where `N` is the number of signals, prints `[INVALID]`, as its manual page says;
- `osigopen(record, NULL, 0)`, for which C reads before the start of an array, closes the output signals already open (unless the record name begins with `+`), reads the header (an error if it cannot be read), opens nothing and returns 0.

The complete list, with the C behavior and the WFDB-x choice for each, is the table "Undefined behaviors (not reproduced)" of `docs/KNOWN_C_BEHAVIORS.md`. The conformance suite checks most of these choices, with reference results recorded from WFDB-x instead of C.

4.8 Additions

These are new features, not changes of the original behavior, so they have no legacy counterpart.

- **The high-level API**: in every language, a few functions that read and write whole records and annotation files as arrays, in one call, with named options and errors reported in the language's way (Part II).
- **An output directory** (`wfdbx_set_output_dir`, `WFDBX_OUTPUT_DIR`, and the `dir` option of the high-level writers), in which output files are created, and an input directory (`wfdbx_set_input_dir`, `WFDBX_INPUT_DIR`); see Chapter 3.
- **A record checker**, `check_record(record, annotators)`, which checks a record's header, signal files and annotation files, and reports each problem it finds with a severity (error, warning or information): missing or inconsistent signal specifications, ADC resolutions and ranges that the storage format cannot hold, samples outside the ADC range, checksum and length mismatches, problematic names, inconsistent segments of multi-segment records, and annotations that are out of order, undefined or after the end of the signals. Its report, including the text of every message, is identical in all seven implementations. It looks for problems similar to those found by Benjamin Moody's separate C program `wfdbcheck`, but was written independently.
- **Fast reading and writing.** The high-level functions read and write whole blocks of samples, and `write_annotations` encodes a whole MIT annotation file at once. The multi-frame readers added to the library functions (`getframes` in Python, R and MATLAB/Octave, `getvecs` in C++, Rust, Julia and Kotlin) decode whole blocks as well, including multi-frequency and multi-segment records. The samples and files are exactly those of the frame-by-frame functions.
- **Signal names without surrounding spaces.** Some headers end a signal's description with spaces (`"ABP "`). The high-level readers give the names without white space at either end, and `read_record` finds a channel by that name; the library functions and the tools keep the descriptions as the C library does.

- **The header and the start time as numbers.** `read_header` returns the header of a record without opening its signal files: the sampling and counter frequencies, the base counter value, the number of samples, the signal specifications, the info strings, the metadata, and the start time and date. `read_header` and `read_record` give the start time as text (`base_time`), in milliseconds after midnight (`base_time_ms`, which keeps the milliseconds of a start time such as `14:00:03.500`), and the start date as year, month and day (`base_date`).
- **Write errors and frame checks in the high-level writers.** `write_record` and `write_annotations` raise an error (`write error on record REC`, `write error on annotator ANN for record REC`) when a file cannot be written completely, or when `putann` reports an error (version 1.2.2). `write_record` also refuses, before creating any file, frame sizes that are not integers of at least 1 (`samps_per_frame must hold integers >= 1`) and signals whose number of columns is not the number of samples in a frame (`signals has N columns, but samps_per_frame adds up to M`).
- **The high-level writers check every argument before touching any file, and leave the existing files intact after a failure** (version 1.2.2). `write_record` refuses an array of signals that is not 1-D or 2-D (`signals must be a 1-D or 2-D array`), baselines, ADC zeros and ADC resolutions that are not whole numbers within the range of a 32-bit integer (`baselines must be whole numbers within the range of a 32-bit integer`), and a record name, signal names, units or comments that cannot be written as header text (in Python, a character that is not Latin-1). `write_annotations` refuses an array of times (or of codes, subtypes, channels or numbers) that is not 1-D (`samples must be a 1-D array`), annotation codes outside 0 to 58 (`annotation code 61 is reserved (59 to 63 are the control codes of the annotation file format)`, `annotation code 64 is out of range (0 to 58)`: the codes 59 to 63 are the control codes `SKIP`, `NUM`, `SUB`, `CHN` and `AUX` of the MIT format, and an annotation written with one of them corrupts the file), and the times -2^{63} and $2^{63}-1$ (`annotation time T cannot be written (annotation times must be integers from -9223372036854775807 to 9223372036854775806)`), which `putann` refuses (Section 4.14); every other time can be written. Both writers then write their files to temporary files in the directories of the destinations (`.NAME.XXXXXXXX.tmp`), which are renamed over the destinations once all of them are complete, and removed after a failure, so that a failure never leaves a truncated header, signal file or annotation file in place of an existing one. A destination that is not a regular file, such as a device, is written directly; for a symbolic link, the file it leads to is replaced, and the link is kept. The files written are unchanged, byte for byte, and keep the permissions of the files they replace. (In legacy mode, an annotation file that must be sorted is renamed over its destination before the `sortann` program sorts it there.) The library functions (`putann`, `putvec`, `newheader`, ...) and the tools keep the behavior of C.
- **Read errors in the high-level readers.** `read_record` raises an error (`can't read signal data of record REC (getvec status S)`) when a signal file ends before the length its header gives, or cannot be read, instead of returning the samples read so far. A read failure is tracked separately from the checksum: when a signal file ends in the frame at which the checksum of a signal is verified, for which `getvec` returns -4 (checksum error) instead of -3, the high-level readers still raise the read error (`getvec status -3`) and never return an incomplete frame, and `check_record` reports the file as shorter than its header says, without a checksum error (version 1.2.2). `read_record` allocates at most 2^{26} samples in advance, whatever length the header gives, and grows its result as the samples are read, so that a header that gives far more samples than its files hold ends in the read error, not in an attempt to allocate memory for them. `read_annotations` checks `sampfrom` and `sampto` as `read_record` does (whole numbers, `sampfrom` not negative), before the file is opened, and closes the file whatever happens. A checksum error alone (every sample read, but their sum differs from the header's checksum) is a warning instead (Section 4.13).

- **Header metadata** (version 1.2.0): the functions `read_metadata`, `get_metadata`, `set_metadata`, `remove_metadata` and `update_metadata` read and change the `Key: value` info strings of a header without changing anything else in it, `read_header` returns them, and `write_record` writes them (Chapter 5). The info strings are those that `getinfo` reads, after the record line found as the header parser finds it, in every mode (Section 4.6). The library functions `putinfo` and `setinfo`, and the tools, are unchanged, and the C library reads the changed headers as before.
- **The high-level functions leave the annotation time resolution and the getvec mode as they were** (version 1.2.0). Opening an EDF record sets the time resolution of output annotators (`setafreq`) to the record's sampling frequency when none is set, and adds `WFDB_HIGHRES` to the getvec mode (`setgvmode`), as in C, and `wfdbquit` resets neither. `read_record`, `read_header`, `read_annotations`, `write_record`, `write_annotations` and `check_record` restore both previous values (`getafreq`, `getgvmode`) when they return, also after an error, so that `read_header("100s.edf")` changes nothing for a later `write_annotations` or `getvec`; the `fs` argument of `write_annotations` applies to its own file only. The library functions keep the behavior of C (Section 4.14).
- **Library functions** for these settings: `wfdbx_set_legacy`, `wfdbx_get_legacy` and `wfdbx_legacy_args` (legacy mode), `wfdbx_time_needs_freq`, `wfdbx_write_errors` and `wfdbx_note_write_error`, the functions of the input and output directories and of network access, and `wfdbx_input_path` and `wfdbx_output_path` (version 1.2.0), which give the path under which a program opens a file that it reads or creates itself (Section 3.3.2).

The high-level functions behave identically in all seven implementations, in legacy mode too, since they are not part of the original package.

4.9 Corrections of version 1.2.0

Version 1.2.0 changes the behavior of WFDB-x in three points, which were errors of the earlier versions:

- **The files that a tool reads or creates itself follow the input and output directories** (Section 3.3.2). The `-i` file of `wrsamp`, `edf2mit`, `wav2mit`, `a2m` and `ad2m`, the input files of `ahaecg2mit` and the `-S` script of `xform` are read from the input directory; the WAV file of `mit2wav`, the EDF file of `mit2edf`, the master header of `snip -m` and the `.des`, `.ecg`, `.key` and `.txt` files of `parsescp` are created in the output directory; `wfdbdesc` measures a signal file of unknown length in the header's directory. They were opened relative to the current directory: with `WFDBX_OUTPUT_DIR` set, `mit2wav` left a WAV header without samples in the current directory and the samples without a header in the output directory. In legacy mode, input files are opened as named, and the output directory is a prefix of every output name. New library functions: `wfdbx_input_path` and `wfdbx_output_path` (Section 13.11).
- **`osigopen(record, NULL, 0)`** returns 0 in every language (Section 4.7); the Python implementation raised an `IndexError`.
- **The high-level functions leave the annotation time resolution and the getvec mode as they were** (Section 4.8). After `read_header("100s.edf")`, or any high-level function that opens an EDF record, a later `write_annotations` in the same process no longer records the EDF record's sampling frequency as the time resolution of its file, and the getvec mode no longer stays in high resolution. The library functions keep the behavior of C.

4.10 Corrections of version 1.2.2

Version 1.2.2 corrects the following points, in all seven implementations. The library functions and the tools keep the behavior of C, except for `getinfo` and the URL schemes.

- **Truncated records.** A signal file that ends before the length that its header gives, or that cannot be read, is a read error of `read_record` (`can't read signal data of record REC (getvec status -3)`) even when the checksum of a signal is verified in the incomplete frame. `getvec` returns the checksum status (-4) for that frame, which replaced the end of file (-3), so `read_record` returned the incomplete frame as data: a format-16 file holding the samples 1 and 2, whose header gives 3 samples, was read as 1, 2, 257, with a checksum warning. The high-level readers now track a read failure separately from the checksum, and never return an incomplete frame; `check_record` reports such a file as shorter than its header says. A checksum error alone is still a warning (Section 4.8).
- **`read_record` allocates no more than it can read.** Its result was allocated from the length that the header gives (a header giving 10^{12} samples for a file of 4 bytes attempted to allocate terabytes); at most 2^{26} samples are now allocated in advance.
- **The high-level writers check everything before touching any file, and leave the existing files intact after a failure** (Section 4.8). `write_record` refuses, before any file is opened, arrays that are not 1-D or 2-D, baselines, ADC zeros and ADC resolutions that are not whole numbers within the range of a 32-bit integer, and text that cannot be written in a header (in Python, `units="µV"` left the existing header empty). `write_annotations` refuses arrays of times that are not 1-D (a 2-D array truncated the existing file), the annotation codes 59 to 63 and any code outside 0 to 58, and the times -2^{63} and $2^{63}-1$, which were dropped while success was reported. An error that `putann` reports is an error of `write_annotations`. Both writers write through temporary files that replace the destinations only when all of them are complete, keeping their permissions and symbolic links.
- **`read_annotations` checks `sampfrom` and `sampto`** as `read_record` does (a fraction, NaN or an infinite value is an error, before the file is opened: `sampfrom must be a whole number`, `sampfrom must not be negative`), and closes the annotation file whatever happens.
- **The record line of a header** is found by `getinfo` as the header parser finds it, except in legacy mode, and by the metadata functions in every mode (Section 4.6): for a header whose first line is empty or blank, the comments before the record line were returned as info strings (in `comments` and in the notes of `wfdbdesc`).
- **A remote file that could not be read is requested again** (Section 3.4). A download that failed (a server error, a timeout, a file not yet published) was remembered until the process ended, so the file could not be read even after the server recovered; only the files that have been read are now kept.
- **Only `http://` and `https://` URLs** (Section 3.4). A name that begins with another URL scheme (`ftp://`, `sftp://`, `file://`, ...) was opened as a local path (`ftp://h/db/100` as the directory `ftp:`, then `h/db/100`); it is now refused in normal mode, with or without network access, with the message `WFDB-x: can't open NAME: only http:// and https:// URLs can be read (mount the server as a local folder, or download the files)`, and so are an input directory, a file named in a header, the files that a tool reads or creates itself, and an output directory with such a scheme. Legacy mode is unchanged.
- **`time2sec -r` needs the sampling frequency of the record**: a record whose header gives none, or cannot be read, is an error (exit status 2: `time2sec: sampling frequency of record RECORD is not specified`). It printed `-9223372036854775808` with the exit status 0 (a division by the unspecified frequency, 0); the C tool converted the time at 250 Hz, as legacy mode still does (Section 4.1).

- **`write_annotations` requires a label for each annotation**, in every language: with one annotation or more and neither symbols nor codes, it is an argument error (`symbols or codes must be given`; in R, `symbol or code must be given`), before any file is touched. C++, Rust and Julia wrote such annotations with the code 0, which is not an annotation code; Python, R and Kotlin refused even an empty list, which now writes an empty file, as in the other languages.
- **`write_record` removes a trailing `.hea`** from the record name, as the readers do: `write_record("x.hea", ...)` writes `x.hea` and `x.dat`. Python failed (`can't create signal file for record x.hea`), and the other languages named the signal file `x.hea.dat`.
- **Metadata changes** keep the permissions of the header, whatever its file name, and their temporary file is never more open than the header (Section 5.3); corrections in R, MATLAB/Octave and Kotlin.
- **Licenses.** The file `NOTICE` (the licenses of WFDB-x, and the BSD-3-Clause notice of libFLAC, whose stream decoder the FLAC decoders translate) is included in the package of every language, with the license files (Section 1.9). The R package declares `License: file LICENSE`, and in MATLAB/Octave the dispatcher of the `wfdbx` command (`+wfdbx/+internal/cli_main.m` and `app_run.m`), which only the tools use, is under the GPL, as the launchers of the other languages are.
- **C++ package.** The installation provides `wfdbxConfigVersion.cmake`, so that `find_package(wfdbx 1.2.2 CONFIG REQUIRED)` checks the version, and the `wfdbx` target requires C++17 of the programs that link to it.

4.11 Corrections of version 1.2.3

Version 1.2.3 corrects the following points, reported by users, in all seven implementations. The library functions and the tools are unchanged.

- **`write_record` checks that every sample fits the storage format**, before any file is written. A sample outside the range of the format (Table 15.1), in format 8 a sample more than 127 away from the one before it or an invalid sample (format 8 stores differences, and has no invalid value), and more than 8 signals in a FLAC format (a FLAC file holds at most 8) are argument errors (`signal 0 has a sample (40000) outside the range of format 16 (-32768 to 32767)`). Such samples were silently wrapped around (40000 read back as -25536) or clipped by `putvec`, which keeps that behavior of C.
- **A warning for samples that will be read back as invalid**: the samples equal to the smallest value of the format (the value of an invalid sample, such as -32768 in format 16) or to `WFDB_INVALID_SAMPLE`, except for the NaN samples of physical units, which mark missing samples deliberately. The new option `invalid_warning` turns it off.
- **Tools run in the same process write to a captured standard output**: in Python, a tool whose `sys.stdout` was a text stream without a binary buffer (`io.StringIO`) failed with a `TypeError`; in R, the output of `wfdbx_tool` went past `sink()` and `capture.output()`.

4.12 Legacy mode

Legacy mode makes WFDB-x behave exactly as the original package, for reproducing old results. The default is the behavior described in the rest of this manual. It is turned on in three ways:

- **Tools**: `--legacy` as the *first* argument of any tool (`rdsamp --legacy -r 100s -t 10`; in Python `wfdbx rdsamp --legacy -r 100s -t 10`). It is removed before the options are parsed, and the tool then behaves exactly as the original one: same options (the options added by WFDB-x,

such as `-F` in `rdann` or `-R` in `wrsamp`, are unrecognized), outputs, messages, help texts and exit statuses. `--legacy` anywhere else is not recognized.

- **Library**: `wfdbx_set_legacy(1)` turns legacy mode on, and `wfdbx_set_legacy(0)` off; each returns the previous setting, and `wfdbx_get_legacy()` returns the current one. The setting applies to the whole library. `wfdbx_legacy_args` is the function that the tools call to recognize and remove `--legacy`.
- **High-level API**: the `legacy` option of `read_record`, `read_header`, `read_annotations`, `write_record`, `write_annotations` and the related functions turns the library's legacy mode on for the duration of the call, and restores the previous setting afterwards. The functions also follow the original rules when the library is in legacy mode.

No environment variable turns legacy mode on.

In legacy mode, every library difference of this chapter behaves as in the original: a header without a frequency means 250 Hz; a missing ADC resolution is inferred from the format or taken as 12 bits; the time conversions use 1 sample per second when no frequency is known; `setifreq` reports `no open input record`; the unit conversions assume a gain of 200 (and a baseline of 0 for invalid signal numbers); files are found through the WFDB search path, without network access (Section 3.3); out-of-order annotation files are sorted by running `sortann`, which loses annotations as the original does; an output file may be one of the input files; `getinfo` takes the first line that does not begin with `#` for the record line; the whole record name is checked when a header is written, and headers name records and files as given. The input directory and `WFDBX_NETWORK` are not used, no remote file is read (every name containing `://` is refused), and the tools open the files that they read themselves as named; the output directory is used, also for the files that the tools create themselves. The tools exit with the original status after a write error, and print no new messages. In `write_record`, legacy mode gives a gain of 200 and makes the units optional; its default ADC resolution is the storage width of the format (8 bits for formats 80 and 508, 10 for 310 and 311, 12 for 212, 24 for 24 and 524, 32 for 32) and 16 bits for every other format, which differs from `wrsamp --legacy` only for format 8 (12 bits there, as in the original `wrsamp`).

Legacy mode is tested against the unmodified C package by the legacy suite of the conformance tests (886 cases, each run with `--legacy`), and the library's legacy values by the unit tests.

> **Note** Legacy mode is kept for compatibility only. The values it assumes are exactly the silent defaults that WFDB-x removed, so results produced with it carry the same risks: a time, interval, resampled signal or physical value computed from an assumed frequency, gain, resolution or unit is wrong if the assumption is, and nothing in the output says so.

4.13 Errors and warnings

The library functions report errors as the C library does, in every language: a function returns an error status (a negative number, a null pointer, ..., as its C counterpart does), and the library prints a message on the standard error output, such as `init: can't open header for record 100`. `wfdbquiet()` stops the messages from being printed, and `wfdbverbose()` prints them again; `wfdberror()` returns the last message (before any error, `WFDB library version 10.7.0 (Jun 10 2022)`.), and `wfdb_error(format, ...)` reports a message as the library does. The messages are those of the C library, byte for byte, so that the tools print what the C tools print.

The high-level functions report errors in the usual way of each language, with a message that says which operation failed, such as `can't read record 100s`; the library's own message, if any, is printed too, unless `wfdbquiet()` was called, and returned by `wfdberror()`.

- Python raises `OSError` (`IOError`) when a file cannot be read or written, `ValueError` for an invalid argument, `TypeError` for an argument of the wrong type, and `FileNotFoundError` when an output directory (`dir`) does not exist.
- C++ throws `wfdbx::Error`, or, in the metadata functions (and in `write_record`, for invalid metadata), one of its subclasses `ArgumentError` and `FileError` (in the namespace `wfdbx`).
- Rust returns a `Result` whose error type is `wfdbx::record::Error`.
- Julia throws a `WfdbError`, or, for some invalid arguments of the metadata functions and of `check_record`, an `ArgumentError`.
- R signals an R error (in the metadata functions, of class `wfdbx_argument_error` or `wfdbx_file_error`).
- MATLAB/Octave throws an error, with an identifier such as `wfdbx:read_record`.
- Kotlin throws a `WfdbException`, or, for an invalid argument of the metadata functions, the writers and `readAnnotations`, its subclass `WfdbArgumentException`.

The chapters of Part II list the errors of each function.

The checksum warning. When a high-level reader has read every sample of a signal but their sum differs from the checksum of the header, it returns every frame, the last one included, with the warning

```
checksum error in record REC: the signal data do not match the checksums in its header
```

in each language's usual way: Python `warnings` (a `RuntimeWarning`), R `warning()`, Julia `@warn`, MATLAB/Octave `warning()` (identifier `wfdbx:checksum`), and the standard error output in C++, Rust and Kotlin (also silenced by `wfdbquiet()`; in Kotlin, an `onWarning` callback can receive it instead). The `checksum_warning` option (`checksumWarning` in Kotlin) turns it off, and the record's `checksum_ok` field is false in either case, so that a program can test for it. The library function `getvec` reports such an error as in C: `getvec: checksum error in signal N`, and the status -4 for the last frame; the tools, such as `rdsamp`, therefore stop before printing that frame, as the C tools do. A signal file that ends early is a read error of the high-level readers, not a checksum warning, even when the checksum is verified in its last, incomplete frame (Section 4.8).

4.14 Known C behaviors that WFDB-x reproduces

WFDB-x reproduces the C package exactly, including behaviors that may look surprising; where the documentation of the C package and its code disagree, WFDB-x follows the code. The file `docs/KNOWN_C_BEHAVIORS.md` lists them, and says for each whether the official documentation describes it. Among them:

- **Checksums.** `getvec` returns -4 for the frame at which a checksum mismatch is found, so that programs that loop while `getvec(...) >= 0`, such as `rdsamp`, stop before printing the last frame.
- **`getinfo` and the header parser disagree on where the record line is.** For a header whose first line is empty or blank, the C function `getinfo` takes that line for the record line, and returns the comments between it and the real record line as info strings. This is reproduced in legacy mode only: WFDB-x's `getinfo` finds the record line as the header parser does (Section 4.6).
- **`putann` writes reserved annotation codes, and drops the extreme times.** `putann` does not check the annotation code: an annotation written with one of the codes 59 to 63, the control codes of the MIT format, is read back as a control word, which corrupts the rest of the file, and a code above 63 loses its high bits. An annotation at the time `WFDB_TIME_MIN` or

WFDB_TIME_MAX (-2^{63} or $2^{63}-1$) is refused with `putann: time overflow in annotation file N` and the status -1, and nothing is written for it. The library function putann of WFDB-x does the same; the high-level write_annotations refuses such codes and times before writing anything (Section 4.8).

- **Modification labels.** annopen consumes the NOTE annotations at time 0 (subtype 0) at the beginning of an annotation file, so getann does not return them; a file whose table redefines mnemonics makes every annotation file written later in the same process repeat the table.
- **The time resolution note.** A program that calls setafreq with a positive value (as wrann does) writes a `## time resolution:` note at the beginning of its output files.
- **Duplicates at time 0.** An annotation at time 0 that follows another at time 0 counts as in order, so that annotations written twice at a time-0 location are not sorted, and both are kept.
- **State that survives `wfdbquit`:** the counter frequency and base counter value, the output annotation time resolution, the buffer sizes, and the mnemonic and description tables are not reset.
- **Opening an EDF record** sets the time resolution of output annotators to the record's sampling frequency, when setafreq has not set one, and adds WFDB_HIGHRES to the getvec mode; wfdbquit resets neither, so that an annotation file written later by the same process begins with a time-resolution note (`## time resolution: 360` after 100s.edf), and later records are read in high resolution. The library functions do the same; the high-level functions restore both settings (Section 4.8).
- **Tools.** sortann keeps only the last of several annotations with the same time, num and chan; wrann turns an unrecognized mnemonic into a NOTE whose aux string starts with the mnemonic, and keeps the subtype, chan and num of the previous line when a line omits them; rr2ann repeats the previous interval for an empty line; revise rejects an old-style header of exactly 32 signals; edf2mit adds stray bytes to a 16-character signal label; ahaecg2mit writes the annotations of .ecg input one sample early; mit2edf uses the current date and time for a record that has no start time.
- **Documentation and code disagree** on a few points, and WFDB-x follows the code: wfdbquit does not reset the counter frequency; the variable that turns sorting off is WFDBANNSORT, not WFDBNOSORT; wrsamp's -c has no effect and -l is obsolete; its default field separators are space, tab, CR, LF and comma; edf2mit and wav2mit name their default output records after the input file name.
- **Build-specific values.** wfdbversion, wfdbldflags, wfdbcflags and wfdbdefwfdb return the values of a release build of 10.7.0 installed in /usr/local: `"10.7.0"`, `""`, `""`, and the default WFDB path of the C package, `". /usr/local/database http://physionet.org/physiobank/database"`; wfdbdefwfdbcal returns `"wfdbcal"`.

Behaviors that depend on the platform. The reference results were produced on x86-64 Linux with glibc, where char is signed, long is 64 bits, and WFDB_Time is 64 bits. WFDB-x reproduces that platform's behavior where C's would differ elsewhere; for example, floating-point operations are never contracted into fused multiply-adds (which some compilers do by default, and which can change results such as the baselines that edf2mit computes). One case remains: when an output file is also an input file in legacy mode (Section 4.4), what is read depends on how far the runtime reads ahead, as in C; the R implementation on macOS reads further ahead than glibc, and fails one legacy conformance case for that reason.

CHAPTER 5

Header metadata

Many datasets store facts about a recording in the info strings of its header (Section 2.2.3), as `Key: value` lines, for example:

```
# Age: 63
# Sex: F
# Dx: 164889003
```

These entries are the *metadata* of the record. Five functions of the high-level API, added in version 1.2.0, read and change them without changing anything else in the header: `read_metadata`, `get_metadata`, `set_metadata`, `remove_metadata` and `update_metadata`. `read_header` also returns the entries, and `write_record` writes them. The C package has no counterpart. The library functions `putinfo` and `setinfo`, and the tools, are unchanged, and so is `getinfo`, except for the rule by which it finds the record line (Section 2.2.3), which the metadata functions share; the C library reads the headers that these functions change exactly as before (`tests/metadata/check_c.py` checks this with the original programs). The seven implementations follow the rules of this chapter identically: they pass the same cases (`tests/metadata/`) and write byte-identical headers.

5.1 Entries

A header is split into lines after each line feed (LF). Each line has a line ending: CR LF, LF, or none (the last line of a file that does not end with a line feed). The *content* of a line is the line without its ending.

Info strings. The record line is found as the header parser of the library (`readheader`) finds it: it is the first line whose content, up to a NUL character if there is one, has a token (separated by spaces, tabs, carriage returns and line feeds) that does not begin with `#`; in other words, the first line that holds something other than white space and does not begin, after white space, with `#`. The lines before it (comments, and empty or blank lines) are not info strings, and are never read or changed. Every line after the record line whose content begins with `#` is an info string, as `getinfo` reads a header (so a comment between two signal lines is an info string too). A header with no record line has no info strings, and changing its metadata is an argument error. This rule is the same in every mode, legacy mode included (since version 1.2.2; before, the record line was the first line whose content does not begin with `#`, the rule of the C function `getinfo`, which takes an empty first line for the record line).

Entries and free comments. An info string is an *entry* if the text after its `#` contains a colon. The *key* is the text before the first colon, and the *value* is the text after it; spaces, tabs, vertical tabs, form feeds and carriage returns are removed at both ends of each. If the key is then empty, or contains `#`, the info string is a *free comment*. Free comments, and info strings without a colon, are never returned and never changed. So `#Age:45`, `# Age: 45` and `#  Age : 45` all give the key `Age` and the value `45`, and Table 5.1 shows a few other cases.

Table 5.1 How info strings are read.

Info string	Entry (key, value)
`#Age: 63`	(`Age`, `63`)
`# Sex:F`	(`Sex`, `F`)
`#  Dx : 164889003, 59118001`	(`Dx`, `164889003, 59118001`)
`#Notes:`	(`Notes`, the empty string)
`#Time: 12:30:00`	(`Time`, `12:30:00`): the key ends at the first colon
`#<tab>Tab key<tab>:<tab>value with tabs`	(`Tab key`, `value with tabs`)
`# A free comment without a colon`	none: a free comment
`#: empty key`	none: the key is empty
`## Hash: not an entry`	none: the key (`# Hash`) contains `#`
`#Produced by xform from record 100, beginning at 0:0`	(`Produced by xform from record 100, beginning at 0`, `0`)

The last line of Table 5.1 is an info string of the sample record `100s`: free text that happens to contain a colon is read as an entry. Such an entry is harmless: it is changed only if its key is named.

Keys are compared without regard to letter case, for the letters A–Z and a–z only (no other character is changed: É and é differ). So `get_metadata(record, "age")` finds the entry `Age`. Entries are returned in file order.

Duplicate keys. A header should not hold two entries whose keys are equal, ignoring letter case. If one does (a file written by other software), the readers (`read_metadata`, `get_metadata`, `read_header`) return the first of them, and give one warning for each such key, naming the line numbers of its entries (from 1), such as `header of record dups has more than one entry for key age (lines 4, 6); the first is used`; `set_metadata` and `update_metadata` refuse to change that key (an argument error), and `remove_metadata` removes all of them, which repairs the header.

5.2 Arguments

Before a key or a value given to a function is checked, the spaces, tabs, vertical tabs, form feeds and carriage returns at both ends are removed.

- A **key** must then not be empty, and must not contain `:`, `#`, a carriage return, a line feed or a NUL character. It may contain spaces, as in `Date of birth`.
- A **value** is text, which may be a whole sentence. It must not contain a carriage return, a line feed or a NUL character; it may be empty.
- Text is one byte per character (as in the rest of WFDB-x: in Python, Latin-1 strings, or bytes as given).
- In `update_metadata`, no two keys may be equal, ignoring letter case, and a value of none (`None`, `nothing`, `NULL`, `null`, or a missing value, depending on the language) means “remove”.
- The record name `-` (the standard input) is an argument error, and so is `get_metadata` with an invalid key.

Every argument is checked before the header is read, and every change before anything is written: after an error, the header is unchanged.

5.3 Writing

The written form. An entry is written as `# Key: value`: `#`, a space, the key as the caller spelled it, `:`, and, if the value is not empty, a space and the value. An entry with an empty value is therefore written `# Key:`, with no trailing space.

The changes.

- `set_metadata`: if the header has one entry for the key, its line is replaced, at the same position, keeping its line ending, even if the value is the same (the line is then rewritten in the written form, with the caller's spelling of the key). Otherwise a new line is inserted.
- `remove_metadata`: every line of an entry for the key is deleted, with its line ending. The function returns the number of lines deleted (0 if there is none).
- `update_metadata`: the changes are applied in the order given, each as `set_metadata` or `remove_metadata` would apply it, and written at once.

Where new lines go. A new line is inserted after the last info string of the header, or, if the header has none, after its last line whose content is not blank (not made only of spaces, tabs, vertical tabs, form feeds and carriage returns). Its line ending is that of the record line, or, if the record line has none, that of the first line that has one, or CR LF (the line ending of the headers written by the WFDB library). If the line before it has no line ending, that line gets the same ending.

What is kept. Every other byte of the header is kept: the comments and the empty or blank lines before the record line, the record line, the signal lines and their checksums, the free comments, and the line endings. If nothing changes (and `dir` is not given), nothing is written.

Atomic replacement. The new header is written to a temporary file in the directory of the destination, which is never more open than the destination (in Python, it is created readable and writable by its owner only). The temporary file then gets the permissions of the destination (or, for a new file in `dir`, those of a new file) and replaces it (by renaming it). An error never leaves a partly written header, and no other file is left in the directory.

5.4 Which file

The header is opened as the library opens the header of a record, after removing a trailing `.hea` from the name: under the input directory, or as given for an absolute path, a ~ path or a URL (a URL only if network access is on); see Section 3.2. The functions take the options of the high-level readers for this, an input directory (`input_dir`) and a network permission (`network`), for the duration of the call. A header that cannot be opened is a file error.

A change is written to the file that was read, unless the call gives an output directory `dir`: the changed header is then written to the file of the same name in that directory, which must exist (else a file error), and the header that was read is unchanged. The signal files are not copied. A header read from a URL cannot be changed without `dir` (an argument error).

Multi-segment records. Only the named header is read or changed: for a multi-segment record, that is the master header, unless the name of a segment is given. (The info strings of a master header are read by `getinfo` like those of any other header.)

Info files. Only the header file (`.hea`) is read, never the info file (`.info`) whose comment lines `getinfo` also returns.

5.5 read_header and write_record

`read_header` returns the entries of a header, as `read_metadata` returns them (from the `.hea` file only), as the `metadata` of its result, beside its `comments`, which are all the info strings.

`write_record` writes the entries given in its `metadata` argument (a mapping from keys to values, or a list of key and value pairs, depending on the language) after its `comments`, in the written form `# Key: value`, through `putinfo`. Its keys must be unique, ignoring letter case, and none may be the key of an entry in `comments`; a value of none is an argument error. This is checked before any file is created.

5.6 An example

The header `tests/metadata/headers/basic.hea` (with CR LF line endings) is:

```
# Converted on: 2026-11-01 (a comment before the record line)
basic 2 360 21600
100s.dat 212 200 11 1024 995 21537 0 MLII
100s.dat 212 200 11 1024 1011 -3962 0 V5
#Age: 63
# Sex:F
#  Dx : 164889003, 59118001
# A free comment without a colon
#Notes:
# Recorded by: Holter monitor, model X
```

`read_metadata("basic")` returns the five entries, in file order: (`Age`, `63`), (`Sex`, `F`), (`Dx`, `164889003, 59118001`), (`Notes`, the empty string) and (`Recorded by`, `Holter monitor, model X`). The first line is before the record line, and is not an info string; the free comment is not an entry. In Python, the changes

```
wfdbx.update_metadata("basic", [("age", "64"), ("Sex", None),
                                ("Weight", "72"), ("Height", "170")])
```

replace the entry `Age` (at its position, with the caller's spelling `age`), remove the entry `Sex`, and add two new entries after the last info string, with the line ending of the record line (CR LF). The other lines are unchanged, byte for byte, including the unusual spacing of the `Dx` line. The header becomes `tests/metadata/expected/update_basic.hea`:

```
# Converted on: 2026-11-01 (a comment before the record line)
basic 2 360 21600
100s.dat 212 200 11 1024 995 21537 0 MLII
100s.dat 212 200 11 1024 1011 -3962 0 V5
# age: 64
#  Dx : 164889003, 59118001
# A free comment without a colon
#Notes:
# Recorded by: Holter monitor, model X
# Weight: 72
# Height: 170
```

A header whose first line is empty, such as `tests/metadata/headers/leading.hea`:

```
(an empty line)
# Age: preamble
leading 2 360 21600
100s.dat 212 200 11 1024 995 21537 0 MLII
```

```
100s.dat 212 200 11 1024 1011 -3962 0 V5
# Age: actual
```

has one entry, (`Age`, `actual`): its record line is the third line, as for the header parser, and `# Age: preamble`, before it, is not an info string. `set_metadata("leading", "Sex", "F")` adds `# Sex: F` after `# Age: actual`, and leaves the first two lines as they are. In Python:

```
import wfdbx
with open("leading.hea", "w") as f:
    f.write("\n# Age: preamble\nleading 2 360 21600\n"
            "100s.dat 212 200 11 1024 995 21537 0 MLII\n"
            "100s.dat 212 200 11 1024 1011 -3962 0 V5\n# Age: actual\n")
print(wfdbx.read_metadata("leading"))           # [('Age', 'actual')]
print(wfdbx.read_header("leading").comments)    # [' Age: actual']
```

The following Python example writes a record with metadata, from the first second of record `100s`, and changes its metadata:

```
import wfdbx
rec = wfdbx.read_record("100s", sampto=360)
wfdbx.write_record("demo", rec.signals, fs=rec.fs, fmt=212, gains=rec.gains,
                   baselines=rec.baselines, adczeros=1024, adcres=11,
                   units="mV", names=rec.names,
                   metadata={"Age": "69", "Sex": "M"})
print(wfdbx.read_metadata("demo"))   # [('Age', '69'), ('Sex', 'M')]
wfdbx.set_metadata("demo", "age", "70")
wfdbx.update_metadata("demo", {"Sex": None, "Dx": "Normal sinus rhythm"})
print(wfdbx.get_metadata("demo", "AGE"))         # 70
print(wfdbx.read_metadata("demo"))   # [('age', '70'), ('Dx', 'Normal sinus rhythm')]
print(wfdbx.remove_metadata("demo", "dx"))       # 1
```

After the call of `update_metadata`, the header `demo.hea` is:

```
demo 2 360 360
demo.dat 212 200/mV 11 1024 995 20844 0 MLII
demo.dat 212 200/mV 11 1024 1011 28153 0 V5
# age: 70
# Dx: Normal sinus rhythm
```

5.7 The functions in each language

Table 5.2 lists the five functions, with their names in each language. They take the record name and, as options, the input directory and the network permission for reading the header, and, for the functions that change it, the output directory `dir`. The chapters of Part II give their exact signatures, results and errors (Chapters 6 to 12).

The functions are called with the prefix of each language: `wfdbx.` in Python and MATLAB/Octave, `wfdbx::` in C++ and Rust, `WFDBX.` in Julia, and none in R and Kotlin. In Rust, each function also has a form with options, `read_metadata_with`, `get_metadata_with`, and so on. The errors are those of the high-level functions of each language (Section 4.13). An argument error and a file error are told apart: in Python, `ValueError` (`TypeError` for an argument of the wrong type) and `OSError` (`FileNotFoundError` for a missing `dir`); in C++, `wfdbx::ArgumentError` and `wfdbx::FileError`; in Julia, `ArgumentError` and `WfdbError`; in R, the condition classes `wfdbx_argument_error` and `wfdbx_file_error`; in Kotlin, `WfdbArgumentException` and `WfdbException`.

Table 5.2 The header metadata functions.

What it does	Python, C++, Rust, Julia, R	MATLAB/Octave	Kotlin
Read all the entries, in file order, as (key, value) pairs	`read_metadata`	`wfdbx.read_metadata`	`readMetadata`
Return the value of one entry, or none	`get_metadata`	`wfdbx.get_metadata`	`getMetadata`
Change the value of an entry, or add it	`set_metadata`	`wfdbx.set_metadata`	`setMetadata`
Remove the entries for a key; return the number of lines removed	`remove_metadata`	`wfdbx.remove_metadata`	`removeMetadata`
Make several changes in one write	`update_metadata`	`wfdbx.update_metadata`	`updateMetadata`

Patient data Metadata often holds information about patients, such as ages, dates, diagnoses or names. WFDB-x stores it as plain text, exactly as given, and does not de-identify it, in the header or anywhere else. Removing identifying data from headers (their metadata, and the other info strings) before records are shared is the responsibility of the user.

PART II

The seven languages

CHAPTER 6

Python

The Python implementation of WFDB-x is the package `wfdbx`, published on PyPI as `wfdb-x`. It is the reference implementation: the six other languages are translations of it, and its results are those that they must reproduce. It is written in Python alone, on top of NumPy, and never calls the C library.

It offers WFDB-x in three ways:

- **The high-level API** (Section 6.3): functions that read and write whole records, headers, annotation files and header metadata as NumPy arrays and Python objects, in one call, with keyword arguments, and that raise exceptions on errors. They read signal files in large blocks, and are the fastest way to read a record. They are an addition of WFDB-x: the C package has no equivalent. They are the recommended interface for new code, analysis and notebooks.
- **The WFDB library functions** under their C names (`isigopen`, `getvec`, `getann`, `putann`, `...`), with the behavior of the C library (Section 6.4). They are documented once, for all the languages, in Chapter 13.
- **The command-line tools** (`rdsamp`, `rdann`, `wrsamp`, `xform` and 29 others), with the options, output and exit status of the C tools, run through the `wfdbx` command (Section 6.5). They are documented in Chapter 14.

The high-level functions return exactly the samples and annotations that the library functions (and the C library) give.

6.1 Installation and requirements

6.1.1 Requirements

WFDB-x for Python needs Python 3.9 or later and NumPy 1.17 or later, its only dependency. The package is pure Python: it has no compiled code and needs no C compiler, no WFDB library and no libFLAC (the FLAC formats 508, 516 and 524 are read and written by WFDB-x's own codec). It runs on Linux, macOS and Windows.

Two sets of optional dependencies are declared:

- `examples`: matplotlib and Jupyter, to run the example notebooks;
- `test`: pytest, nbformat, nbclient and ipykernel, to run the tests (Section 6.6).

6.1.2 Installing from PyPI

```
pip install wfdb-x                # the package and the wfdbx command
pip install "wfdb-x[examples]"    # also matplotlib and Jupyter, for the notebooks
```

The package is imported as `wfdbx` (`import wfdbx`); the installation also creates the command `wfdbx`, which runs the command-line tools. The installed version is `wfdbx.__version__`, or the output of `wfdbx --version`:

```
python -c "import wfdbx; print(wfdbx.__version__)"    # 1.2.3
wfdbx --version                                       # WFDB-x 1.2.3
```

The wheel and the source distribution include the licenses, `LICENSE` (the library, LGPL-2.1-or-later) and `LICENSE-GPL` (the command-line tools, GPL-3.0-or-later), and the file `NOTICE`, which says which parts are under which license and gives the BSD-3-Clause notice of libFLAC, from which the FLAC decoder (`wfdbx/_flac.py`) is translated (Section 1.9). The license expression of the package is `LGPL-2.1-or-later AND GPL-3.0-or-later AND BSD-3-Clause`.

6.1.3 Installing from the repository

From the root of a copy of the repository, the package is in the directory `python`:

```
pip install ./python                 # install the package
pip install -e "./python[test]"      # or: editable, with the test dependencies
```

Without installation, the package can also be used from its source directory by adding `python/src` to `PYTHONPATH`; the command-line tools are then run as `python -m wfdbx.cli` (Section 6.5).

6.1.4 The sample record and the example notebooks

The package includes the sample record `100s`, the sample record of the WFDB Software Package: the first minute of record 100 of the MIT-BIH Arrhythmia Database, two ECG signals (`MLII` and `V5`) sampled at 360 Hz, with the reference beat annotations (annotator `atr`). Its files (`100s.hea`, `100s.dat`, `100s.atr`) are in the directory returned by `wfdbx.sample_data_dir()`, so that the examples and the notebooks run without a network connection. Work that uses the MIT-BIH Arrhythmia Database cites it as described on PhysioNet.

The package also includes seven Jupyter notebooks (Table 6.1), in the directory returned by `wfdbx.examples_dir()`. Each runs without a network connection (`06_physionet_records` reads record 100 from PhysioNet when it can, and the sample record otherwise). The function `copy_examples`, or the command `wfdbx --examples DIR`, copies them to a directory where they can be opened and run:

```
wfdbx --examples my-examples      # without DIR: ./wfdbx-examples
cd my-examples
jupyter lab
```

Table 6.1 The example notebooks of the Python package

Notebook	Contents
`01_getting_started`	reading headers and signals; ADC and physical units
`02_annotations_and_heart_rate`	reading annotations; RR intervals and heart rate
`03_writing_records`	writing records and annotations
`04_command_line_tools`	running the tools
`05_c_library_api`	the WFDB library functions
`06_physionet_records`	reading a record from PhysioNet
`07_legacy_mode`	the original package's behavior, on request

6.2 Quick start

This example reads the header, ten seconds of the signals and the annotations of the sample record `100s`, from the package's data directory (`input_dir`), and computes the mean heart rate.

It then writes the first ten seconds as a new record, `first10`, in the current directory, with two metadata entries in its header.

```
import numpy as np
import wfdbx

sample = wfdbx.sample_data_dir()        # the directory of the sample record 100s

# The header only: no samples are read.
hdr = wfdbx.read_header("100s", input_dir=sample)
print(hdr.nsig, hdr.fs, hdr.nsamples)   # 2 360.0 21600
print([s.desc for s in hdr.siginfo])    # ['MLII', 'V5']

# Seconds 10 to 20 of both signals, in physical units (float64): one row per sample,
# one column per signal.
rec = wfdbx.read_record("100s", sampfrom="0:10", sampto="0:20", physical=True,
                        input_dir=sample)
print(rec.signals.shape, rec.signals.dtype)    # (3600, 2) float64
print(rec.signals[0])                          # [-0.39  -0.275]

# The beat annotations, and the mean heart rate.
ann = wfdbx.read_annotations("100s", "atr", input_dir=sample)
print(len(ann), ann.symbols[:3], ann.samples[:3])    # 75 ['+', 'N', 'N'] [ 18  77 370]
beats = ann.samples[[bool(wfdbx.isqrs(c)) for c in ann.codes]]
rr = np.diff(beats) / ann.fs                         # RR intervals, in seconds
print(len(beats), "beats,", round(60 / rr.mean(), 1), "beats per minute")   # 74 beats, 73.9 ...

# Write the first 10 seconds, in ADC units, as the record "first10" (first10.hea and
# first10.dat, in the current directory), with the signal specifications of 100s.
x = wfdbx.read_record("100s", sampto="0:10", input_dir=sample)
wfdbx.write_record("first10", x.signals, fs=x.fs, fmt=16, gains=x.gains,
                   baselines=x.baselines, adczeros=[s.adczero for s in x.siginfo],
                   adcres=[s.adcres for s in x.siginfo], units="mV", names=x.names,
                   metadata={"Source": "MIT-BIH record 100"})

# Header metadata: "# Key: value" lines of the header.
wfdbx.set_metadata("first10", "Age", "69")
print(wfdbx.read_metadata("first10"))   # [('Source', 'MIT-BIH record 100'), ('Age', '69')]
print(wfdbx.check_record("first10"))    # no problems found
```

The header `100s.hea` gives no units: the example writes `"mV"` explicitly, since WFDB-x never assumes units, a gain, an ADC resolution or a sampling frequency that a record does not give (Chapter 4).

6.3 The high-level API

The high-level API is made of the functions of this section, all available at the top level of the package (`wfdbx.read_record`, ...), and of the classes of their results (Section 6.3.7). They are defined in the modules `wfdbx.highlevel`, `wfdbx.metadata`, `wfdbx.check` and `wfdbx.resources`; `help()` gives the documentation of each.

6.3.1 Conventions

Record names and directories. A record is named as in the other parts of WFDB-x (Chapter 2 and Section 3.2), without the `.hea` extension (a trailing `".hea"` is removed, by the readers and the writers). A record name is one of:

- a full path, absolute (`"/data/mitdb/100"`) or in the home directory (`"~/Documents/100s"`), used as given;
- a relative path (`"100s"`, `"data/100s"`), read from the input directory if one is set, otherwise from the current directory, and written to the output directory if one is set, otherwise to the current directory (missing directories are created);
- a URL (`"https://physionet.org/files/mitdb/1.0.0/100"`), read only if network access is on; a URL cannot be written.

The input and output directories are set with `wfdbx_set_input_dir` and `wfdbx_set_output_dir` (Section 6.4.4) or with the environment variables `WFDBX_INPUT_DIR` and `WFDBX_OUTPUT_DIR`. The functions that read take the optional keyword arguments `input_dir` and `network`, and those that write take `dir`: they apply to that one call, and the previous settings are restored when it returns, also after an error. `input_dir` applies to relative names only, not to full paths or URLs. A writer given a name with a path creates the files in that directory, and the header names the record and its signal file without the path (`rec`, `rec.dat`), so that the record opens under any path that leads to it; in legacy mode, as in the original, the header gives the name with its path. With `dir`, the name must be a plain name (`"rec"`): the files are created in `dir`, and the record opens from that directory under its plain name. The signal files that a header names are read from the header's directory.

Times. The arguments `sampfrom` and `sampto` are sample numbers (whole numbers) or WFDB time strings, as `strtim` reads them: `"0:10"` (10 seconds), `"1:02:03.5"`, `"s3600"` (sample 3600), `"e"` (the end of the record), or a time of day in brackets, `"[14:00:00]"`. A time in seconds needs the sampling frequency; when the record does not give it, such a time raises `ValueError`.

Per-signal and per-annotation arguments. The arguments that describe each signal (`fmt`, `gains`, `units`, ...) or each annotation (`subtypes`, `chans`, ...) take one value for all, or a list (or tuple, or NumPy array) with exactly one value per signal or per annotation.

Signal parameters. No sampling frequency, gain, ADC resolution or units is ever assumed. Where one is needed and the record does not give it, it must be given as an argument; otherwise the function raises `ValueError`. A record whose header gives no sampling frequency has `fs = 0`.

Legacy mode. With `legacy=True`, a function behaves as in the original WFDB package, with its assumed signal parameters and its search for files (Chapter 4); the library's legacy mode is on for the duration of the call only. With `legacy=False` (the default), the current setting of `wfdbx_set_legacy` applies, which is off unless it was turned on.

State. The functions that read and write records and annotation files close anything opened with the WFDB library functions (they call `wfdbquit()`) before they start, and close their own files before they return, whatever the outcome. `read_record` sets the getvec mode for its own reading only. `read_record`, `read_header`, `read_annotations`, `write_record`, `write_annotations` and `check_record` leave the time resolution of output annotators (`getafreq()`) and the getvec mode (`getgvmode()`) as they were, also after reading an EDF record, whose opening sets both (as in C; the library functions keep that behavior). The metadata functions open no signals and leave the library's state as it is.

The examples of this section read the test records of WFDB-x (among them `100s`) from the current directory, and write records of their own there. To run them elsewhere, copy the sample record to the current directory first, for example with `shutil.copytree(wfdbx.sample_data_dir(), ".", dirs_exist_ok=True)`.

6.3.2 Reading records

read_record Read the signals of a record

```
read_record(record, channels=None, sampfrom=0, sampto=None, physical=False, highres=False, pad=
    ↪ False, *, input_dir=None, network=None, legacy=False, checksum_warning=True)
```

Reads the signals of a record, or part of it, into a NumPy array with one row per sample and one column per signal, together with the description of the record. The samples are those that `getvec` returns (Chapter 13), read in large blocks. Invalid samples are `wfdbx.WFDB_INVALID_SAMPLE` (-32768) in ADC units, and NaN in physical units. When the read reaches the end of the record, the samples are compared with the checksums of the header (Section 6.3.8). The result is allocated for at most 2^{26} samples in advance, whatever length the header gives, and grows as the samples are read.

record
: Record name (a string), without the `.hea` extension (a trailing `".hea"` is also accepted): a full path, absolute (`"/data/mitdb/100"`) or in the home directory (`"~/Documents/100s"`), used as given; a relative path (`"100s"`, `"data/100s"`), read from the input directory if one is set, otherwise from the current directory; or a URL (`"https://physionet.org/files/mitdb/1.0.0/100"`), read only if network access is on.

channels
: The signals to read, in the order in which they are returned: a list of signal numbers (`int`, from 0) or signal descriptions (`str`, as in `Record.names`, such as `"MLII"`), which may be mixed. Default `None`: all the signals, in the order of the header.

sampfrom
: The first sample to read: a sample number (a whole number ≥ 0) or a time string. Default `0`: the beginning of the record.

sampto
: The sample after the last one to read: a sample number (a whole number) or a time string. A value at or before `sampfrom` reads no samples. Default `None`: the end of the record.

physical
: `False` (the default) to return ADC units, as `int32`; `True` to return physical units, as `float64`, computed as $(adc - baseline)/gain$. Every signal read must then have a gain.

highres
: `False` (the default) to read a multi-frequency record at its frame frequency, with the samples of the faster signals averaged to one per frame; `True` to read every sample, at the highest sampling frequency of the record (the samples of the slower signals are repeated). `Record.fs` is then that frequency.

pad `True` to replace each invalid sample by the previous valid sample of the same signal. Default `False`.

input_dir
: The input directory for this call only (optional): a local directory, a ~ path or the URL of a directory, under which relative record names are read. It does not apply to full paths or URLs. Default `None`: the current setting (`wfdbx_set_input_dir`).

network
: `True` to allow reading URLs for this call only, `False` to forbid it. Default `None`: the current setting (`wfdbx_set_network`).

legacy
: `True` to behave as the original WFDB package (legacy mode, for this call only). Default `False`.

checksum_warning
: `True` (the default) to issue a `RuntimeWarning` when the samples do not match the checksums of the header; `False` for no warning. The samples are returned either way, and `checksum_ok` of the result is `False`.

Returns A `Record` (Table 6.2).

Errors `IOError` (that is, `OSError`) if the header or a signal file cannot be read, if the record has no signals, if `sampfrom` is beyond the end of the record, or if a signal file is shorter than its header says or cannot be read ("can't read signal data of record REC (getvec status -3)"), even if the checksums do not match either: no incomplete frame is ever returned; `ValueError` if `sampfrom` is negative, if `sampfrom` or `sampto` is not a whole number (a fraction, NaN or an infinite value), if a channel is not in the record, if a time string needs the sampling frequency and the header gives none, or if `physical=True` and a signal read has no gain. Warns with a `RuntimeWarning` on a checksum error (see `checksum_warning`).

```
import wfdbx

# Seconds 10 to 20 of lead MLII, in physical units.
rec = wfdbx.read_record("100s", channels=["MLII"], sampfrom="0:10", sampto="0:20",
                        physical=True)
print(rec.signals.shape, rec.signals.dtype)   # (3600, 1) float64
print(rec.names, rec.fs, rec.sampfrom)        # ['MLII'] 360.0 3600

# The first 3 samples of both signals, in ADC units, V5 first.
rec = wfdbx.read_record("100s", channels=[1, 0], sampto=3)
print(rec.signals.tolist())     # [[1011, 995], [1011, 995], [1011, 995]]
print(rec.names, rec.gains, rec.baselines, rec.units)
# ['V5', 'MLII'] [200.0, 200.0] [1024, 1024] [None, None]

# The whole record: the checksums are verified.
rec = wfdbx.read_record("100s")
print(rec.signals.shape, rec.checksum_ok)     # (21600, 2) True
```

A multi-frequency record (here the test record `mfreq`, whose signals have 1, 2 and 4 samples per frame at a frame frequency of 100 Hz) is read at its frame frequency, or at its highest sampling frequency:

```
import wfdbx

rec = wfdbx.read_record("mfreq")
print(rec.fs, rec.signals.shape)       # 100.0 (600, 3)
rec = wfdbx.read_record("mfreq", highres=True)
print(rec.fs, rec.signals.shape)       # 400.0 (2400, 3)
print([s.spf for s in rec.siginfo])    # [1, 2, 4]
```

read_header

Read the header of a record

```
read_header(record, *, input_dir=None, network=None, legacy=False)
```

Reads the header of a record, without opening its signal files. The result describes the record and each of its signals, and holds the info strings and the metadata entries of the header. For a multi-segment record, the master header is read, and `nsamples` is the length of the whole record.

`record`
: Record name (a string), as for `read_record`: a full path (absolute or ~), used as given; a relative path, read from the input directory if one is set, otherwise from the current directory; or a URL, read only if network access is on.

`input_dir`
: The input directory for this call only (optional; as for `read_record`), under which relative record names are read; it does not apply to full paths or URLs. Default `None`: the current setting.

`network`
: `True` to allow reading URLs for this call only, `False` to forbid it. Default `None`: the current setting.

`legacy`
: `True` to read the header as the original WFDB package does. Default `False`.

Returns A `Header` (Table 6.3).

Errors `IOError` if the header cannot be found or read.

```
import wfdbx

hdr = wfdbx.read_header("100s")
print(hdr.name, hdr.fs, hdr.nsig, hdr.nsamples)    # 100s 360.0 2 21600
for s in hdr.siginfo:
    print(s.desc, s.fname, s.fmt, s.gain, s.adcres, s.units)
# MLII 100s.dat 212 200.0 11 None
# V5 100s.dat 212 200.0 11 None
print(hdr.comments[0])          # " 69 M 1085 1629 x1": the first info string
print(hdr.base_time)            # None: the header gives no start time
```

`read_annotations` — Read an annotation file

```
read_annotations(record, annotator, sampfrom=0, sampto=None, fs=None, *, input_dir=None, network=
    ↪ None, legacy=False)
```

Reads the annotations of an annotation file, in file order, into NumPy arrays and lists. The sampling frequency, needed to convert times in seconds, is that of the record's header, or, for a record without a header or whose header gives none, the argument `fs`.

`record`
: Record name (a string), as for `read_record`: a full path (absolute or ~), used as given; a relative path, read from the input directory if one is set, otherwise from the current directory; or a URL, read only if network access is on. Its header, if there is one, gives the sampling frequency.

`annotator`
: Annotator name: the suffix of the annotation file. `"atr"` reads the file `RECORD.atr`.

`sampfrom`
: The first time to read: a sample number (a whole number ≥ 0) or a time string. Default `0`: the beginning of the file.

sampto
: The time at which to stop: annotations at or after it are not read. A sample number (a whole number) or a time string. Default `None`: the end of the file.

fs
: The sampling frequency, in Hz (a number > 0), to use when the record has no header or its header gives none; it is ignored otherwise. Default `None`: none.

input_dir
: The input directory for this call only (optional; as for `read_record`), under which relative record names are read; it does not apply to full paths or URLs. Default `None`: the current setting.

network
: `True` to allow reading URLs for this call only, `False` to forbid it. Default `None`: the current setting.

legacy
: `True` to behave as the original WFDB package (which assumes 250 Hz for a record without a frequency). Default `False`.

Returns An `Annotations` object (Table 6.4). Its `fs` is the frequency in which the times are counted: the one written in the annotation file, if there is one, or else the sampling frequency; it is 0 if none of them is known.

Errors `IOError` if the annotation file cannot be opened or read, or if it ends before its end-of-file marker ("can't read annotation file ANN of record REC (getann status -3)"); `ValueError` if `fs` is given and is not > 0, if `sampfrom` is negative ("sampfrom must not be negative"), if `sampfrom` or `sampto` is not a whole number (a fraction, NaN or an infinite value: "sampfrom must be a whole number"), or if a time string needs the sampling frequency and none is known. The arguments are checked before the annotation file is opened (a time string is converted with the sampling frequency of the record's header, which is read for it), and the annotation file is closed whatever happens.

```
import wfdbx

ann = wfdbx.read_annotations("100s", "atr", sampfrom="0:10", sampto="0:30")
print(len(ann), ann.fs)                  # 24 360.0
print(ann.samples[:3], ann.symbols[:3])  # [3862 4170 4466] ['N', 'N', 'N']

# The whole file: the first annotation is a rhythm change, with an aux string.
ann = wfdbx.read_annotations("100s", "atr")
print(ann.symbols[0], ann.codes[0] == wfdbx.RHYTHM, ann.aux[0])   # + True (N
print(sorted(set(ann.symbols)))          # ['+', 'A', 'N']
```

For a record without a header, `fs` gives the frequency in which times in seconds are converted (the library reports on the standard error that it cannot open the header):

```
import wfdbx

wfdbx.write_annotations("beats", "qrs", [100, 460, 820], symbols="N")   # no header
ann = wfdbx.read_annotations("beats", "qrs", sampto="0:02", fs=360)
print(ann.samples, ann.fs)               # [100 460] 360.0
```

6.3.3 Writing records

write_record Write a header and a signal file

```
write_record(record, signals, fs, fmt=16, gains=None, baselines=None, adczeros=0, adcres=None,
    ↪ units=None, names=None, base_time=None, comments=None, physical=False, samps_per_frame=
    ↪ None, *, metadata=None, dir=None, legacy=False, invalid_warning=True)
```

Writes signals as a new record: the header `RECORD.hea` and one signal file, `RECORD.dat`, which holds all the signals (so they share one format). Existing files of these names are replaced: both files are written to temporary files in the destination directory (`.NAME.XXXXXXXX.tmp`, readable and writable by the owner only), which replace the existing files, with their permissions, only when both are complete; after an error, the temporary files are removed and the existing files are left as they were. For a symbolic link, the file it leads to is replaced; a destination that is not a regular file (a device) is written directly. The sampling frequency, the gain and ADC resolution of each signal, and the units of each calibrated signal must be given: there are no defaults. The per-signal arguments (`fmt`, `gains`, `baselines`, `adczeros`, `adcres`, `units`, `names`) take one value for all the signals, or a list with one value per signal. All the arguments are checked before any file is created or changed.

record
: Record name (a non-empty string), without the `.hea` extension (a trailing `".hea"` is removed, as by the readers): a plain name (`"rec"`) or a relative path (`"data/rec"`), created in the output directory if one is set, otherwise in the current directory, missing directories included; or a full path, absolute or in the home directory (`"~/Documents/rec"`), used as given. With a path, the files are created in that directory, and the header names the record and its signal file without the path (`rec`, `rec.dat`), so that the record opens under any path that leads to it (in legacy mode, as in the original, the header gives the name with its path). A URL cannot be written.

signals
: The samples: an array (or anything `numpy.asarray` accepts) with one row per sample and one column per signal (2-D); a 1-D array is one signal. In ADC units (the default), the values are integers, or floats with whole values, within the range of a 32-bit integer; with `physical=True`, they are in physical units.

fs The sampling frequency, in Hz: a finite number > 0. Required.

fmt The storage format, such as `16`, `212`, `24` or `516` (FLAC); one value, or one per signal, all equal (Chapter 15). Every sample must fit the format: within its range (Table 15.1; -2048 to 2047 for format 212, for example), and, in format 8, within -128 to 127 of the sample before it. The smallest value of a format is the value of an invalid (missing) sample: a sample equal to it, or to `WFDB_INVALID_SAMPLE`, is read back as invalid (see `invalid_warning`). A FLAC file holds at most 8 signals. Default `16`.

gains
: The gain of each signal, in ADC units per physical unit: a finite number; `0` marks an uncalibrated signal, which is allowed only for samples given in ADC units. Negative gains are allowed. Required (default `None`: not given).

baselines
: The ADC value that corresponds to 0 physical units (a whole number within the range of a 32-bit integer). Default `None`: the ADC zero of the signal.

adczeros
The ADC value at the middle of the ADC range (a whole number within the range of a 32-bit integer). Default `0`.

adcres
The ADC resolution, in bits (a whole number ≥ 1). Required (default `None`: not given).

units
The physical units, such as `"mV"` (Latin-1 text, one byte per character, as all text in `wfdbx`). Required for each signal whose gain is not 0. Default `None`: not given.

names
The signal descriptions, such as `"MLII"` (Latin-1 text). Default `None`: no descriptions.

base_time
The start time, with an optional date: `"hh:mm:ss"` or `"hh:mm:ss dd/mm/yyyy"`, with optional decimals of the seconds. Default `None`: no start time.

comments
A list of info strings (Latin-1 text), written to the header as comment lines (`"#"` followed by the string). Default `None`: none.

physical
`True` if `signals` are in physical units. They are converted to ADC units as $\mathrm{round}(x \cdot gain) + baseline$ (halves rounded away from zero); NaN becomes an invalid sample, and infinite values and values beyond the range of 32-bit ADC units are errors. Default `False`: ADC units.

samps_per_frame
For a multi-frequency record, the number of samples per frame of each signal (integers ≥ 1). `signals` then has one row per frame and `sum(samps_per_frame)` columns: the samples of signal 0, then those of signal 1, and so on. `fs` is then the frame frequency. Default `None`: one sample per frame for every signal.

metadata
Metadata entries (Chapter 5) to write to the header after the comments, as `# Key: value`: a mapping from key to value (strings), or a sequence of (key, value) pairs. Keys and values follow the rules of `set_metadata`; no two keys may differ only in letter case, nor be the key of an entry in `comments`. Default `None`: none.

dir The directory (which must exist) in which to create the files, for this call only. `record` must then be a plain name, without a directory, and the record opens from that directory under its plain name. Default `None`: the output directory.

legacy
`True` to behave as the original WFDB package: the gain defaults to 200, and the ADC resolution to one based on the format. Default `False`.

invalid_warning
`True` to issue a `RuntimeWarning` when samples will be read back as invalid (missing) samples ("N sample(s) of signal K will be read back as invalid (missing) samples (format F stores an invalid sample as V)"): those equal to the invalid value of the format or to `WFDB_INVALID_SAMPLE`, other than the NaN samples of physical units, which mark missing samples deliberately. `False` for no warning, when such samples are deliberate. Default `True`.

Returns None.

Errors `ValueError` if `record` is empty; if `signals` is not a 1-D or 2-D array ("signals must be a 1-D or 2-D array"); if it has no signals ("at least one signal is required"); if `fs`, a gain, an ADC resolution or the units of a calibrated signal is missing or invalid (`fs` and the gains must be finite, and `fs` > 0); if a baseline, an ADC zero or an ADC resolution is not a whole number within

the range of a 32-bit integer (“baselines must be whole numbers within the range of a 32-bit integer”); if the record name, a signal name, the units or a comment has a character that is not Latin-1 (“units ’XX’ has a character that is not Latin-1 (one byte per character)”); if `base_time` cannot be parsed; if a list of per-signal values does not have one value per signal; if a signal in physical units has a gain of 0; if a sample is not valid (see `signals` and `physical`); if the signals do not all have the same format; if a sample does not fit the format (“signal K has a sample (V) outside the range of format F (LO to HI)”; in format 8, “signal K changes by D between two samples, more than format 8 can store (-128 to 127)” or “signal K has an invalid sample, which format 8 cannot store”); if a FLAC format is given for more than 8 signals (“format 516 (FLAC) stores at most 8 signals in a file; the record has 12”); if a `samps_per_frame` value is not an integer ≥ 1, or the number of columns of `signals` is not their sum; if an entry of `metadata` is not valid; or if `record` contains a directory and `dir` is given. `TypeError` if `metadata` is neither a mapping nor a sequence of pairs, or a key or value is not a string. `FileNotFoundError` if `dir` does not exist. `IOError` if the signal file or the header cannot be created, or cannot be written completely, for example on a full disk (“write error on record REC”).

```python
import numpy as np
import wfdbx

# Two signals of random numbers, in millivolts, at 360 Hz: the record "rand"
# (rand.hea and rand.dat).
x = np.random.default_rng(1).standard_normal((3600, 2))
wfdbx.write_record("rand", x, fs=360, fmt=16, physical=True, gains=1000, adcres=16,
                   units="mV", names=["A", "B"], base_time="14:00:03 21/06/2024",
                   comments=["random numbers"], metadata=[("Generator", "numpy")])
rec = wfdbx.read_record("rand", physical=True)
print(rec.signals.shape, rec.base_time)       # (3600, 2) [14:00:03 21/06/2024]
print(np.abs(rec.signals - x).max() <= 0.0005)        # True: rounded to 1/1000 mV
print(wfdbx.read_header("rand").comments)     # ['random numbers', ' Generator: numpy']

# An uncalibrated signal (gain 0, no units), in ADC units, in the directory "out".
import os
os.makedirs("out", exist_ok=True)
wfdbx.write_record("counts", np.arange(1000), fs=100, gains=0, adcres=12, dir="out")
print(wfdbx.read_record("out/counts").signals[:3, 0])    # [0 1 2]
```

A multi-frequency record: signal `fast` has 2 samples per frame, signal `slow` one; each row of `signals` is one frame, with the two samples of `fast` and then the sample of `slow`:

```python
import numpy as np
import wfdbx

frames = 100                                    # 1 second at 100 frames per second
fast = np.arange(2 * frames).reshape(frames, 2)     # 200 samples per second
slow = np.full((frames, 1), 500)                    # 100 samples per second
wfdbx.write_record("mf", np.hstack([fast, slow]), fs=100, gains=200, adcres=12,
                   units="mV", names=["fast", "slow"], samps_per_frame=[2, 1])
rec = wfdbx.read_record("mf", highres=True)
print(rec.fs, rec.signals[:3].tolist())    # 200.0 [[0, 500], [1, 500], [2, 500]]
```

`write_annotations` Write an annotation file

```
write_annotations(record, annotator, samples, symbols=None, codes=None, subtypes=0, chans=0, nums
    ↪ =0, aux=None, fs=None, *, sort=None, dir=None, legacy=False)
```

Writes an annotation file, `RECORD.ANNOTATOR`, in the MIT format. An existing file of that name is replaced: the new file is written to a temporary file in the same directory (readable and writable by the owner only), which replaces it, with its permissions, only when it is complete (and, if needed, sorted); after an error, the temporary file is removed and the existing file is left as it was. The record need not have a header. The annotation types are given as `symbols` or as `codes` (Chapter 15). Each per-annotation argument is one value for all the annotations, or a list with exactly one value per annotation. If the annotations are not in time order, the file is sorted when it is written, unless `sort=False`. All the arguments are checked before the file is created or changed.

`record`
: Record name (a string), without the `.hea` extension (a trailing `".hea"` is also accepted): a plain name or a relative path (`"data/100s"`), created in the output directory if one is set, otherwise in the current directory; or a full path, absolute or in the home directory (`"~/Documents/100s"`), used as given. The file is `RECORD.ANNOTATOR` at that place (`data/100s.qrs`). A URL cannot be written.

`annotator`
: Annotator name: the suffix of the file, made of letters, digits, _, - and ~.

`samples`
: The annotation times, in samples: a 1-D array or sequence of integers, one per annotation, from $-2^{63}+1$ to $2^{63}-2$ (the smallest and largest 64-bit integers, `WFDB_TIME_MIN` and `WFDB_TIME_MAX`, cannot be written).

`symbols`
: The mnemonics of the annotation types, such as `"N"` or `"V"`: one per annotation, or one string for all. An unknown mnemonic becomes code 0 (`NOTQRS`). Default `None`: the types are given by `codes`.

`codes`
: The annotation codes (`wfdbx.NORMAL`, `wfdbx.PVC`, ...), from 0 to 58 (59 to 63 are the control codes of the file format, Section 15.3.1), instead of `symbols`: one per annotation, or one value for all. If both are given (as `read_annotations` returns them), each symbol must be that of its code (an empty symbol is not checked). Default `None`.

`subtypes`
: The subtype of each annotation, stored as a signed byte (other values wrap around to $-128 \ldots 127$, as in C). Default `0`.

`chans`
: The channel number of each annotation, stored as an unsigned byte (other values wrap around to $0 \ldots 255$). Default `0`.

`nums`
: The annotator number of each annotation, stored as a signed byte. Default `0`.

`aux` The auxiliary text of each annotation, with `None` where there is none: a list, or one text for all. Texts longer than 255 bytes are cut. Default `None`: none.

`fs` If given (a finite number > 0), the frequency in which the times are counted, written to the file; it applies to this file only. Default `None`: none is written.

`sort`
: `True` to always sort the file, `False` to keep the given order (without the library's advice to

run `sortann`), for this call only. Default `None`: the environment variable `WFDBANNSORT` decides (sorting is on unless it is `0`).

`dir` The directory (which must exist) in which to create the file, for this call only. `record` must then be a plain name, without a directory: the file is `DIR/RECORD.ANNOTATOR`. Default `None`: the output directory.

`legacy`
`True` to behave as the original WFDB package. Default `False`.

Returns `None`.

Errors `ValueError` if neither `symbols` nor `codes` is given for one annotation or more (`symbols or codes must be given`; with no annotations, an empty file is written); if both are given and disagree ("symbols and codes disagree at annotation K"); if `samples` (or a sequence of codes, subtypes, channels or annotator numbers) is not 1-D ("samples must be a 1-D array"); if a list of per-annotation values does not have one value per annotation; if a time is not an integer within the range of a 64-bit integer (NaN, an infinite value, a fraction or a huge value), or is -2^{63} or $2^{63}-1$ ("annotation time T cannot be written"); if a code is not from 0 to 58 ("annotation code 61 is reserved (59 to 63 are the control codes of the annotation file format)", "annotation code 64 is out of range (0 to 58)"); if a sample, code, subtype, channel or annotator number is NaN or `None`; if an aux text has a character that is not Latin-1; if `fs` is not finite and > 0; or if `record` contains a directory and `dir` is given. `FileNotFoundError` if `dir` does not exist. `IOError` if the annotation file cannot be created, or cannot be written completely, or the library (`putann`) reports an error while writing it ("write error on annotator ANN for record REC").

```
import wfdbx

# Three beats and a rhythm change, by symbol, with an aux string for the rhythm.
wfdbx.write_annotations("100s", "test", [18, 77, 370, 662],
                        symbols=["+", "N", "V", "N"], aux=["(N", None, None, None])
ann = wfdbx.read_annotations("100s", "test")
print(ann.symbols, ann.aux)     # ['+', 'N', 'V', 'N'] ['(N', None, None, None]

# A copy of the reference annotations, as read_annotations returns them.
a = wfdbx.read_annotations("100s", "atr")
wfdbx.write_annotations("100s", "copy", a.samples, symbols=a.symbols, codes=a.codes,
                        subtypes=a.subtypes, chans=a.chans, nums=a.nums, aux=a.aux)
b = wfdbx.read_annotations("100s", "copy")
print((b.samples == a.samples).all(), b.symbols == a.symbols, b.aux == a.aux)
# True True True

# Annotations out of time order are sorted when the file is written (the library
# reports it on the standard error).
wfdbx.write_annotations("100s", "unsorted", [500, 100], codes=wfdbx.NORMAL)
print(wfdbx.read_annotations("100s", "unsorted").samples)      # [100 500]

# A reserved code is refused before anything is written: the file is unchanged.
try:
    wfdbx.write_annotations("100s", "unsorted", [100, 200], codes=[wfdbx.NORMAL, 61])
except ValueError as e:
    print("ValueError:", e)   # ValueError: annotation code 61 is reserved (59 to 63 ...)
print(wfdbx.read_annotations("100s", "unsorted").samples)      # [100 500]
```

6.3.4 Checking records

check_record Check a record and its annotation files

```
check_record(record, annotators=(), *, input_dir=None, network=None)
```

Checks the header of a record, its signal specifications, every sample of its signal files, each segment of a multi-segment record, and the annotation files named, and reports each problem found. A problem in the record never raises an exception: it is reported, with a severity (`"error"`: the record is damaged or does not follow the WFDB format, and programs will read wrong data or fail; `"warning"`: the record can be read, but something is missing or unusual, and may give wrong results; `"info"`: a fact worth knowing, such as invalid samples). The problems, their codes and their messages are the same in the seven languages (Table 6.7). The checks run in normal mode, even if legacy mode is on, and the library prints no messages during the call. They are similar to those of Benjamin Moody's `wfdbcheck`.

record
: Record name (a string), as for `read_record`: a full path (absolute or ~), used as given; a relative path, read from the input directory if one is set, otherwise from the current directory; or a URL, read only if network access is on.

annotators
: The annotator names whose files are checked, such as `["atr"]`; one name may be given as a string. Default `()`: none.

input_dir
: The input directory for this call only (optional; as for `read_record`), under which relative record names are read; it does not apply to full paths or URLs. Default `None`: the current setting.

network
: `True` to allow reading URLs for this call only, `False` to forbid it. Default `None`: the current setting.

Returns A `CheckReport` (Table 6.5): `report.ok` is `True` if no problem is an error, `report.issues` lists the problems (`Issue` objects, Table 6.6), and `print(report)` prints one line per problem, or `no problems found`.

Errors `TypeError` if `record` or an annotator name is not a string; `ValueError` if `record` is empty. A record that cannot be read is reported (code `header_unreadable`), not raised.

```
import numpy as np
import wfdbx

report = wfdbx.check_record("100s", ["atr"])
print(report.ok)          # True: no errors
print(report)
# warning: signal 0: the units are not specified
# warning: signal 1: the units are not specified

# A damaged record: one sample of a 12-bit signal is overwritten with 2048.
wfdbx.write_record("damaged", np.arange(100), fs=100, gains=100, adcres=12, units="mV",
                   names="ramp")
with open("damaged.dat", "r+b") as f:
    f.seek(20)                    # sample 10 (2 bytes per sample in format 16)
```

```
    f.write(b"\x00\x08")          # 0x0800 = 2048, little-endian
report = wfdbx.check_record("damaged")
print(report.ok)          # False
for issue in report.issues:
    print(issue.severity, issue.code, issue.where)
# error sample_range signal 0
# error checksum signal 0
print(report.lines()[0])
# error: signal 0: 1 sample is outside the ADC range (-2048 to 2047): sample 10 (value 2048)
```

6.3.5 Header metadata

These functions read and change the metadata of a header: its info strings of the form `# Key: value`, such as `# Age: 63` (Chapter 5). In short: keys are compared without regard to letter case; a key must not be empty and must not contain `:`, `#`, a line break or a NUL character, and a value must not contain a line break or a NUL character; spaces at either end of keys and values are removed; text is one byte per character (Latin-1), and `bytes` are also accepted, as given. A change rewrites the header in one step, through a temporary file, keeping every other byte of it; all the arguments are checked first, so an error leaves the header unchanged. Only the `.hea` file is read (not a `.info` file); for a multi-segment record, it is the master header, unless a segment is named. Metadata often holds information about patients: WFDB-x stores it as plain text, exactly as given, and does not de-identify it.

> **Patient data** A header can hold identifying data about a patient (a name, a date of birth, a medical record number). The metadata functions read and write such data as given. Records that are shared must be de-identified first.

`read_metadata` — Read the metadata of a record

```
read_metadata(record, *, input_dir=None, network=None)
```

Reads the metadata entries of the header of a record, in file order. An info string is an entry if it contains a colon: the key is the text before the first colon, the value the rest of the line, both without spaces at either end. Info strings without a colon are free comments, which are not returned.

`record`
: Record name (a string), without the `.hea` extension (a trailing `".hea"` is also accepted): a full path, absolute (`"/data/mitdb/100"`) or in the home directory (`"~/Documents/100s"`), used as given; a relative path (`"100s"`, `"data/100s"`), read from the input directory if one is set, otherwise from the current directory; or a URL (`"https://physionet.org/files/mitdb/1.0.0/100"`), read only if network access is on.

`input_dir`
: The input directory for this call only (optional): a local directory, a ~ path or a URL, under which relative record names are read. It does not apply to full paths or URLs. Default `None`: the current setting.

`network`
: `True` to allow reading URLs for this call only, `False` to forbid it. Default `None`: the current setting.

Returns A list of (key, value) pairs of strings, in file order. A key that appears more than once (ignoring letter case) is returned once, with its first value, and a `UserWarning` is issued (Section 6.3.8).

Errors `IOError` if the header cannot be found or read; `ValueError` if `record` is empty or `"-"` (the standard input).

```
import numpy as np
import wfdbx

wfdbx.write_record("pat", np.zeros(360, dtype=int), fs=360, gains=200, adcres=12,
                   units="mV", names="ECG", comments=["recorded at rest"],
                   metadata={"Age": "63", "Sex": "F"})
print(wfdbx.read_metadata("pat"))      # [('Age', '63'), ('Sex', 'F')]
print(wfdbx.read_header("pat").comments)
# ['recorded at rest', ' Age: 63', ' Sex: F']: the free comment is not an entry
```

get_metadata

Return the value of a metadata entry

```
get_metadata(record, key, *, input_dir=None, network=None)
```

Returns the value of one metadata entry of the header of a record.

record
: Record name (as for `read_metadata`).

key The key, such as `"Age"`. Letter case is ignored, and spaces at either end are removed.

input_dir
: The input directory for this call only (optional; as for `read_metadata`): it applies to relative record names only. Default `None`: the current setting.

network
: `True` to allow reading URLs for this call only, `False` to forbid it. Default `None`: the current setting.

Returns The value, a string (`""` for an entry with an empty value), or `None` if the header has no entry for `key`. If it has more than one, the first is returned, and a `UserWarning` is issued.

Errors `IOError` if the header cannot be found or read; `ValueError` if `record` is empty or `"-"`, or if `key` is not a valid key; `TypeError` if `key` is not a string (or `bytes`).

```
import numpy as np
import wfdbx

wfdbx.write_record("pat", np.zeros(360, dtype=int), fs=360, gains=200, adcres=12,
                   units="mV", metadata={"Age": "63", "Sex": "F"})
print(wfdbx.get_metadata("pat", "age"))      # 63
print(wfdbx.get_metadata("pat", "Height"))   # None
```

set_metadata

Set the value of a metadata entry

```
set_metadata(record, key, value, *, input_dir=None, network=None, dir=None)
```

Sets the value of a metadata entry in the header of a record. If the header has an entry for `key` (ignoring letter case), its line is replaced by `# Key: value`, at the same position, with the spelling of `key` given here; otherwise the entry is added after the last info string of the header (or, if it has none, after its last line that is not blank). The entry is written as `# Key: value`, or `# Key:` if the value is empty. Every other line is kept, byte for byte, including its line ending.

`record`
: Record name (as for `read_metadata`). The header is changed where it was read, unless `dir` is given; a header read from a URL can only be written elsewhere, with `dir`.

`key` The key, such as `"Age"` or `"Date of birth"`. Spaces at either end are removed. It must not be empty, and must not contain `:`, `#`, a line break or a NUL character.

`value`
: The value: a string, which may be a sentence. Spaces at either end are removed. It must not contain a line break or a NUL character; it may be empty.

`input_dir`
: The input directory in which the header is read, for this call only (optional; as for `read_metadata`): it applies to relative record names only. Default `None`: the current setting.

`network`
: `True` to allow reading the header from a URL, for this call only, `False` to forbid it. Default `None`: the current setting.

`dir` A directory, which must exist, in which to write the changed header, under the same file name, instead of changing the header itself; the signal files are not copied. Default `None`: the header itself is changed.

Returns `None`.

Errors `IOError` if the header cannot be read, or the changed header cannot be written; `ValueError` if `key` or `value` is not valid, if the header has more than one entry for `key`, if it has no record line, if it was read from a URL and `dir` is not given, or if `record` is empty or `"-"`; `TypeError` if `key` or `value` is not a string (or `bytes`); `FileNotFoundError` if `dir` does not exist.

```
import os
import numpy as np
import wfdbx

wfdbx.write_record("pat", np.zeros(360, dtype=int), fs=360, gains=200, adcres=12,
                   units="mV", metadata={"Age": "63", "Sex": "F"})
wfdbx.set_metadata("pat", "AGE", "64")              # replaces "# Age: 63"
wfdbx.set_metadata("pat", "Diagnosis", "sinus rhythm, no ectopy")   # a new entry
print(wfdbx.read_metadata("pat"))
# [('AGE', '64'), ('Sex', 'F'), ('Diagnosis', 'sinus rhythm, no ectopy')]

# Write the changed header to another directory; pat.hea is left unchanged.
os.makedirs("changed", exist_ok=True)
wfdbx.set_metadata("pat", "Sex", "M", dir="changed")
print(wfdbx.get_metadata("pat", "Sex"), wfdbx.get_metadata("changed/pat", "Sex"))   # F M
```

`remove_metadata` — Remove a metadata entry

```
remove_metadata(record, key, *, input_dir=None, network=None, dir=None)
```

Removes every entry for `key` (ignoring letter case) from the header of a record, so it also repairs a header that has more than one. Other lines are kept, byte for byte. Nothing is written if the header has no entry for `key`, unless `dir` is given.

record
: Record name (as for `set_metadata`).

key
: The key (as for `set_metadata`).

input_dir
: The input directory in which the header is read, for this call only (optional; as for `read_metadata`): it applies to relative record names only. Default `None`: the current setting.

network
: `True` to allow reading the header from a URL, for this call only, `False` to forbid it. Default `None`: the current setting.

dir
: A directory, which must exist, in which to write the changed header instead of changing the header itself. Default `None`: the header itself is changed.

Returns The number of lines removed (an `int`; 0 if the header has no entry for `key`).

Errors As `set_metadata`, except that more than one entry for `key` is not an error.

```
import numpy as np
import wfdbx

wfdbx.write_record("pat", np.zeros(360, dtype=int), fs=360, gains=200, adcres=12,
                   units="mV", metadata={"Age": "63", "Sex": "F"})
print(wfdbx.remove_metadata("pat", "sex"))      # 1
print(wfdbx.remove_metadata("pat", "Sex"))      # 0: there is no such entry now
print(wfdbx.read_metadata("pat"))               # [('Age', '63')]
```

---

update_metadata

Make several changes to the metadata

```
update_metadata(record, changes, *, input_dir=None, network=None, dir=None)
```

Makes several changes to the metadata of a record, in one write of the header. Each change is applied in order, as `set_metadata` does, or, for a value of `None`, as `remove_metadata` does. All the changes are checked first: if one is not valid, none is made.

record
: Record name (as for `set_metadata`).

changes
: A mapping from key to value, or a sequence of (key, value) pairs; a value of `None` removes the entry. No two keys may differ only in letter case.

input_dir
: The input directory in which the header is read, for this call only (optional; as for `read_metadata`): it applies to relative record names only. Default `None`: the current setting.

network
: `True` to allow reading the header from a URL, for this call only, `False` to forbid it. Default `None`: the current setting.

dir
: A directory, which must exist, in which to write the changed header instead of changing the header itself. Default `None`: the header itself is changed.

Returns None.

Errors As `set_metadata`; also `ValueError` if two keys of `changes` differ only in letter case, and `TypeError` if `changes` is neither a mapping nor a sequence of pairs.

```
import numpy as np
import wfdbx

wfdbx.write_record("pat", np.zeros(360, dtype=int), fs=360, gains=200, adcres=12,
                   units="mV", metadata={"Age": "63", "Sex": "F"})
wfdbx.update_metadata("pat", {"Age": "64", "Sex": None, "Dx": "164889003"})
print(wfdbx.read_metadata("pat"))      # [('Age', '64'), ('Dx', '164889003')]
try:
    wfdbx.update_metadata("pat", [("Dx", "426783006"), ("dx", None)])
except ValueError as e:
    print(e)        # metadata key 'dx' is given more than once (...)
print(wfdbx.get_metadata("pat", "Dx"))  # 164889003: nothing was changed
```

6.3.6 The sample record and the example notebooks

`sample_data_dir` The directory of the sample record

```
sample_data_dir()
```

Returns the directory, within the installed package, that holds the sample record `100s` (Section 6.1.4): its header, signal file and reference annotations. The function has no parameters.

Returns The absolute path of the directory, a string.

Errors Nothing.

```
import os
import wfdbx

d = wfdbx.sample_data_dir()
print(sorted(f for f in os.listdir(d) if f.startswith("100s")))
# ['100s.atr', '100s.dat', '100s.hea']
rec = wfdbx.read_record("100s", sampto=10, input_dir=d)
print(rec.signals.shape)        # (10, 2)
```

`examples_dir` The directory of the example notebooks

```
examples_dir()
```

Returns the directory, within the installed package, that holds the example notebooks (`*.ipynb`, Table 6.1). The notebooks are meant to be copied elsewhere (`copy_examples`) before they are run. The function has no parameters.

Returns The absolute path of the directory, a string.

Errors Nothing.

```
import os
import wfdbx
```

```
print(sorted(os.listdir(wfdbx.examples_dir()))[0])     # 01_getting_started.ipynb
```

copy_examples Copy the example notebooks

```
copy_examples(dest)
```

Copies the example notebooks to a directory, created if needed, where they can be opened and run. Existing files of the same names are replaced. The command `wfdbx --examples DIR` does the same (Section 6.5).

dest
The directory into which the notebooks are copied (a path).

Returns The list of the paths of the copied notebooks (`os.path.join(dest, name)`), in the order of their names.

Errors `OSError` if the directory cannot be created or a notebook cannot be written.

```
import os
import wfdbx

paths = wfdbx.copy_examples("notebooks")
print(len(paths), os.path.basename(paths[0]))    # 7 01_getting_started.ipynb
```

6.3.7 The result classes

The readers return objects of the classes `Record`, `Header` and `Annotations`, whose attributes are listed in Tables 6.2 to 6.4; `check_record` returns a `CheckReport`, which holds `Issue` objects (Tables 6.5 and 6.6). The signals of a record are described by `Siginfo` objects (Table 6.8), which are those of the WFDB library (Section 6.4). All these classes are available at the top level of the package (`wfdbx.Record`, ...). The attributes of `Record`, `Header` and `Annotations` are plain Python attributes, which may be changed.

Table 6.2 The attributes of `Record`, the result of `read_record`

Attribute	Contents
`name`	the record name, as given to `read_record`
`fs`	the sampling frequency in Hz (a `float`), or 0 if the header gives none; with `highres=True`, the highest sampling frequency of a multi-frequency record
`signals`	a NumPy array with one row per sample and one column per signal read: `int32` ADC units, or `float64` physical units with `physical=True`; invalid samples are `WFDB_INVALID_SAMPLE`, or NaN in physical units
`siginfo`	a list of `Siginfo` objects, one per signal read, in the order of the columns; `desc` has no white space at either end, and `adcres` is 0 if the header does not give it
`names`	the description of each signal, such as `"MLII"`, without the white space that a header may put at either end of it
`units`	the physical units of each signal, or `None` where the header gives none
`gains`	the gain of each signal, in ADC units per physical unit (`float`); 0 means that the signal is uncalibrated
`baselines`	the ADC value of each signal that corresponds to 0 physical units

Attribute	Contents
base_time	the start time and date as text, to the second, such as "[14:00:03 21/06/2024]" ("[14:00:03]" without a date), or None if the header gives none
base_time_ms	the start time in milliseconds after midnight, such as 50403500 for 14:00:03.500, or None if base_time is None
base_date	the start date as a tuple (year, month, day), or None if the header gives none
sampfrom	the number of the first sample returned
comments	the list of the info strings of the header (without the #): its lines that begin with # after the record line, found as the header parser finds it (Section 2.2.3)
checksum_ok	False if the samples do not match the checksums of the header; checksums are checked only when a read reaches the end of the record, so a shorter read gives True

Table 6.3 The attributes of Header, the result of read_header

Attribute	Contents
name	the record name, as given to read_header
fs	the sampling frequency in Hz, or 0 if the header gives none
counter_freq	the counter frequency, as getcfreq() returns it (the sampling frequency if the header gives none)
base_counter	the base counter value, as getbasecount() returns it
nsamples	the number of samples per signal, or 0 if the header does not give it
nsig	the number of signals
siginfo	a list of Siginfo objects, one per signal; desc has no white space at either end, and adcres is 0 if the header does not give it
base_time, base_time_ms, base_date	the start time and date, as in Record
comments	the list of the info strings of the header (without the #), as in Record
metadata	the list of the (key, value) pairs of the metadata entries of the header, in file order, as read_metadata returns them

Table 6.4 The attributes of Annotations, the result of read_annotations. len(ann) is the number of annotations, which are in file order.

Attribute	Contents
record	the record name, as given
annotator	the annotator name, such as "atr"
samples	the annotation times, in samples (an int64 array)
codes	the annotation codes (an int64 array), such as wfdbx.NORMAL or wfdbx.PVC
symbols	the list of the mnemonics of the codes, such as "N", "V" or "+"
subtypes	the subtype of each annotation (an int64 array)
chans	the channel number of each annotation (an int64 array)
nums	the annotator number of each annotation (an int64 array)
aux	the list of the auxiliary texts of the annotations (strings), with None where there is none
fs	the frequency in which samples are counted, normally the sampling frequency of the record; 0 if neither the record, nor the file, nor the argument fs of read_annotations gives one

Two Issue objects are equal (==) when their four attributes are.

Table 6.5 The attributes and methods of `CheckReport`, the result of `check_record`

Attribute	Contents
`record`	the record name, as given
`ok`	`True` if no issue has the severity `"error"`
`issues`	the list of the `Issue` objects, in the order of the checks
`lines()`	the text form of the report, as a list of lines: one line per issue (`str(issue)`), or the one line `no problems found`
`str(report)`	the lines of `lines()`, joined by line feeds (what `print(report)` prints)

Table 6.6 The attributes of `Issue`, a problem found by `check_record`

Attribute	Contents
`severity`	`"error"`, `"warning"` or `"info"`
`code`	a stable identifier of the kind of problem, such as `"checksum"` or `"gain_missing"` (Table 6.7)
`where`	where the problem is: `""` (the record), `"signal N"`, `"segment SEG"`, `"segment SEG, signal N"` or `"annotator ANN"`
`message`	the description of the problem
`str(issue)`	`"SEVERITY: WHERE: MESSAGE"`, or `"SEVERITY: MESSAGE"` when `where` is empty

Table 6.7 The codes of the issues reported by `check_record`

Code	Severity	Problem
The header		
`header_unreadable`	error	the header cannot be read
`header_line_long`	error	a line of the header is longer than 255 characters
`header_not_ascii`	warning	a line of the header contains non-ASCII characters
`header_name`	warning	the header names a record other than its file name
`fs_missing`	warning	the sampling frequency is not specified
`length_missing`	warning	the record length is not specified
`length_large`	info	the record has more than 2147483647 samples
`no_signals`	info	the record has no signals
The signal specifications		
`desc_missing`	warning	the signal has no description
`desc_number`	warning	the description is only a number
`desc_control`	error	the description contains control characters
`desc_duplicate`	warning	several signals have the same description
`gain_missing`	warning	the signal is uncalibrated (no gain is given)
`gain_negative`	info	the gain is negative
`units_missing`	warning	the units are not specified
`units_control`	error	the units contain control characters
`adcres_missing`	warning	the ADC resolution is not specified
`adcres_too_large`	error	the ADC resolution exceeds what the format can store
`adc_range`	error	the ADC range does not fit in the format
`format_difference`	warning	format 8 (first differences) cannot follow steep slopes
`filename`	warning	the signal file name contains characters other than letters, digits, _, - and .

Code	Severity	Problem
The signal files		
`file_unreadable`	error	a signal file cannot be read
`file_large`	warning	a signal file has more than 2147483647 bytes
`read_error`	error	the signal data cannot be read after some frames
`sample_range`	error	samples are outside the ADC range
`invalid_samples`	info	samples are invalid (missing data)
`checksum`	error	the checksum of a signal does not match its samples
`length_short`	error	the signal files end before the length given in the header
`length_long`	warning	the signal files hold more than the length given in the header
`length_unspecified_frames`	info	the number of frames that the signal files hold, when the header does not give the length
The segments of a multi-segment record		
`segment_unreadable`	error	a segment cannot be read
`nested`	error	a segment is itself a multi-segment record
`segment_fs`	error	the sampling frequency of a segment differs from the record's
`segment_lengths`	warning	the segment lengths do not add up to the length in the master header
`layout_signals`	error	a segment has more signals than the record
`layout_mismatch`	error	a signal of a segment differs from the record's signal
`layout_missing_signal`	error	a signal of a segment is not in the layout
The annotation files		
`annotator_name`	warning	the annotator name consists only of digits
`annotations_unreadable`	error	the annotation file cannot be read
`annotations_truncated`	error	the annotation file ends unexpectedly
`annotations_order`	error	annotations are out of order
`annotation_code`	error	annotations have an invalid code
`annotation_null`	warning	annotations have code 0 (`NOTQRS`)
`annotation_undefined`	warning	annotations have a code without a defined mnemonic
`annotations_after_end`	warning	annotations are after the end of the record

Table 6.8 The attributes of `Siginfo`, the specification of a signal

Attribute	Contents
`fname`	the name of the signal file, as the header gives it
`desc`	the signal description, such as `"MLII"`, or `None`
`units`	the physical units, such as `"mV"`, or `None` if not specified
`gain`	the gain, in ADC units per physical unit (a `float`; 0: uncalibrated)
`initval`	the value of the first sample
`group`	the signal group number (the signals of one group share one file)
`fmt`	the storage format, such as `16`, `212` or `516`
`spf`	the number of samples per frame (more than 1 for the faster signals of a multi-frequency record)
`bsize`	the block size, in bytes (0 for ordinary files)
`adcres`	the ADC resolution, in bits (0: not specified)
`adczero`	the ADC output for an input of 0 V

Attribute	Contents
`baseline`	the ADC value that corresponds to 0 physical units
`nsamp`	the number of samples (0 if unknown)
`cksum`	the 16-bit checksum of the samples

6.3.8 Warnings and errors

Exceptions. The high-level functions report errors by exceptions of the standard classes (Table 6.9). Each function checks its arguments before it creates or changes a file, so an invalid argument leaves the files as they were; `write_record` and `write_annotations` also write through temporary files, so that a failure while writing (a full disk, for example) leaves the existing files as they were too. The library may also print a message on the standard error when it cannot open a file (as in `init: can't open header for record REC`); `wfdbquiet()` turns these messages off (Section 6.4).

Table 6.9 The exceptions of the high-level API

Exception	Raised when
`IOError` (`OSError`)	a header, signal file or annotation file cannot be found, read, created or written completely (for example on a full disk); a signal or annotation file ends before its header or its end-of-file marker says
`FileNotFoundError`	the directory given as `dir` does not exist (a subclass of `OSError`)
`ValueError`	an argument is invalid, or the operation needs a value that the record does not give (a time in seconds without a sampling frequency, physical units without a gain)
`TypeError`	an argument has the wrong type (a record or annotator name that is not a string in `check_record`; a metadata key or value that is not a string; metadata that is neither a mapping nor a sequence of pairs)

The checksum warning. When `read_record` reads to the end of a record whose samples do not match the checksums of its header, it returns all the samples and issues a `RuntimeWarning`: "checksum error in record REC: the signal data do not match the checksums in its header". `checksum_warning=False` turns the warning off; `checksum_ok` of the result is `False` either way. A signal file that ends early, or cannot be read, is an `IOError`, not a checksum warning, even when the checksum is verified in its last, incomplete frame: no incomplete frame is returned.

The duplicate-key warning. When a header holds two metadata entries whose keys differ only in letter case (written by other software), `read_metadata`, `get_metadata` and `read_header` use the first and issue a `UserWarning`: "header of record REC has more than one entry for key KEY (lines L1, L2); the first is used". `set_metadata` and `update_metadata` then refuse to change that key (`ValueError`), and `remove_metadata` removes all of its entries.

Both warnings are issued with the module `warnings` of Python, so they can be filtered, recorded or turned into errors (`warnings.simplefilter("error", RuntimeWarning)`) as any other warning:

```
import warnings
import numpy as np
import wfdbx

# A record whose last sample is changed after it was written: the checksum fails.
wfdbx.write_record("ck", np.arange(100), fs=100, gains=100, adcres=12, units="mV")
with open("ck.dat", "r+b") as f:
```

```
    f.seek(-2, 2)
    f.write(b"\x00\x00")
with warnings.catch_warnings(record=True) as caught:
    warnings.simplefilter("always")
    rec = wfdbx.read_record("ck")
print(rec.signals.shape, rec.checksum_ok)    # (100, 1) False: all the samples
print(caught[0].category.__name__)           # RuntimeWarning
rec = wfdbx.read_record("ck", checksum_warning=False)    # no warning

# Two entries for one key: the first is used, with a warning.
wfdbx.write_record("dup", np.zeros(10, dtype=int), fs=100, gains=0, adcres=12,
                   comments=[" Age: 63", " AGE: 64"])
with warnings.catch_warnings(record=True) as caught:
    warnings.simplefilter("always")
    print(wfdbx.get_metadata("dup", "age"))      # 63
print(caught[0].category.__name__)               # UserWarning
try:
    wfdbx.set_metadata("dup", "Age", "65")
except ValueError as e:
    print("ValueError:", e)
print(wfdbx.remove_metadata("dup", "age"))       # 2
```

```
import numpy as np
import wfdbx

try:
    wfdbx.read_record("nosuchrecord")
except OSError as e:                  # IOError is OSError
    print("OSError:", e)              # OSError: can't read record nosuchrecord

wfdbx.write_record("raw", np.arange(10), fs=100, gains=0, adcres=12)   # uncalibrated
try:
    wfdbx.read_record("raw", physical=True)
except ValueError as e:
    print("ValueError:", e)           # ValueError: signal 0 of record raw is uncalibrated
```

6.4 The WFDB library functions in Python

Every function of the WFDB library is available at the top level of the package under its C name, with the same arguments, in the same order, and the same return values, so that programs and documentation written for the C library carry over directly. The functions, their arguments and their results are documented once, for all the languages, in Chapter 13; this section gives the rules by which the C declarations become Python functions. `help(wfdbx.NAME)` gives the documentation of each function, including what WFDB-x changes in it.

6.4.1 Conventions

Names. The functions, the constants (`WFDB_READ`, `WFDB_INVALID_SAMPLE`, `WFDB_HIGHRES`, ...) and the annotation codes (`NORMAL`, `PVC`, ..., Chapter 15) keep their C names. The macros of `<wfdb/ecgmap.h>` are functions, under both their macro names and the names of the functions behind them: `isqrs` and `wfdb_isqrs`, `map1` and `wfdb_map1`, and so on. The structures are classes (Section 6.4.3).

NULL is `None`. A `NULL` pointer argument or result is `None`: `sampfreq(None)` returns the frequency of the open record, `getinfo(None)` continues with the info strings of the last record, and `getinfo`

returns `None` after the last one. Some trailing arguments have defaults (`nann=None` and `nsig=None`: all the entries of the list):

```
isigopen(record, siarray=None, nsig=0)
osigopen(record, siarray=None, nsig=0)
osigfopen(siarray, nsig=None)
setheader(record, siarray, nsig=None)
annopen(record, aiarray, nann=None)
```

Arrays and output arguments. Where a C function fills an array that the caller passes, the Python function fills a list that the caller passes:

- `isigopen` and `osigopen` empty the list `siarray` and append one `Siginfo` object (a copy) per signal; `siarray` may be `None`. With `nsig` ≤ 0, `isigopen` reads the header only, and returns the number of signals of the record; `osigopen(record, None, 0)` closes the open output signals, reads the header, opens nothing and returns 0.
- `getvec`, `getframe` and `putvec` take a list (or a NumPy array) of integers, one per signal (`getframe` and `putvec`: one per sample of a frame); `getvec` and `getframe` replace its first entries, and it must have room for them (`[0] * nsig`), or `IndexError` is raised.
- A C function that fills a structure through a pointer replaces the attributes of the object passed: `getann(n, annot)` fills an `Annotation`, `getcal(desc, units, cal)` a `Calinfo`. `putann(n, annot)` and `putcal(cal)` read them.
- An array of structures passed in is a list of objects: `annopen(record, [Anninfo("atr", WFDB_READ)])`, `osigfopen([Siginfo(...), ...])`.
- `getseginfo()` returns a list of `Seginfo` objects (empty if the open record is not a multi-segment record), instead of a pointer to an array.

Strings are `str`. Text and binary data are handled as ISO-8859-1 (Latin-1), one character per byte, so every byte sequence round-trips unchanged. The `aux` attribute of an annotation is `None`, or `bytes` whose first byte is the length of the text that follows, as in C; `Annotation.aux_string()` returns the text (up to its first NUL byte), or `None`.

Return values and errors. The functions return what the C functions return: an `int` status (negative on errors), a number of signals, a `float`, a string or `None`. Errors are reported by these values and by a message on the standard error, as in C, never by exceptions (only a misuse of Python, such as a vector that is too short, raises one). `wfdberror()` returns the last message; `wfdbquiet()` turns the messages off and `wfdbverbose()` on again.

State. As in C, the library keeps its state in the process (module-level state): the open record, its input signals, the output signals and the annotators remain open between calls, until `wfdbquit()` closes them. `wfdbquit()` leaves the input and output directories and the network setting as they are, and so, as in C, the time resolution of output annotators (`getafreq()`), which opening an EDF record sets to its sampling frequency when none is set, and the getvec mode (`getgvmode()`), to which it adds `WFDB_HIGHRES`. The high-level functions of Section 6.3 call `wfdbquit()`, which closes anything opened with the library functions, and leave those two settings as they were.

Additions. `getframes(n)` reads up to `n` frames at once, as `n` calls of `getvec` would, but decoding whole blocks of the signal files. It returns a tuple `(array, stat)`: an `int64` NumPy array with one row per frame read and one column per signal, and the `getvec` status of the first frame that could not be read, or the number of signals if all `n` frames were read.

6.4.2 Module layout

All the functions, classes and constants are imported by `import wfdbx`. They are defined in these modules of the package:

Table 6.10 The modules of the package `wfdbx`

Module	Contents
`wfdbx._wfdbio`	file lookup, the input and output directories, network access, error messages (`lib/wfdbio.c`); the settings functions
`wfdbx._signal`	headers, signal files in every format, time and date conversions, unit conversions (`lib/signal.c`); legacy mode
`wfdbx._annot`	annotation files, in the MIT and AHA formats (`lib/annot.c`)
`wfdbx._calib`	the calibration list (`lib/calib.c`)
`wfdbx._wfdbinit`	`wfdbinit`, `wfdbquit`, `wfdbflush` (`lib/wfdbinit.c`)
`wfdbx._ecgcodes`	the annotation codes and the mapping functions (`lib/ecgcodes.h`, `lib/ecgmap.h`)
`wfdbx._flac`	the FLAC codec of the formats 508, 516 and 524
`wfdbx._cstd`	the exact behavior of the C standard library functions that the WFDB library relies on (`strtol`, `sscanf`, `printf`, `...`)
`wfdbx.highlevel`	`read_record`, `read_header`, `read_annotations`, `write_record`, `write_annotations` and their result classes
`wfdbx.metadata`	the header metadata functions
`wfdbx.check`	`check_record`, `CheckReport`, `Issue`
`wfdbx.resources`	`sample_data_dir`, `examples_dir`, `copy_examples`
`wfdbx.cli`	the command `wfdbx`
`wfdbx.apps`	the command-line tools, one module per tool (`wfdbx.apps.rdsamp`, `...`)

The modules whose names begin with an underscore are internal: their functions are used through the top level of the package (`wfdbx.isigopen`, not `wfdbx._signal.isigopen`).

6.4.3 The structures

The structures of the C library are classes whose attributes have the names of the fields of the C structures (Table 6.11). Their constructors take the attributes as arguments, in this order, all with defaults:

```
Siginfo(fname=None, desc=None, units=None, gain=0.0, initval=0, group=0, fmt=0, spf=1,
        bsize=0, adcres=0, adczero=0, baseline=0, nsamp=0, cksum=0)
Seginfo(recname="", nsamp=0, samp0=0)
Anninfo(name=None, stat=WFDB_READ)
Annotation(time=0, anntyp=0, subtyp=0, chan=0, num=0, aux=None)
Calinfo(low=0.0, high=0.0, scale=0.0, sigtype=None, units=None, caltype=0)
```

`Siginfo`, `Annotation` and `Calinfo` have a method `copy()`; `Annotation` also has `assign(other)`, which copies the attributes of another annotation, and `aux_string()`.

Table 6.11 The classes of the C structures

Class (C structure)	Attributes
`Siginfo` (`WFDB_Siginfo`)	`fname`, `desc`, `units`, `gain`, `initval`, `group`, `fmt`, `spf`, `bsize`, `adcres`, `adczero`, `baseline`, `nsamp`, `cksum` (Table 6.8)

Class (C structure)	Attributes
`Seginfo` (`WFDB_Seginfo`)	`recname` (the name of the segment), `nsamp` (its number of samples), `samp0` (the number of its first sample in the record)
`Anninfo` (`WFDB_Anninfo`)	`name` (the annotator name, the suffix of the file), `stat` (`WFDB_READ` or `WFDB_WRITE` for the MIT format, `WFDB_AHA_READ` or `WFDB_AHA_WRITE` for the AHA format)
`Annotation` (`WFDB_Annotation`)	`time` (in sample intervals), `anntyp` (the annotation code), `subtyp`, `chan`, `num`, `aux` (`None`, or `bytes` whose first byte is the length of the text that follows)
`Calinfo` (`WFDB_Calinfo`)	`low`, `high` (the levels of the calibration pulse, in physical units), `scale` (physical units per cm), `sigtype` (matched against the beginning of a signal description), `units`, `caltype` (`WFDB_AC_COUPLED` or `WFDB_DC_COUPLED`, or-ed with `WFDB_CAL_SQUARE`, `WFDB_CAL_SINE` or `WFDB_CAL_SAWTOOTH`)

6.4.4 The settings functions

The functions of Table 6.12 are additions of WFDB-x to the library: they set where files are read and written, network access and legacy mode, for the high-level functions, the library functions and the tools alike (Chapters 3 and 4), and give the path under which a program opens a file that it reads or creates itself, as the tools do (`wfdbx_input_path`, `wfdbx_output_path`; Section 3.3.2). Each `set` function returns the previous setting, so that it can be restored. The C function `setwfdb(p)` sets the input directory too (`p` must name at most one directory), and `getwfdb()` returns it (`"."` if none is set); in legacy mode they set and return the WFDB path, as in C.

Table 6.12 The settings functions

Function	Effect and result
`wfdbx_set_input_dir(d)`	sets the input directory: a local directory, a ~ path or a URL; `None` or `""` restores the default (`WFDBX_INPUT_DIR`, or else the current directory). Returns the previous input directory, or `None` if none was set
`wfdbx_get_input_dir()`	returns the input directory, or `None` if none is set
`wfdbx_set_output_dir(d)`	sets the output directory: a local directory or a ~ path (a URL is refused, with a message); `None` or `""` restores the default (`WFDBX_OUTPUT_DIR`, or else the current directory). Returns the previous output directory, or `None` (the current one if `d` is refused)
`wfdbx_get_output_dir()`	returns the output directory, or `None` if none is set
`wfdbx_set_network(on)`	turns network access on (`True` or 1) or off (`False` or 0); `None` or a negative number restores the default (on if `WFDBX_NETWORK` is a nonzero number). Returns the previous setting, 1 or 0
`wfdbx_get_network()`	returns 1 if network access is on, 0 otherwise
`wfdbx_set_legacy(on)`	turns legacy mode on (`True` or 1) or off, for the whole library. Returns the previous setting, 1 or 0
`wfdbx_get_legacy()`	returns 1 if legacy mode is on, 0 otherwise
`wfdbx_legacy_args(argv)`	if `argv[1]` (`argv`: the argument list, program name first) is `"--legacy"`, removes it from the list, turns legacy mode on and returns 1; otherwise returns 0
`wfdbx_time_needs_freq(string)`	returns 1 if `strtim(string)` needs a sampling frequency, 0 otherwise
`wfdbx_write_errors()`	returns the number of write errors since the program started (output files that could not be written, flushed or closed)
`wfdbx_note_write_error()`	counts a write error, for a program that writes a file itself

Function	Effect and result
`wfdbx_input_path(name)`	returns the path under which a program opens a file that it reads itself: in normal mode, a relative name under the input directory, a ~ name expanded, an absolute name as given; in legacy mode, the name as given. `None`, `""` and `"-"` are returned as given; `None` for a URL, or a name that becomes one under the input directory
`wfdbx_output_path(name)`	returns the path under which a program creates a file that it writes itself: in normal mode, a relative name under the output directory (whose missing directories are then created), a ~ name expanded, an absolute name as given; in legacy mode, any name under the output directory, as a plain prefix. `None`, `""` and `"-"` are returned as given; `None` for a URL

6.4.5 Examples

Reading the samples and the annotations of a record, as a C program does (the record `100s` is read from the package's data directory):

```
import wfdbx

previous = wfdbx.wfdbx_set_input_dir(wfdbx.sample_data_dir())

si = []                                 # receives one Siginfo per signal
nsig = wfdbx.isigopen("100s", si, 2)    # open 2 signals
print(nsig, [(s.desc, s.fmt, s.gain) for s in si])   # 2 [('MLII', 212, 200.0), ...]
print(wfdbx.sampfreq(None))             # 360.0: the frequency of the open record

v = [0] * nsig                          # receives one sample per signal
wfdbx.isigsettime(wfdbx.strtim("0:30"))  # skip to 30 seconds
wfdbx.getvec(v)
print(v, [wfdbx.aduphys(k, v[k]) for k in range(nsig)])

frames, stat = wfdbx.getframes(360)     # the next second, as a NumPy array
print(frames.shape, stat)               # (360, 2) 2

a = wfdbx.Annotation()
wfdbx.annopen("100s", [wfdbx.Anninfo("atr", wfdbx.WFDB_READ)])
n = 0
while wfdbx.getann(0, a) == 0:          # 0: an annotation was read; -1: the end
    if n < 2:
        print(wfdbx.mstimstr(a.time), wfdbx.annstr(a.anntyp), a.aux_string())
    n += 1
print(n, "annotations")                 # 75 annotations

wfdbx.wfdbquiet()                       # no messages on the standard error
print(wfdbx.isigopen("nosuchrecord", None, 0), wfdbx.wfdberror().strip())
wfdbx.wfdbverbose()

wfdbx.wfdbquit()                        # close everything
wfdbx.wfdbx_set_input_dir(previous)     # restore the input directory
```

Writing a record and an annotation file, sample by sample and annotation by annotation:

```
import wfdbx
```

```
out = [wfdbx.Siginfo(fname="ramp.dat", desc="ramp", units="mV", gain=200.0, fmt=16,
                     adcres=12)]
print(wfdbx.osigfopen(out))         # 1: one output signal
wfdbx.setsampfreq(250.0)
for k in range(500):
    wfdbx.putvec([k])
print(wfdbx.newheader("ramp"))      # 0: ramp.hea is written
wfdbx.annopen("ramp", [wfdbx.Anninfo("qrs", wfdbx.WFDB_WRITE)])
for t in (100, 350):
    wfdbx.putann(0, wfdbx.Annotation(time=t, anntyp=wfdbx.NORMAL))
wfdbx.wfdbquit()                    # completes ramp.dat and ramp.qrs

print(open("ramp.hea").read().splitlines()[0])        # ramp 1 250 500
print(wfdbx.read_annotations("ramp", "qrs").samples)  # [100 350]
```

6.5 The command-line tools in Python

The 33 command-line tools (`rdsamp`, `rdann`, `wrsamp`, `xform`, ...) are run through the command `wfdbx`, installed with the package, followed by the name of the tool and its arguments. The tools, their options, their output and their exit status are those of the C tools, and are documented in Chapter 14.

```
export WFDBX_INPUT_DIR=$(python -c "import wfdbx; print(wfdbx.sample_data_dir())")
wfdbx rdsamp -r 100s -t 10 -p     # the first 10 seconds of 100s, in physical units
wfdbx rdann -r 100s -a atr        # its annotations
wfdbx rdsamp --legacy -r 100s     # legacy mode: --legacy as the first argument
```

The command `wfdbx` also takes these options, instead of a tool:

Table 6.13 The options of the command `wfdbx`

Command	Effect
`wfdbx TOOL [ARGUMENTS ...]`	runs the tool `TOOL`, and exits with its status
`wfdbx --list`	lists the names of the tools, one per line
`wfdbx --examples [DIR]`	copies the example notebooks to the directory `DIR` (default `wfdbx-examples`), and prints their paths
`wfdbx --version`	prints the version, `WFDB-x 1.2.3`
`wfdbx --help`, `wfdbx -h`	prints the usage

Without arguments, `wfdbx` prints the usage and exits with status 1; an unknown tool is an error (status 1). When an output file or the standard output cannot be written (a full disk, for example), a tool reports it and exits with status 1.

Without the `wfdbx` command. The package has no module `wfdbx.__main__`, so `python -m wfdbx` is not supported. The command is the module `wfdbx.cli`, and each tool is a module of `wfdbx.apps`, so that, with `python/src` on `PYTHONPATH` for example:

```
python -m wfdbx.cli rdsamp -r 100s -t 10     # as wfdbx rdsamp -r 100s -t 10
python -m wfdbx.apps.rdsamp -r 100s -t 10    # the tool's module itself
```

From Python, `wfdbx.cli.main(argv)` runs the command with the argument list `argv` (without the program name), and the function `main(argv)` of a tool's module runs the tool with `argv`, the complete argument list, program name first, as in C. Both return the exit status; the tool writes to the standard output and error, `sys.stdout` and `sys.stderr`. Its output can be captured

with `contextlib.redirect_stdout`, also into a text stream such as `io.StringIO` (the bytes are then written as Latin-1 text, one character per byte; until version 1.2.3, a text stream without a binary buffer gave a `TypeError`). The input and output directories of the tools are the process-wide settings (Section 6.4.4).

```
import contextlib, io
import wfdbx.cli
from wfdbx.apps import rdsamp

status = wfdbx.cli.main(["rdsamp", "-r", "100s", "-t", "s2"])   # the first 2 samples
print(status)                                                    # 0
out = io.StringIO()
with contextlib.redirect_stdout(out):
    rdsamp.main(["rdsamp", "-r", "100s", "-f", "s10", "-t", "s11", "-p"])
print(out.getvalue().splitlines()[0])   # the first line of the output
```

Environment variables. The tools, like the rest of the package, use the environment variables of Table 6.14, also for the files that they read or create themselves (Section 3.3.2): the `-i` file of `wrsamp`, `edf2mit`, `wav2mit`, `a2m` and `ad2m`, the input files of `ahaecg2mit` and the `-S` script of `xform` are read from the input directory, and the WAV file of `mit2wav`, the EDF file of `mit2edf`, the master header of `snip -m` and the `.des`, `.ecg`, `.key` and `.txt` files of `parsescp` are created in the output directory. Such a file cannot be read from a URL. In legacy mode, the files that a tool reads itself are opened as named, and the output directory, if set, is a plain prefix of every output name.

Table 6.14 The environment variables used by the Python package

Variable	Meaning
`WFDBX_INPUT_DIR`	the input directory (a directory, a ~ path or a URL), when `wfdbx_set_input_dir` does not set one
`WFDBX_OUTPUT_DIR`	the output directory, when `wfdbx_set_output_dir` does not set one
`WFDBX_NETWORK`	network access is on if it is a nonzero number, when `wfdbx_set_network` does not set it
`WFDBANNSORT`	`0` to keep annotation files in the order written (no sorting when they are closed)
`WFDBGVMODE`	the default getvec mode (`setgvmode` with a negative mode)
`WFDBCAL`	the calibration file read by `calopen(None)` (default `wfdbcal`)
`WFDB`	the WFDB path, used in legacy mode only (outside legacy mode, it is ignored, with a message)
`WFDB_FLAC_CHECK_MD5`	if set, the MD5 signature of FLAC signal files is verified when they are read

6.6 Testing

The tests of the Python package are in `python/tests`. From the root of the repository:

```
pip install -e "./python[test]"       # the package and pytest, nbformat, nbclient, ipykernel
pytest python/tests
```

This runs the unit tests of the library, the high-level functions, the metadata functions (with the cases shared by the seven languages, in `tests/metadata`) and `check_record`; it runs the example notebooks (when nbformat, nbclient and ipykernel are installed); and it runs the cross-language conformance suite, which compares the output of the tools with that of the original C programs, case by case (1155 cases, a few minutes). The tests run against `python/src` when they are run

from the repository, in a temporary copy of the test records of `tests/data`. The tests that write to `/dev/full` (write errors) are skipped where it does not exist, on macOS and Windows.

The environment variable `WFDBX_CONFORMANCE` selects the conformance cases that `pytest` runs: `0` skips the suite, and a list of tools, such as `WFDBX_CONFORMANCE=rdsamp,rdann`, runs the cases of those tools only.

```
WFDBX_CONFORMANCE=0 pytest python/tests            # the unit tests only
WFDBX_CONFORMANCE=rdsamp,rdann pytest python/tests/test_conformance.py
```

The conformance suite can also be run on its own, with more options (the number of parallel jobs, the legacy suite of 886 cases, which runs each tool with `--legacy`, verbose output, selected cases):

```
PYTHONPATH=$PWD/python/src python3 tests/conformance/harness.py check --impl python -j4
PYTHONPATH=$PWD/python/src python3 tests/conformance/harness.py check --impl python --legacy
```

CHAPTER 7

C++

The C++ implementation of WFDB-x is the library `libwfdbx`, with its headers in `cpp/include/wfdbx` and everything in namespace `wfdbx`, and the 33 command-line tools, built as separate programs. It is a complete C++17 implementation of the WFDB Software Package: it needs only the C++17 standard library, and never calls the C library.

It offers WFDB-x in three ways:

- **The high-level API** (Section 7.3): functions, declared in `<wfdbx/record.hpp>`, that read and write whole records, headers, annotation files and header metadata as `std::vectors` and structures, in one call, with options given as fields of a structure, and that throw `wfdbx::Error` on errors. They read signal files in large blocks, and are the fastest way to read a record. They are an addition of WFDB-x: the C package has no equivalent. They are the recommended interface for new code and analysis.
- **The WFDB library functions** under their C names (`isigopen`, `getvec`, `getann`, `putann`, ...), declared in `<wfdbx/wfdb.hpp>`, with the behavior of the C library (Section 7.4). They are documented once, for all the languages, in Chapter 13.
- **The command-line tools** (`rdsamp`, `rdann`, `wrsamp`, `xform` and 29 others), with the options, output and exit status of the C tools, one program per tool, run by its name (Section 7.5). They are documented in Chapter 14.

The high-level functions return exactly the samples and annotations that the library functions (and the C library) give.

7.1 Installation and requirements

7.1.1 Requirements

WFDB-x for C++ builds with CMake 3.14 or later and a C++17 compiler whose standard library has `std::filesystem`: GCC 9, Clang 9 or MSVC 2019 and later (macOS 10.15 or later), on Linux, macOS and Windows. It has no other dependency: no WFDB library and no libFLAC (the FLAC formats 508, 516 and 524 are read and written by WFDB-x's own codec). Network access (Section 3.4) runs the program `curl`, which must then be installed. The unit tests of `check_record` also run `python3` (Section 7.6).

7.1.2 Building and installing

From the root of a copy of the repository, these commands build the library, the tools and the unit tests (a Release build), and install them:

```
cmake -S cpp -B cpp/build                       # a Release build
cmake --build cpp/build -j8
cmake --install cpp/build --prefix /usr/local   # optional
```

The library is `cpp/build/libwfdbx.a`, and the tools are in `cpp/build/bin/`. The install step copies the library, the headers (`include/wfdbx/`), the tools, the CMake package files (`lib/cmake/wfdbx/`

`wfdbxConfig.cmake`, or `lib64/cmake/...` on the systems whose library directory is `lib64`, with the target `wfdbx::wfdbx`, and `wfdbxConfigVersion.cmake`) and the license files `LICENSE`, `LICENSE-GPL` and `NOTICE` of the repository (to `share/doc/wfdbx/`). The library is distributed under the LGPL (`LICENSE`), the tools under the GPL (`LICENSE-GPL`), and the FLAC decoder, a translation of the stream decoder of libFLAC, carries libFLAC's BSD-3-Clause notice; `NOTICE` gives these licenses and that notice, and a distribution of the library or the tools must include the three files. These options of the build change what is built:

Table 7.1 The options of the CMake build

Option	Default	Effect
`WFDBX_BUILD_TOOLS`	`ON`	build the command-line tools
`WFDBX_BUILD_TESTS`	`ON`	build the unit tests
`BUILD_SHARED_LIBS`	`OFF`	build a shared library
`CMAKE_BUILD_TYPE`	`Release`	the build type, when none is given

The version of the library is that of the CMake project, `1.2.3`. The tools, like the C tools, have no `--version` option.

7.1.3 Using the library from CMake

An installed library is found with `find_package`, which defines the imported target `wfdbx::wfdbx` (the library and its include directory):

```
find_package(wfdbx 1.2.3 CONFIG REQUIRED)
target_link_libraries(myprogram PRIVATE wfdbx::wfdbx)
```

The version is optional; any later version of the same major version is accepted. The headers need C++17, which the target requires of the programs that link to it, whatever their own `CMAKE_CXX_STANDARD`.

A copy of the directory `cpp` (for example in a subdirectory of a project) can also be built as part of a project with `add_subdirectory`, which defines the target `wfdbx` and its alias `wfdbx::wfdbx`. The options of Table 7.1 then select what is built; for the library alone:

```
set(WFDBX_BUILD_TOOLS OFF CACHE BOOL "Build the command-line tools" FORCE)
set(WFDBX_BUILD_TESTS OFF CACHE BOOL "Build the unit tests" FORCE)
add_subdirectory(path/to/wfdb-x/cpp ${CMAKE_CURRENT_BINARY_DIR}/wfdbx EXCLUDE_FROM_ALL)
target_link_libraries(myprogram PRIVATE wfdbx::wfdbx)
```

Without CMake, a program is compiled against the headers and linked with the library, installed (`-lwfdbx`) or in the build directory:

```
c++ -std=c++17 myprogram.cpp -lwfdbx                                  # installed
c++ -std=c++17 -Icpp/include myprogram.cpp cpp/build/libwfdbx.a -o myprogram
```

7.1.4 The sample record

The repository includes the sample record `100s`, the sample record of the WFDB Software Package: the first minute of record 100 of the MIT-BIH Arrhythmia Database, two ECG signals (`MLII` and `V5`) sampled at 360 Hz, with the reference beat annotations (annotator `atr`). Its files (`100s.hea`, `100s.dat`, `100s.atr`) are in the directory `tests/data` of a copy of the repository, with the other test records of WFDB-x. The C++ library does not install them: programs name that directory as their

input directory (`wfdbx_set_input_dir("tests/data")`). Work that uses the MIT-BIH Arrhythmia Database cites it as described on PhysioNet.

7.2 Quick start

This program reads the header, ten seconds of the signals and the annotations of the sample record `100s`, from the directory `tests/data` of the repository (the input directory of each call), and computes the mean heart rate. It then writes the first ten seconds as a new record, `first10`, in the current directory, with two metadata entries in its header.

```
#include <cstdio>
#include <vector>
#include <wfdbx/ecgmap.hpp>
#include <wfdbx/record.hpp>

int main() {
    wfdbx::InputOptions sample;
    sample.input_dir = "tests/data";            // the sample record 100s

    // The header only: no samples are read.
    wfdbx::Header hdr = wfdbx::read_header("100s", false, sample);
    std::printf("%d %g %lld\n", hdr.nsig, hdr.fs, hdr.nsamples);   // 2 360 21600
    std::printf("%s %s\n", hdr.siginfo[0].desc->c_str(),
                hdr.siginfo[1].desc->c_str());                     // MLII V5

    // Seconds 10 to 20 of both signals, frame by frame, in ADC units.
    wfdbx::ReadOptions opt;
    opt.input_dir = "tests/data";
    opt.from = "0:10";
    opt.to = "0:20";
    wfdbx::Record rec = wfdbx::read_record("100s", opt);
    std::printf("%zu %d\n", rec.nsamples(), rec.nsig);             // 3600 2
    std::printf("%d %g\n", rec.at(0, 0), rec.physical(0, 0));      // 946 -0.39

    // The beat annotations, and the mean heart rate.
    wfdbx::Annotations ann = wfdbx::read_annotations("100s", "atr", {}, {},
                                                     std::nullopt, false, sample);
    std::vector<long long> beats;
    for (size_t i = 0; i < ann.size(); i++)
        if (wfdbx::isqrs(ann.codes[i])) beats.push_back(ann.samples[i]);
    double rr = (beats.back() - beats.front()) / ann.fs / (beats.size() - 1);
    std::printf("%zu beats, %.1f beats per minute\n", beats.size(), 60 / rr);
    // 74 beats, 73.9 beats per minute

    // Write the first 10 seconds as the record "first10" (first10.hea and first10.dat,
    // in the current directory), with the signal specifications of 100s.
    opt.from = 0;
    opt.to = "0:10";
    wfdbx::Record x = wfdbx::read_record("100s", opt);
    wfdbx::WriteOptions w;
    w.gains = {200};
    w.baselines = {1024};
    w.adczeros = {1024};
    w.adcres = {11};
    w.units = {"mV"};
    w.names = x.names();
```

```
    w.metadata = {{"Source", "MIT-BIH record 100"}};
    wfdbx::write_record("first10", x.samples, x.nsig, x.fs, w);

    // Header metadata: "# Key: value" lines of the header.
    wfdbx::set_metadata("first10", "Age", "69");
    for (const auto &e : wfdbx::read_metadata("first10"))
        std::printf("%s = %s\n", e.first.c_str(), e.second.c_str());
    // Source = MIT-BIH record 100
    // Age = 69
    std::printf("%s\n", wfdbx::check_record("first10").to_string().c_str());
    // no problems found
}
```

Compiled and run from the root of the repository:

```
c++ -std=c++17 -Icpp/include quick.cpp cpp/build/libwfdbx.a -o quick
./quick
```

The header `100s.hea` gives no units: the program writes `"mV"` explicitly, since WFDB-x never assumes units, a gain, an ADC resolution or a sampling frequency that a record does not give (Chapter 4).

7.3 The high-level API

The high-level API is made of the functions of this section, declared in `<wfdbx/record.hpp>` in namespace `wfdbx`, of the structures of their options (Section 7.3.6) and of their results (Section 7.3.7), and of the exceptions `Error`, `ArgumentError` and `FileError` (Section 7.3.8). The comments of the header document each of them.

7.3.1 Conventions

Record names and directories. A record is named as in the other parts of WFDB-x (Chapters 2 and 3), without the `.hea` extension (the readers and `write_record` also accept a trailing `".hea"`, which is removed). A record name is one of: a full path, absolute (`"/data/mitdb/100"`) or in the home directory (`"~/Documents/100s"`), used as given; a relative path (`"100s"`, `"data/100s"`), read from the input directory if one is set, otherwise from the current directory, and written to the output directory if one is set, otherwise to the current directory (missing directories are created); or a URL (`"https://physionet.org/files/mitdb/1.0.0/100"`), read only if network access is on, and which cannot be written. The input and output directories are set with `wfdbx_set_input_dir` and `wfdbx_set_output_dir` (Section 7.4.3) or with the environment variables `WFDBX_INPUT_DIR` and `WFDBX_OUTPUT_DIR`. A writer given a name with a path creates the files in that directory, and the header names the record and its signal file without the path (`rec`, `rec.dat`), so that the record opens under any path that leads to it; in legacy mode, as in the original, the header gives the name with its path. The readers take the fields `input_dir` and `network` of an `InputOptions` (or of a structure derived from it), and the writers a `dir`: they apply to that one call, and the previous settings are restored when it returns, also when it throws. The `input_dir` option is optional, and applies to relative names only, not to full paths or URLs. With `dir`, the record name must be a plain name (`"rec"`): the files are created in `dir`, and the record opens from that directory under its plain name. The signal files that a header names are read from the header's directory.

Options. The options of a function are the fields of a structure (`ReadOptions`, `InputOptions`, `WriteOptions`, `MetadataOptions`), given as its last argument, with a default value `{}`: a field that is not set keeps its default. An option that may be absent is a `std::optional`.

Times. The fields `from` and `to` of `ReadOptions`, and the arguments `from` and `to` of `read_annotations`, are `TimeSpec` values: a sample number or a WFDB time string, as `strtim` reads it (`"0:10"`, 10 seconds; `"1:02:03.5"`; `"s3600"`, sample 3600; `"e"`, the end of the record; or a time of day in brackets, `"[14:00:00]"`). A `TimeSpec` converts from integers and strings, so `opt.from = 3600;` and `opt.from = "0:10";` both work; a default-constructed `TimeSpec` is empty (no time is given). A time in seconds needs the sampling frequency; when the record does not give it, such a time throws `Error`.

Samples. Samples are `WFDB_Sample` values (`int`), in ADC units, stored frame by frame in one `std::vector`: `samples[i * nsig + s]` is sample `i` of signal `s`. Invalid samples are `WFDB_INVALID_SAMPLE` (-32768), and NaN in physical units.

Per-signal and per-annotation values. The per-signal fields of `WriteOptions` (`gains`, `units`, ...) hold one value for all the signals, or one value per signal; the per-annotation fields of `Annotations` (`symbols`, `chans`, ...) are empty or hold one value per annotation. Any other length throws `Error` before anything is written.

Signal parameters. No sampling frequency, gain, ADC resolution or units is ever assumed. Where one is needed and the record does not give it, it must be given; otherwise the function throws `Error`. A record whose header gives no sampling frequency has `fs = 0`.

Text. Strings are `std::string`, handled as bytes: every byte sequence round-trips unchanged, and UTF-8 text is stored as its bytes.

Legacy mode. With `legacy` set to `true`, a function behaves as in the original WFDB package, with its assumed signal parameters and its search for files (Chapter 4); the library's legacy mode is on for the duration of the call only. With `false` (the default), the current setting of `wfdbx_set_legacy` applies, which is off unless it was turned on.

State. The functions that read and write records and annotation files call `wfdbquit()` before and after their work, also when they throw, which closes anything opened with the WFDB library functions. The library's settings (legacy mode, getvec mode, time resolution of output annotators, input and output directories) are the same after a call as before it, also when it throws. This includes the two settings that opening an EDF record changes, as in C, and that `wfdbquit()` does not reset: the time resolution of output annotators (`getafreq()`), set to the record's sampling frequency when none is set, and the getvec mode (`getgvmode()`), to which `WFDB_HIGHRES` is added; so a later `write_annotations` does not record that frequency. The C-style functions keep the behavior of C. The metadata functions do not call `wfdbquit()`: they open only the header, and leave the library's open files as they are. The functions are not thread-safe. Results are returned by value.

The examples of this section are the body of a function `main`, compiled after these lines, and run in a copy of the test records of `tests/data` (among them `100s`), which is the current directory and holds the records that the examples write:

```
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <filesystem>
#include <random>
#include <vector>
#include <wfdbx/ecgcodes.hpp>
#include <wfdbx/record.hpp>
using namespace wfdbx;
```

7.3.2 Reading records

read_record Read the signals of a record

```
Record read_record(const std::string &record, const ReadOptions &opt = {});
```

Reads the signals of a record from `opt.from` up to `opt.to` (exclusive), in ADC units, frame by frame, together with the description of the record. The samples are those that `getvec` returns (Chapter 13), read in large blocks with `getvecs`. If the record ends before `opt.to`, the samples up to its end are returned. When the read reaches the end of the record, the samples are compared with the checksums of the header (Section 7.3.8). No more than 2^{26} samples are allocated in advance, whatever length the header or `opt.to` gives; the result grows as the samples are read.

record
: Record name, without the `.hea` extension (a trailing `".hea"` is also accepted): a full path, absolute (`"/data/mitdb/100"`) or in the home directory (`"~/Documents/100s"`), used as given; a relative path (`"100s"`, `"data/100s"`), read from the input directory if one is set, otherwise from the current directory; or a URL (`"https://physionet.org/files/mitdb/1.0.0/100"`), read only if network access is on.

opt
: The options (Table 7.3): the signals to read (`channels`), the interval (`from`, `to`), `highres`, `pad`, `checksum_warning`, `legacy`, and the input directory, under which relative record names are read, and network access of the call (`input_dir`, `network`). Default `{}`: every signal of the whole record, with the current settings.

Returns A `Record` (Table 7.7), with the samples in ADC units.

Errors `Error` if the record cannot be read (“can’t read record REC”) or has no signals; if a requested signal does not exist (“record REC has no signal C”); if `opt.from` is negative (“read_record: from must not be negative”) or cannot be reached (“can’t seek to sample N of record REC”); if a time string needs the sampling frequency and the header gives none (“sampling frequency of record REC is not specified”); or if a signal file is shorter than its header says, or cannot be read or decoded (“can’t read signal data of record REC (getvec status -3)”), even if the checksums do not match either (the frame in which the file ends is then the one at which they are verified): an incomplete frame is never returned. Data that do not match the checksums of the header are returned, with a warning (see `checksum_warning`).

```
// Seconds 10 to 20 of lead MLII, in physical units.
ReadOptions opt;
opt.channels = {"MLII"};
opt.from = "0:10";
opt.to = "0:20";
Record rec = read_record("100s", opt);
std::vector<double> mlii = rec.physical_signal(0);
std::printf("%zu %g %lld %s\n", mlii.size(), rec.fs, rec.sampfrom,
            rec.names()[0].c_str());                    // 3600 360 3600 MLII

// The first 3 samples of both signals, in ADC units, V5 first.
ReadOptions first;
first.channels = {"1", "0"};
first.to = 3;
rec = read_record("100s", first);
for (size_t i = 0; i < rec.nsamples(); i++)
    std::printf("%d %d\n", rec.at(i, 0), rec.at(i, 1));   // 1011 995, 3 times
```

```
std::printf("%g %d\n", rec.siginfo[0].gain, rec.siginfo[0].baseline);   // 200 1024

// The whole record: the checksums are verified.
rec = read_record("100s");
std::printf("%zu %d\n", rec.nsamples(), rec.checksum_ok);   // 21600 1
```

A multi-frequency record (here the test record `mfreq`, whose signals have 1, 2 and 4 samples per frame at a frame frequency of 100 Hz) is read at its frame frequency, or at its highest sampling frequency:

```
Record rec = read_record("mfreq");
std::printf("%g %zu\n", rec.fs, rec.nsamples());       // 100 600
ReadOptions opt;
opt.highres = true;
rec = read_record("mfreq", opt);
std::printf("%g %zu\n", rec.fs, rec.nsamples());       // 400 2400
for (const WFDB_Siginfo &s : rec.siginfo) std::printf("%d ", s.spf);   // 1 2 4
std::printf("\n");
```

`read_header` — Read the header of a record

```
Header read_header(const std::string &record, bool legacy = false,
                   const InputOptions &input = {});
```

Reads the header of a record, without opening its signal files. The result describes the record and each of its signals, and holds the info strings and the metadata entries of the header (from the `.hea` file only, as `read_metadata` reads them, with its warning for a key given more than once). For an EDF record, which has no `.hea` file, and for a header read from the standard input (`"-"`), the metadata is empty, without an error. For a multi-segment record, the master header is read, and `nsamples` is the length of the whole record.

`record`
: Record name, without the `.hea` extension (a trailing `".hea"` is also accepted): a full path, absolute (`"/data/mitdb/100"`) or in the home directory (`"~/Documents/100s"`), used as given; a relative path (`"100s"`, `"data/100s"`), read from the input directory if one is set, otherwise from the current directory; or a URL (`"https://physionet.org/files/mitdb/1.0.0/100"`), read only if network access is on.

`legacy`
: `true` to read the header as the original WFDB package does. Default `false`.

`input`
: The input directory for this call only (optional), under which relative record names are read (it does not apply to full paths or URLs), and network access (Table 7.2). Default `{}`: the current settings.

Returns A `Header` (Table 7.8).

Errors `Error` if the header cannot be read (“can’t read header for record REC”).

```
Header hdr = read_header("100s");
std::printf("%s %g %d %lld\n", hdr.name.c_str(), hdr.fs, hdr.nsig, hdr.nsamples);
// 100s 360 2 21600
for (const WFDB_Siginfo &s : hdr.siginfo)
    std::printf("%s %s %d %g %d %s\n", s.desc->c_str(), s.fname->c_str(), s.fmt,
```

```
                s.gain, s.adcres, s.units ? s.units->c_str() : "(none)");
// MLII 100s.dat 212 200 11 (none)
// V5 100s.dat 212 200 11 (none)
std::printf("[%s]\n", hdr.comments[0].c_str());   // [ 69 M 1085 1629 x1]
std::printf("%d\n", hdr.base_time.has_value());   // 0: the header gives no start time
```

read_annotations

Read an annotation file

```
Annotations read_annotations(const std::string &record,
                             const std::string &annotator,
                             const TimeSpec &from = {},
                             const TimeSpec &to = {},
                             std::optional<WFDB_Frequency> fs = std::nullopt,
                             bool legacy = false,
                             const InputOptions &input = {});
```

Reads the annotations of an annotation file, in file order, as parallel vectors. The sampling frequency, needed to convert times in seconds, is that of the record's header, or, for a record without a header or whose header gives none, the argument `fs`, as the option `-F` of `rdann`. Whatever happens, the annotation file is closed (`wfdbquit()`) before the function returns.

record
: Record name, without the `.hea` extension (a trailing `".hea"` is also accepted): a full path, absolute (`"/data/mitdb/100"`) or in the home directory (`"~/Documents/100s"`), used as given; a relative path (`"100s"`, `"data/100s"`), read from the input directory if one is set, otherwise from the current directory; or a URL (`"https://physionet.org/files/mitdb/1.0.0/100"`), read only if network access is on. Its header, if there is one, gives the sampling frequency.

annotator
: Annotator name: the suffix of the annotation file. `"atr"` reads the file `RECORD.atr`.

from
: The first time to read: a sample number (≥ 0) or a time string. Default `{}`: the beginning of the file.

to
: The time at which to stop: annotations at or after it are not read. Default `{}`: the end of the file.

fs
: The sampling frequency, in Hz (> 0), to use when the record has no header or its header gives none; it is ignored otherwise. Default `std::nullopt`: none.

legacy
: `true` to behave as the original WFDB package (which assumes 250 Hz for a record without a frequency). Default `false`.

input
: The input directory for this call only (optional), under which relative record names are read (it does not apply to full paths or URLs), and network access (Table 7.2). Default `{}`: the current settings.

Returns An `Annotations` (Table 7.9). Its `fs` is the frequency in which the times are counted: the one written in the annotation file, if there is one, or else the sampling frequency; it is 0 if none of them is known.

Errors `Error` if `from` is a negative sample number ("read_annotations: from must not be negative", as `read_record` checks it, before anything is opened); if `fs` is given and is not > 0 ("read_

annotations: fs must be > 0"); if the annotation file cannot be opened ("can't read annotator ANN for record REC"); if a time string needs the sampling frequency and none is known ("sampling frequency of record REC is not specified"); or if the file ends early or cannot be read before `to` is reached ("can't read annotation file ANN of record REC (getann status S)", where S is the status that `getann` returned, such as -3 for an unexpected end of file).

```
Annotations ann = read_annotations("100s", "atr", "0:10", "0:30");
std::printf("%zu %g\n", ann.size(), ann.fs);                  // 24 360
std::printf("%lld %s\n", ann.samples[0], ann.symbols[0].c_str());   // 3862 N

// The whole file: the first annotation is a rhythm change, with an aux string.
ann = read_annotations("100s", "atr");
std::printf("%s %d %s\n", ann.symbols[0].c_str(), ann.codes[0] == RHYTHM,
            ann.aux[0]->c_str());                                // + 1 (N
```

For a record without a header, `fs` gives the frequency in which times in seconds are converted (the library reports on the standard error that it cannot open the header):

```
Annotations beats;
beats.samples = {100, 460, 820};
beats.symbols = {"N", "N", "N"};
write_annotations("beats", "qrs", beats);             // no header
Annotations ann = read_annotations("beats", "qrs", {}, "0:02", 360.0);
std::printf("%zu %g\n", ann.size(), ann.fs);          // 2 360
```

7.3.3 Writing records

`write_record` Write a header and a signal file

```
void write_record(const std::string &record,
                  const std::vector<WFDB_Sample> &samples, int nsig,
                  WFDB_Frequency fs, const WriteOptions &opt = {});
```

Writes samples in ADC units as a new record: the header `RECORD.hea` and one signal file, `RECORD.dat`, which holds all the signals (so they share one format). Existing files of these names are replaced: the header and the signal file are written to temporary files in the directories of their destinations (`.NAME.XXXXXXXX.tmp`), which replace the existing files, with their permissions (or, for a new file, 0666 less the umask), only when both are complete, so that an error leaves the existing files as they were, and no temporary file. For a symbolic link, the file it leads to is replaced; an existing file that is not a regular file (a device, a FIFO) is written directly. The sampling frequency, the gain and ADC resolution of each signal, and the units of each calibrated signal must be given: there are no defaults. Every check except those of the writing itself comes before any file is created or changed. (The ADC resolutions, zeros and baselines are `int`s, and the text is bytes, so that any value of these types can be written.)

`record`

Record name (not empty), without the `.hea` extension (a trailing `".hea"` is removed, as by the readers): a plain name (`"rec"`) or a relative path (`"data/rec"`), created in the output directory if one is set, otherwise in the current directory, missing directories included; or a full path, absolute or in the home directory (`"~/Documents/rec"`), used as given. With a path, the files are created in that directory, and the header names the record and its signal file without the path (`rec`, `rec.dat`), so that the record opens under any path that leads to it

(in legacy mode, as in the original, the header gives the name with its path). A URL cannot be written.

samples
The samples, in ADC units, frame by frame: `nsig` values per frame, or, for a multi-frequency record, the sum of `opt.samps_per_frame` (the samples of signal 0, then those of signal 1, and so on). `WFDB_INVALID_SAMPLE` marks invalid samples. The number of samples must be a multiple of the frame size.

nsig
The number of signals (> 0).

fs The sampling frequency, in Hz: a finite number > 0 (for a multi-frequency record, the frame frequency).

opt The options (Table 7.4): the format, the signal specifications (`gains`, `adcres` and `units` are required), the start time, the comments and metadata, the directory `dir` (which must exist) in which to create the files for this call only, with a plain record name, `legacy`, and `invalid_warning`.

Returns Nothing.

Errors `Error`, in this order: with `opt.dir`, if the record name contains a directory ("record name must not contain a directory when dir is given") or the directory does not exist ("directory D does not exist"); if the record name is empty; if a `samps_per_frame` value is less than 1 ("write_record: samps_per_frame must hold integers >= 1"); if `nsig` is not > 0 ("write_record: at least one signal is required"); if a per-signal field of `opt` has another length than 0, 1 or `nsig`; if the gains, the ADC resolutions (each at least 1) or the units of a calibrated signal are missing ("write_record: gains must be specified", "write_record: ADC resolution(s) must be specified", "write_record: units must be specified"); if `fs` is not finite or not > 0; if a gain is not finite ("write_record: gains must be finite"); if `opt.base_time` is not a valid time ("write_record: incorrect base time 'T'"); if `samples.size()` is not a multiple of the frame size; if a FLAC format is given for more than 8 signals ("write_record: format 516 (FLAC) stores at most 8 signals in a file; the record has 12"); if a sample does not fit the format ("write_record: signal K has a sample (V) outside the range of format F (LO to HI)"; in format 8, "write_record: signal K changes by D between two samples, more than format 8 can store (-128 to 127)" or "write_record: signal K has an invalid sample, which format 8 cannot store"), the samples of each signal being checked in order (for a multi-frequency signal, its samples of each frame, frame after frame); `ArgumentError` (a subclass of `Error`) if an entry of `opt.metadata` is not valid ("write_record: metadata key 'K' is also given in comments", for example); then `Error` if the files cannot be created, or if they cannot be written completely, for example on a full disk ("write error on record REC").

```
// A ramp of 1000 samples of one signal, in millivolts, at 100 Hz: the record "ramp"
// (ramp.hea and ramp.dat).
std::vector<WFDB_Sample> x(1000);
for (int i = 0; i < 1000; i++) x[i] = i;
WriteOptions w;
w.gains = {200};
w.adcres = {12};
w.units = {"mV"};
w.names = {"ramp"};
w.base_time = "14:00:03 21/06/2024";
w.comments = {"a ramp"};
w.metadata = {{"Generator", "loop"}};
write_record("ramp", x, 1, 100, w);
Record rec = read_record("ramp");
std::printf("%zu %s %g\n", rec.nsamples(), rec.base_time->c_str(),
```

```
            rec.physical(10, 0));                    // 1000 [14:00:03 21/06/2024] 0.05
for (const std::string &c : read_header("ramp").comments)
    std::printf("[%s]\n", c.c_str());
// [a ramp]
// [ Generator: loop]
```

A multi-frequency record: signal `fast` has 2 samples per frame, signal `slow` one; each frame holds the two samples of `fast` and then the sample of `slow`:

```
std::vector<WFDB_Sample> frames;
for (int i = 0; i < 100; i++) {          // 1 second at 100 frames per second
    frames.push_back(2 * i);             // fast: 200 samples per second
    frames.push_back(2 * i + 1);
    frames.push_back(500);               // slow: 100 samples per second
}
WriteOptions w;
w.gains = {200};
w.adcres = {12};
w.units = {"mV"};
w.names = {"fast", "slow"};
w.samps_per_frame = {2, 1};
write_record("mf", frames, 2, 100, w);
ReadOptions opt;
opt.highres = true;
Record rec = read_record("mf", opt);
std::printf("%g %d %d %d %d\n", rec.fs, rec.at(0, 0), rec.at(0, 1), rec.at(1, 0),
            rec.at(1, 1));                        // 200 0 500 1 500
```

`write_record_physical` — Write a record from samples in physical units

```
void write_record_physical(const std::string &record,
                           const std::vector<double> &samples, int nsig,
                           WFDB_Frequency fs, const WriteOptions &opt = {});
```

As `write_record`, with the samples in physical units. Each sample v is stored as $\mathrm{round}(v \cdot gain) + baseline$ (halves rounded away from zero), and NaN is stored as an invalid sample. Every signal must have a gain other than 0.

`record`
: Record name, as for `write_record`.

`samples`
: The samples, in physical units, frame by frame (as for `write_record`).

`nsig`
: The number of signals (> 0).

`fs` The sampling frequency, in Hz: a finite number > 0.

`opt` The options, as for `write_record` (Table 7.4).

Returns Nothing.

Errors As `write_record`; also `Error` if a gain is 0 ("write_record: signal N is uncalibrated"), or if a sample is infinite or its ADC value is outside the range of a 32-bit integer ("write_record: sample I is out of range"). Nothing is written then.

```
// Two signals of random numbers, in millivolts, at 360 Hz: the record "rand".
std::mt19937 gen(1);
std::normal_distribution<double> randn;
std::vector<double> x(3600 * 2);                    // 3600 frames of 2 signals
for (double &v : x) v = randn(gen);
WriteOptions w;
w.fmt = 16;
w.gains = {1000};
w.adcres = {16};
w.units = {"mV"};
w.names = {"A", "B"};
write_record_physical("rand", x, 2, 360, w);
Record rec = read_record("rand");
double err = 0;
for (size_t i = 0; i < x.size(); i++)
    err = std::max(err, std::abs(rec.physical(i / 2, i % 2) - x[i]));
std::printf("%zu %d\n", rec.nsamples(), err <= 0.0005);   // 3600 1: rounded to 1/1000 mV
```

write_annotations

Write an annotation file

```
void write_annotations(const std::string &record, const std::string &annotator,
                       const Annotations &ann, WFDB_Frequency fs = 0,
                       const std::string &dir = {},
                       std::optional<bool> sort = std::nullopt);
```

Writes an annotation file, `RECORD.ANNOTATOR`, in the MIT format; an existing file of that name is replaced as by `write_record`: the new file is written to a temporary file in the same directory, which replaces it only when it is complete (and, if needed, sorted), so that an error leaves it as it was. The record need not have a header. The type of each annotation is its `symbols` entry (`"N"`, `"V"`, ...; an unknown mnemonic gives `NOTQRS`), or its `codes` entry (Chapter 15). Both may be given (as `read_annotations` returns them) if they agree: the code of each non-empty symbol (`strann`) is the code given for that annotation; an empty symbol leaves the code. `subtypes`, `chans` and `nums` are stored in 8 bits, as by `putann` (300 is stored as 44, as in C), and `aux` texts are cut to 255 bytes. If the annotations are not in time order, the file is sorted when it is written. The arguments are checked before the file is created or changed. The times must be from $-2^{63}+1$ to $2^{63}-2$ (the smallest and largest 64-bit integers are reserved by the file format), and the codes from 0 to 58 (59 to 63 are the control codes of the annotation file format). `fs` applies to this file only: the time resolution of output annotators (`getafreq()`) is left as it was.

record
: Record name, without the `.hea` extension (a trailing `".hea"` is also accepted): a plain name or a relative path (`"data/100s"`) is created in the output directory if one is set, otherwise in the current directory; a full path, absolute or in the home directory (`"~/Documents/100s"`), is used as given. The file is `RECORD.ANNOTATOR` at that place (`data/100s.qrs`). A URL cannot be written.

annotator
: Annotator name: the suffix of the file, made of letters, digits, _, - and ~.

ann
: The annotations (Table 7.9): `samples`, and `symbols` or `codes` (required when there is at least one annotation; with no annotations, an empty file is written); `subtypes`, `chans`, `nums` and `aux` are optional. Each of these vectors is empty (`subtypes`, `chans` and `nums` are then 0, and

there is no aux) or has one entry per annotation. `ann.record`, `ann.annotator` and `ann.fs` are not used.

fs 0 (none, the default) or a finite frequency > 0, in which the times are counted, recorded in the file.

dir If not empty, the directory (which must exist) in which to create the file, for this call only; `record` must then be a plain name, without a directory: the file is `DIR/RECORD.ANNOTATOR`. Default `{}`: as `record` says.

sort

`true` to always sort the file, `false` to keep the given order (without the library's advice to run `sortann`), for this call only. Default `std::nullopt`: the environment variable `WFDBANNSORT` decides (sorting is on unless it is `0`).

Returns Nothing.

Errors `Error`, in this order: with `dir`, as `write_record` does; if a per-annotation field has another length than 0 or the number of annotations ("write_annotations: symbols must have one entry per annotation (N), or none", for example); if there is at least one annotation and neither `symbols` nor `codes` is given ("write_annotations: symbols or codes must be given"); if `fs` is not 0 and not a finite frequency > 0; if a time is reserved ("write_annotations: annotation time T cannot be written (annotation times must be integers from -9223372036854775807 to 9223372036854775806)"); if a code is not from 0 to 58 ("write_annotations: annotation code C is reserved (59 to 63 are the control codes of the annotation file format)", or "write_annotations: annotation code C is out of range (0 to 58)"); if `symbols` and `codes` are both given and disagree ("write_annotations: symbols and codes disagree at annotation K", K counted from 0); then if the file cannot be created ("can't create annotator ANN for record REC"); if it cannot be written completely, or `putann` reports an error ("write error on annotator ANN for record REC").

```
// Three beats and a rhythm change, by symbol, with an aux string for the rhythm.
Annotations a;
a.samples = {18, 77, 370, 662};
a.symbols = {"+", "N", "V", "N"};
a.aux = {std::string("(N"), std::nullopt, std::nullopt, std::nullopt};
write_annotations("100s", "test", a);
Annotations b = read_annotations("100s", "test");
std::printf("%s %s %s\n", b.symbols[0].c_str(), b.symbols[2].c_str(),
            b.aux[0]->c_str());                          // + V (N

// A copy of the reference annotations, as read_annotations returns them.
Annotations atr = read_annotations("100s", "atr");
write_annotations("100s", "copy", atr);
Annotations c = read_annotations("100s", "copy");
std::printf("%d\n", c.samples == atr.samples && c.aux == atr.aux);   // 1

// Annotations out of time order are sorted when the file is written (the library
// reports it on the standard error).
Annotations u;
u.samples = {500, 100};
u.codes = {NORMAL, NORMAL};
write_annotations("100s", "unsorted", u);
Annotations s = read_annotations("100s", "unsorted");
std::printf("%lld %lld\n", s.samples[0], s.samples[1]);    // 100 500
```

7.3.4 Checking records

check_record Check a record and its annotation files

```
CheckReport check_record(const std::string &record,
                         const std::vector<std::string> &annotators = {},
                         const InputOptions &input = {});
```

Checks the header of a record, its signal specifications, every sample of its signal files, each segment of a multi-segment record, and the annotation files named, and reports each problem found. A problem in the record never throws: it is reported, with a severity (`"error"`: the record is damaged or does not follow the WFDB format, and programs will read wrong data or fail; `"warning"`: the record can be read, but something is missing or unusual, and may give wrong results; `"info"`: a fact worth knowing, such as invalid samples). A signal file that ends early, or cannot be read, ends the reading before its incomplete frame, also in the frame at which the checksums are verified (it is reported as `length_short` or `read_error`, not as a checksum error). The problems, their codes and their messages are the same in the seven languages (Table 6.7). The checks run in normal mode, even if legacy mode is on, and the library prints no messages during the call (the previous settings are restored afterwards). They are similar to those of Benjamin Moody's `wfdbcheck`.

record
: Record name, without the `.hea` extension (a trailing `".hea"` is also accepted): a full path, absolute (`"/data/mitdb/100"`) or in the home directory (`"~/Documents/100s"`), used as given; a relative path (`"100s"`, `"data/100s"`), read from the input directory if one is set, otherwise from the current directory; or a URL (`"https://physionet.org/files/mitdb/1.0.0/100"`), read only if network access is on (as for `read_record`).

annotators
: The annotator names whose files are checked, such as `{"atr"}`. Default `{}`: none.

input
: The input directory for this call only (optional), under which relative record names are read (it does not apply to full paths or URLs), and network access (Table 7.2). Default `{}`: the current settings.

Returns A `CheckReport` (Table 7.10): `report.ok` is `true` if no problem is an error, `report.issues` lists the problems (`Issue` values, Table 7.11), and `report.to_string()` gives one line per problem, or `no problems found`.

Errors `Error` if `record` is empty ("record must not be empty"). A record that cannot be read is reported (code `header_unreadable`), not thrown.

```
CheckReport report = check_record("100s", {"atr"});
std::printf("%d\n", report.ok);                    // 1: no errors
std::printf("%s\n", report.to_string().c_str());
// warning: signal 0: the units are not specified
// warning: signal 1: the units are not specified

// A damaged record: one sample of a 12-bit signal is overwritten with 2048.
std::vector<WFDB_Sample> x(100);
for (int i = 0; i < 100; i++) x[i] = i;
WriteOptions w;
w.gains = {100};
```

```
w.adcres = {12};
w.units = {"mV"};
w.names = {"ramp"};
write_record("damaged", x, 1, 100, w);
std::FILE *f = std::fopen("damaged.dat", "r+b");
std::fseek(f, 20, SEEK_SET);           // sample 10 (2 bytes per sample in format 16)
std::fputc(0x00, f);                   // 0x0800 = 2048, little-endian
std::fputc(0x08, f);
std::fclose(f);
report = check_record("damaged");
std::printf("%d\n", report.ok);                    // 0
for (const Issue &i : report.issues)
    std::printf("%s %s %s\n", i.severity.c_str(), i.code.c_str(), i.where.c_str());
// error sample_range signal 0
// error checksum signal 0
std::printf("%s\n", report.lines()[0].c_str());
// error: signal 0: 1 sample is outside the ADC range (-2048 to 2047): sample 10 (value 2048)
```

7.3.5 Header metadata

These functions read and change the metadata of a header: its info strings of the form `# Key: value`, such as `# Age: 63` (Chapter 5). In short: keys are compared without regard to letter case (A-Z and a-z only); a key must not be empty and must not contain `:`, `#`, CR, LF or NUL, and a value must not contain CR, LF or NUL; spaces, tabs, VT, FF and CR at either end of keys and values are removed; keys and values are bytes (`std::string`). A change rewrites the header in one step, through a temporary file in the directory of the destination, which then replaces it, keeping its permissions and every other byte of it; all the arguments are checked first, so an error leaves the header unchanged, and nothing is written if nothing changes (unless `dir` is given). Only the `.hea` file is read (not a `.info` file); for a multi-segment record, it is the master header, unless a segment is named. The results and arguments are of the types `Metadata` and `MetadataChanges`:

```
using Metadata = std::vector<std::pair<std::string, std::string>>;
using MetadataChanges =
    std::vector<std::pair<std::string, std::optional<std::string>>>;
```

Errors throw `ArgumentError` (an invalid argument) or `FileError` (a file that cannot be read or written), both subclasses of `Error` (Section 7.3.8). These functions do not call `wfdbquit()`. Metadata often holds information about patients: WFDB-x stores it as plain text, exactly as given, and does not de-identify it.

> **Patient data** A header can hold identifying data about a patient (a name, a date of birth, a medical record number). The metadata functions read and write such data as given. Records that are shared must be de-identified first.

`read_metadata` — Read the metadata of a record

```
Metadata read_metadata(const std::string &record,
                       const InputOptions &input = {});
```

Reads the metadata entries of the header of a record, in file order. The info strings are the lines that begin with `#` after the record line, which is found as the header parser finds it, in every mode: the first line that holds something other than spaces, tabs, CRs and LFs and does not begin, after them, with `#` (comment lines before it, even after an empty or blank first line, are never read

or changed). An info string is an entry if it contains a colon: the key is the text before the first colon, the value the rest of the line, both without spaces at either end. Info strings without a colon, or whose key is empty or contains `#`, are free comments, which are not returned.

record
: Record name, without the `.hea` extension (a trailing `".hea"` is also accepted): a full path, absolute (`"/data/mitdb/100"`) or in the home directory (`"~/Documents/100s"`), used as given; a relative path (`"100s"`, `"data/100s"`), read from the input directory if one is set, otherwise from the current directory; or a URL (`"https://physionet.org/files/mitdb/1.0.0/100"`), read only if network access is on.

input
: The input directory for this call only (optional), under which relative record names are read (it does not apply to full paths or URLs), and network access (Table 7.2). Default `{}`: the current settings.

Returns The entries, as (key, value) pairs, in file order. A key that appears more than once (ignoring letter case) is returned once, with its first value, with a warning (Section 7.3.8).

Errors `FileError` if the header cannot be found or read ("can't read header for record REC"); `ArgumentError` if `record` is empty ("record name must not be empty") or `"-"` (the standard input).

```
WriteOptions w;
w.gains = {200};
w.adcres = {12};
w.units = {"mV"};
w.names = {"ECG"};
w.comments = {"recorded at rest"};
w.metadata = {{"Age", "63"}, {"Sex", "F"}};
write_record("pat", std::vector<WFDB_Sample>(360, 0), 1, 360, w);
for (const auto &e : read_metadata("pat"))
    std::printf("%s = %s\n", e.first.c_str(), e.second.c_str());
// Age = 63
// Sex = F
for (const std::string &c : read_header("pat").comments)
    std::printf("[%s]\n", c.c_str());
// [recorded at rest]: the free comment is not an entry
// [ Age: 63]
// [ Sex: F]
```

get_metadata

Return the value of a metadata entry

```
std::optional<std::string> get_metadata(const std::string &record,
                                        const std::string &key,
                                        const InputOptions &input = {});
```

Returns the value of one metadata entry of the header of a record.

record
: Record name (as for `read_metadata`).

key The key, such as `"Age"`. Letter case is ignored, and spaces at either end are removed; it must be a valid key (see `set_metadata`).

input
: The input directory and network access for this call only. Default `{}`: the current settings.

Returns The value (`""` for an entry with an empty value), or `std::nullopt` if the header has no entry for `key`. If it has more than one, the first is returned, with the warning of `read_metadata`.

Errors As `read_metadata`; also `ArgumentError` if `key` is not valid.

```
WriteOptions w;
w.gains = {200};
w.adcres = {12};
w.units = {"mV"};
w.metadata = {{"Age", "63"}, {"Notes", ""}};
write_record("pat", std::vector<WFDB_Sample>(360, 0), 1, 360, w);
std::printf("%s\n", get_metadata("pat", "age")->c_str());       // 63
std::printf("[%s]\n", get_metadata("pat", " NOTES ")->c_str()); // []: an empty value
std::printf("%d\n", get_metadata("pat", "Weight").has_value()); // 0: no such entry
```

---

set_metadata

Set the value of a metadata entry

```
void set_metadata(const std::string &record, const std::string &key,
                  const std::string &value, const MetadataOptions &opt = {});
```

Sets the value of a metadata entry in the header of a record. If the header has an entry for `key` (ignoring letter case), its line is replaced by `# Key: value`, at the same position and with its line ending, with the spelling of `key` given here (even if the value is the same). Otherwise the entry is added after the last info string of the header (or, if it has none, after its last line that is not blank), with the line ending of the record line. Every other byte of the header is kept.

record
: Record name (as for `read_metadata`). The header is changed where it was read, unless `opt.dir` is given; a header read from a URL can only be written elsewhere, with `opt.dir`.

key
: The key, such as `"Age"` or `"Date of birth"`. Spaces at either end are removed; it must not then be empty, and must not contain `:`, `#`, CR, LF or NUL.

value
: The value, which may be a sentence. Spaces at either end are removed; it must not contain CR, LF or NUL, and may be empty (it is then written `# Key:`).

opt
: The input directory and network access for reading the header, and the directory `dir` in which to write the changed header, for this call only (Table 7.5). Default `{}`: the current settings, and the header is changed where it was read.

Returns Nothing.

Errors `ArgumentError` if `key` or `value` is not valid; if the header has more than one entry for `key` (“header of record REC has more than one entry for key KEY (lines L1, L2); remove them first”); if it has no record line; if it was read from a URL and `opt.dir` is not given; or if `record` is empty or `"-"`. `FileError` if the header cannot be read; if the changed header cannot be written (“can’t write header for record REC”); or if `opt.dir` does not exist (“directory D does not exist”).

```
WriteOptions w;
w.gains = {200};
w.adcres = {12};
w.units = {"mV"};
w.metadata = {{"Age", "63"}};
write_record("pat", std::vector<WFDB_Sample>(360, 0), 1, 360, w);
set_metadata("pat", "AGE", "64");                  // replaces the line: "# AGE: 64"
```

```cpp
set_metadata("pat", "Date of birth", "1961-02-03");    // adds "# Date of birth: 1961-02-03"
for (const auto &e : read_metadata("pat"))
    std::printf("%s = %s\n", e.first.c_str(), e.second.c_str());
// AGE = 64
// Date of birth = 1961-02-03
try {
    set_metadata("pat", "a:b", "1");
} catch (const ArgumentError &e) {
    std::printf("%s\n", e.what());      // metadata key 'a:b' must not contain ':'
}
```

remove_metadata

Remove a metadata entry

```cpp
int remove_metadata(const std::string &record, const std::string &key,
                    const MetadataOptions &opt = {});
```

Removes a metadata entry from the header of a record. Every entry for `key` (ignoring letter case) is removed, with its line ending, so this also repairs a header that has more than one. Every other byte is kept. Nothing is written if the header has no entry for `key` (unless `opt.dir` is given).

record
: Record name (as for `set_metadata`).

key The key (as for `set_metadata`).

opt The input directory, network access and output directory `dir` of this call (as for `set_metadata`). Default `{}`.

Returns The number of lines removed (0 if the header has no entry for `key`).

Errors As `set_metadata`.

```cpp
WriteOptions w;
w.gains = {200};
w.adcres = {12};
w.units = {"mV"};
w.metadata = {{"Age", "63"}, {"Sex", "F"}};
write_record("pat", std::vector<WFDB_Sample>(360, 0), 1, 360, w);
std::printf("%d\n", remove_metadata("pat", "sex"));       // 1
std::printf("%d\n", remove_metadata("pat", "Weight"));    // 0: nothing is written
std::printf("%zu\n", read_metadata("pat").size());        // 1
```

update_metadata

Make several changes to the metadata

```cpp
void update_metadata(const std::string &record, const MetadataChanges &changes,
                     const MetadataOptions &opt = {});
```

Makes several changes to the metadata of a record in one write. Each (key, value) of `changes` is applied in order, as `set_metadata` does, or, if the value is `std::nullopt`, as `remove_metadata` does. All the changes are checked first: if one is not valid, none is made.

record
: Record name (as for `set_metadata`).

changes
: The changes, as (key, value) pairs; a value of `std::nullopt` removes the entry. No two keys may differ only in letter case. Empty `changes` change nothing (and write nothing, unless `opt.dir` is given).

opt
: The input directory, network access and output directory `dir` of this call (as for `set_metadata`). Default `{}`.

Returns Nothing.

Errors As `set_metadata`; also `ArgumentError` if two keys differ only in letter case ("metadata key 'K' is given more than once (keys are compared without regard to letter case)").

```
WriteOptions w;
w.gains = {200};
w.adcres = {12};
w.units = {"mV"};
w.metadata = {{"Age", "63"}, {"Dx", "164889003"}};
write_record("pat", std::vector<WFDB_Sample>(360, 0), 1, 360, w);
update_metadata("pat", {{"Age", "64"}, {"Dx", std::nullopt}, {"Sex", "F"}});
for (const auto &e : read_metadata("pat"))
    std::printf("%s = %s\n", e.first.c_str(), e.second.c_str());
// Age = 64
// Sex = F

// The header "pat" in the directory "in" is unchanged; the changed copy is written to
// the directory "out" (which must exist).
std::filesystem::create_directories("in");
std::filesystem::create_directories("out");
std::filesystem::copy_file("pat.hea", "in/pat.hea");
MetadataOptions o;
o.input_dir = "in";
o.dir = "out";
update_metadata("pat", {{"Age", "65"}}, o);
std::printf("%s %s\n", get_metadata("in/pat", "Age")->c_str(),
            get_metadata("out/pat", "Age")->c_str());      // 64 65
```

7.3.6 The options

The options of the high-level functions are the fields of the structures of Tables 7.2 to 7.5. `ReadOptions` and `MetadataOptions` are derived from `InputOptions`, and have its fields too. A field that is not set keeps its default.

Table 7.2 The fields of `InputOptions`: where `read_header`, `read_annotations`, `check_record`, `read_metadata` and `get_metadata` find files, for one call

Field	Meaning
`std::optional<std::string> input_dir`	the input directory for this call only (optional; see `wfdbx_set_input_dir`): a local directory, a ~ path or the URL of a directory, under which relative record names are read. It does not apply to full paths or URLs. `""` means the default (`WFDBX_INPUT_DIR`, or the current directory). Not set: the current setting
`std::optional<bool> network`	`true` to allow opening URLs for this call only, `false` to forbid it (see `wfdbx_set_network`). Not set: the current setting

Table 7.3 The fields of `ReadOptions`, the options of `read_record` (with `input_dir` and `network` of `InputOptions`)

Field	Meaning (default)
`std::vector<std::string> channels`	the signals to return, in this order, by number (`"0"`, `"1"`, ...) or by name (`"V5"`, as in `Record::names()`, then as `findsig` finds it). Default empty: all the signals
`TimeSpec from`	the first sample: a sample number (≥ 0) or a time string. Default empty: 0
`TimeSpec to`	the sample after the last one: a sample number or a time string. Default empty: the end of the record
`bool highres`	`true` to read a multi-frequency record at its highest sampling frequency, repeating the samples of the slower signals; `false` averages the samples of the faster signals to one per frame. Default `false`
`bool pad`	`true` to replace each invalid sample by the previous valid sample of the same signal. Default `false`
`bool checksum_warning`	`true` to print a warning on the standard error when the samples do not match the checksums of the header (Section 7.3.8); the samples are returned either way. Default `true`
`bool legacy`	behave as the original WFDB package, for this call only. Default `false`

Table 7.4 The fields of `WriteOptions`, the options of `write_record` and `write_record_physical`. Each per-signal vector holds one value for all the signals, or one value per signal (or none, for those that have a default)

Field	Meaning (default)
`int fmt`	the storage format, the same for all the signals, such as `16`, `212`, `24` or `516` (FLAC) (Chapter 15). Every sample must fit the format: within its range (Table 15.1; -2048 to 2047 for format 212, for example), and, in format 8, within -128 to 127 of the sample before it. The smallest value of a format is the value of an invalid (missing) sample: a sample equal to it, or to `WFDB_INVALID_SAMPLE`, is read back as invalid (see `invalid_warning`). A FLAC file holds at most 8 signals. Default `16`
`std::vector<WFDB_Gain> gains`	the gains, in ADC units per physical unit: required, and finite. A gain of 0 marks a signal as uncalibrated, which only `write_record` (ADC units) allows. Negative gains are allowed
`std::vector<int> baselines`	the ADC values that correspond to 0 physical units. Default: the `adczeros` values
`std::vector<int> adczeros`	the ADC output for an input of 0 V. Default `{0}`
`std::vector<int> adcres`	the ADC resolutions, in bits: required, each at least 1
`std::vector<std::optional<std::string>> units`	the physical units, such as `"mV"`: required for every signal whose gain is not 0; optional for uncalibrated signals
`std::vector<std::string> names`	the signal descriptions. Default: `""`
`std::vector<int> samps_per_frame`	the number of samples per frame of each signal, for a multi-frequency record (each ≥ 1). Default: 1
`std::optional<std::string> base_time`	the start time and date, such as `"12:00:00 01/01/2020"` (see `setbasetime`). Default: none

Field	Meaning (default)
`std::vector<std::string> comments`	the info strings, written to the header as comment lines (`"#"` followed by the string). Default: none
`Metadata metadata`	metadata entries (Chapter 5) written to the header after the comments, as `# Key: value` (through `putinfo`), in this order. The keys and values follow the rules of `set_metadata`; no two keys may differ only in letter case, and none may be the key of an entry in `comments` (`ArgumentError`, before any file is created). Default: none
`std::string dir`	if not empty, the directory (which must exist) in which the files are created, for this call only; the record name must then be a plain name, without a directory, and the record opens from that directory under its plain name. Default `""`: the output directory
`bool legacy`	behave as the original WFDB package, for this call only: the gain defaults to 200, and the ADC resolution to one based on the format. Default `false`
`bool invalid_warning`	`true` to print a warning on the standard error when samples will be read back as invalid (missing) samples (Section 7.3.8): those equal to the invalid value of the format or to `WFDB_INVALID_SAMPLE`, other than the NaN samples of `write_record_physical`, which mark missing samples deliberately; `false` for no warning, when such samples are deliberate. Default `true`

Table 7.5 The fields of `MetadataOptions`, the options of `set_metadata`, `remove_metadata` and `update_metadata` (with `input_dir` and `network` of `InputOptions`, which say where the header is read from)

Field	Meaning
`std::string dir`	if not empty, the directory (which must exist; a ~ path is expanded) in which the changed header is written, under the file name of the header that was read, instead of changing that header. The header is then written even if nothing changes. The signal files are not copied. Default `""`: the header is changed where it was read

Table 7.6 The members of `TimeSpec`, a sample number or a time string

Member	Meaning
`TimeSpec()`	an empty value: no time is given
`TimeSpec(WFDB_Time t)`, `TimeSpec(int t)`, `TimeSpec(long t)`	the sample number `t`
`TimeSpec(const char *s)`, `TimeSpec(std::string s)`	the time string `s`
`bool empty() const`	`true` if no time was given
`bool needs_freq() const`	`true` if the time is a string that needs a sampling frequency to be converted (see `wfdbx_time_needs_freq`)
`WFDB_Time resolve() const`	the sample number (0 if empty); a time string is converted with `strtim` for the record open at the time, and a time of day gives the positive sample number it corresponds to

7.3.7 The result types

The readers return structures of the types `Record`, `Header` and `Annotations`, whose members are listed in Tables 7.7 to 7.9; `check_record` returns a `CheckReport`, which holds `Issue` values (Tables 7.10 and 7.11). The signals of a record are described by `WFDB_Siginfo` structures (Table 7.12), which are those of the WFDB library (Section 7.4). Their fields are public, and may be changed.

Table 7.7 The members of `Record`, the result of `read_record`

Member	Contents
`std::string name`	the record name, as given to `read_record`
`WFDB_Frequency fs`	the sampling frequency of the returned samples, in Hz, or 0 if the header gives none; with `highres`, the highest sampling frequency of a multi-frequency record
`int nsig`	the number of signals returned
`std::vector<int> signals`	the number, in the record, of each signal returned
`std::vector<WFDB_Sample> samples`	the samples in ADC units, frame by frame: `samples[i * nsig + s]` is sample `i` of signal `s`; invalid samples are `WFDB_INVALID_SAMPLE`
`std::vector<WFDB_Siginfo> siginfo`	one per signal returned, in the order of the signals; `desc` has no white space at either end
`std::optional<std::string> base_time`	the start time and date as text, to the second, such as `"[08:45:00 01/01/2001]"`; `std::nullopt` if the header gives none
`std::optional<long long> base_time_ms`	the start time in milliseconds after midnight (`"14:00:03.500"` gives `50403500`); given exactly when `base_time` is
`std::optional<std::array<int, 3>> base_date`	the start date as `{year, month, day}`; `std::nullopt` if the header gives no date
`WFDB_Time sampfrom`	the number of the first sample returned
`std::vector<std::string> comments`	the info strings of the record: the comments of the header (its lines that begin with `#` after the record line, without the `#`), and the lines of its `.info` file if it has one (see `getinfo`). The record line is found as the header parser finds it: the first line that holds something other than spaces, tabs, CRs and LFs and does not begin, after them, with `#`; an empty or blank first line is not the record line, except in legacy mode (where the C rule of `getinfo` applies)
`bool checksum_ok`	`false` if the samples do not match the checksums of the header; checksums are checked only when a read reaches the end of the record, so a shorter read gives `true`
`bool legacy`	`true` if the record was read in legacy mode; `physical()` then uses a gain of 200 for uncalibrated signals
`size_t nsamples() const`	the number of samples per signal
`WFDB_Sample at(size_t i, int s) const`	sample `i` of signal `s`, in ADC units (not bounds-checked)

Member	Contents
`double physical(size_t i, int s) const`	sample `i` of signal `s` in physical units, $(value - baseline)/gain$; NaN if the sample is invalid. Throws `Error` ("signal N of record REC is uncalibrated") if the gain of the signal is 0
`std::vector<double> physical_signal(int s) const`	all the samples of signal `s` in physical units
`std::vector<std::string> names() const`	the signal descriptions (`""` if absent), without the white space that a header may put at either end of them

Table 7.8 The members of `Header`, the result of `read_header`

Member	Contents
`std::string name`	the record name, as given to `read_header`
`WFDB_Frequency fs`	the sampling frequency in Hz, as `sampfreq()` gives it; 0 if the header gives none
`WFDB_Frequency counter_freq`	the counter frequency (`getcfreq()`)
`double base_counter`	the base counter value (`getbasecount()`)
`WFDB_Time nsamples`	the number of samples per signal, from the record line of the header (0 if the header does not give it)
`int nsig`	the number of signals
`std::vector<WFDB_Siginfo> siginfo`	the signal specifications, one per signal; `desc` has no white space at either end
`base_time`, `base_time_ms`, `base_date`	the start time and date, as in `Record`
`std::vector<std::string> comments`	the info strings of the record, as in `Record`
`Metadata metadata`	the metadata entries of the header, in file order, as `read_metadata` returns them: from the `.hea` file only; empty, without an error, for an EDF record and for a header read from the standard input

Table 7.9 The members of `Annotations`, the result of `read_annotations` and the argument of `write_annotations`: parallel vectors, annotation `i` being `samples[i]`, `codes[i]`, `symbols[i]`, …, in file order

Member	Contents
`std::string record, annotator`	the record and annotator names, such as `"100s"` and `"atr"`
`WFDB_Frequency fs`	the frequency in which `samples` are counted, normally the sampling frequency of the record; 0 if unknown
`std::vector<WFDB_Time> samples`	the annotation times, in samples
`std::vector<int> codes`	the annotation codes, such as `NORMAL` or `PVC`
`std::vector<std::string> symbols`	the mnemonics of the codes, as `annstr` returns them, such as `"N"`, `"V"` or `"+"`
`std::vector<int> subtypes, chans, nums`	the subtype, channel number and annotator number of each annotation
`std::vector<std::optional<std::string>> aux`	the auxiliary text of each annotation, up to its first NUL byte, or `std::nullopt` where there is none
`size_t size() const`	the number of annotations

Table 7.10 The members of `CheckReport`, the result of `check_record`

Member	Contents
`std::string record`	the record name, as given
`bool ok`	`true` if no issue has the severity `"error"`
`std::vector<Issue> issues`	the problems, in the order of the checks
`std::vector<std::string> lines() const`	the text form of the report: one line per issue (`Issue::to_string()`), or the one line `no problems found`
`std::string to_string() const`	the lines of `lines()`, joined by line feeds (no final line feed)

Table 7.11 The members of `Issue`, a problem found by `check_record`

Member	Contents
`std::string severity`	`"error"`, `"warning"` or `"info"`
`std::string code`	a stable identifier of the kind of problem, such as `"checksum"` or `"gain_missing"` (Table 6.7)
`std::string where`	where the problem is: `""` (the record), `"signal N"`, `"segment SEG"`, `"segment SEG, signal N"` or `"annotator ANN"`
`std::string message`	the description of the problem
`std::string to_string() const`	`"SEVERITY: WHERE: MESSAGE"`, or `"SEVERITY: MESSAGE"` when `where` is empty

Table 7.12 The fields of `WFDB_Siginfo` (also named `Siginfo`), the specification of a signal

Field	Contents
`std::optional<std::string> fname`	the name of the signal file, as the header gives it
`std::optional<std::string> desc`	the signal description, such as `"MLII"`
`std::optional<std::string> units`	the physical units, such as `"mV"`; `std::nullopt` if not specified
`WFDB_Gain gain`	the gain, in ADC units per physical unit (0: uncalibrated)
`WFDB_Sample initval`	the value of the first sample
`WFDB_Group group`	the signal group number (the signals of one group share one file)
`int fmt`	the storage format, such as `16`, `212` or `516`
`int spf`	the number of samples per frame (more than 1 for the faster signals of a multi-frequency record)
`int bsize`	the block size, in bytes (0 for ordinary files)
`int adcres`	the ADC resolution, in bits (0: not specified)
`int adczero`	the ADC output for an input of 0 V
`int baseline`	the ADC value that corresponds to 0 physical units
`long nsamp`	the number of samples (0: not specified)
`int cksum`	the 16-bit checksum of the samples

7.3.8 Warnings and errors

Exceptions. The high-level functions report errors by throwing exceptions of the classes of Table 7.13, all declared in `<wfdbx/record.hpp>`. `e.what()` returns the message, which describes the failed operation, such as “can’t read record REC”. Each function checks its arguments before it creates or changes a file, so an invalid argument leaves the files as they were; the writers also write through temporary files, so that a failure while writing leaves the existing files as they were. A signal file that ends before its header says, or cannot be read or decoded, is a read

error of `read_record` (“can’t read signal data of record REC (getvec status -3)”, after the library’s message “getvec: unexpected EOF in signal N”), even in the frame at which the checksums are verified, and `check_record` reports it as `length_short` or `read_error`; a checksum mismatch alone is a warning. The library may also print a message of its own on the standard error when it cannot open a file (as in `init: can't open header for record REC`); `wfdberror()` returns the last such message, and `wfdbquiet()` turns these messages off (Section 7.4).

Table 7.13 The exceptions of the high-level API

Exception	Thrown when
`Error` (derived from `std::runtime_error`)	a record, header or annotation file cannot be found, read, created or written completely (for example on a full disk); a signal or annotation file ends before its header or its end-of-file marker says; an argument is invalid; or the operation needs a value that the record does not give (a time in seconds without a sampling frequency, physical units without a gain)
`ArgumentError` (derived from `Error`)	an argument of a metadata function is invalid: an invalid key or value, a record name that is empty or `"-"`, a key that the header has more than once (`set_metadata`, `update_metadata`), a header without a record line, or a header read from a URL to be changed without `dir`; also an invalid `WriteOptions::metadata` in `write_record`
`FileError` (derived from `Error`)	a metadata function cannot read or write a file: a header that cannot be found or read, a changed header that cannot be written, or a `dir` that does not exist

Warnings. Warnings are printed on the standard error with the library’s messages, as `wfdb_error` prints them: `wfdbquiet()` silences them, and `wfdberror()` returns the last one. There is no callback.

The checksum warning. When `read_record` reads to the end of a record whose samples do not match the checksums of its header, it returns all the samples and prints the warning “WFDB-x: checksum error in record REC: the signal data do not match the checksums in its header”. `checksum_warning = false` turns the warning off; `checksum_ok` of the result is `false` either way.

The invalid-sample warning. When `write_record` or `write_record_physical` writes samples that will be read back as invalid (missing) samples (those equal to the smallest value of the format or to `WFDB_INVALID_SAMPLE`, other than the NaN samples of physical units), it prints, for each such signal, the warning “WFDB-x: N sample(s) of signal K will be read back as invalid (missing) samples (format F stores an invalid sample as LO)”. `WriteOptions::invalid_warning = false` turns the warning off.

The duplicate-key warning. When a header holds two metadata entries whose keys differ only in letter case (written by other software), `read_metadata`, `get_metadata` and `read_header` use the first and print the warning “WFDB-x: header of record REC has more than one entry for key KEY (lines L1, L2); the first is used”. `set_metadata` and `update_metadata` then refuse to change that key (`ArgumentError`), and `remove_metadata` removes all of its entries.

```
// A record whose last sample is changed after it was written: the checksum fails.
std::vector<WFDB_Sample> x(100);
for (int i = 0; i < 100; i++) x[i] = i;
WriteOptions w;
w.gains = {100};
```

```
w.adcres = {12};
w.units = {"mV"};
write_record("ck", x, 1, 100, w);
std::FILE *f = std::fopen("ck.dat", "r+b");
std::fseek(f, -2, SEEK_END);
std::fputc(0, f);
std::fputc(0, f);
std::fclose(f);
Record rec = read_record("ck");      // prints the checksum warning
std::printf("%zu %d\n", rec.nsamples(), rec.checksum_ok);   // 100 0: all the samples
ReadOptions quiet;
quiet.checksum_warning = false;
rec = read_record("ck", quiet);      // no warning

// Two entries for one key: the first is used, with a warning.
WriteOptions d = w;
d.gains = {0};
d.units = {};
d.comments = {" Age: 63", " AGE: 64"};
write_record("dup", std::vector<WFDB_Sample>(10, 0), 1, 100, d);
std::printf("%s\n", get_metadata("dup", "age")->c_str());   // 63, with a warning
try {
    set_metadata("dup", "Age", "65");
} catch (const ArgumentError &e) {
    std::printf("ArgumentError: %s\n", e.what());
    // ArgumentError: header of record dup has more than one entry for key Age
    // (lines 3, 4); remove them first
}
std::printf("%d\n", remove_metadata("dup", "age"));          // 2

try {
    read_record("nosuchrecord");
} catch (const Error &e) {
    std::printf("%s\n", e.what());   // can't read record nosuchrecord
}
try {
    read_metadata("nosuchrecord");
} catch (const FileError &e) {
    std::printf("%s\n", e.what());   // can't read header for record nosuchrecord
}
```

7.4 The WFDB library functions in C++

Every function of the WFDB library is declared in `<wfdbx/wfdb.hpp>`, in namespace `wfdbx`, under its C name, with the same arguments, in the same order, and the same return values, so that programs and documentation written for the C library carry over directly. The functions, their arguments and their results are documented once, for all the languages, in Chapter 13; this section gives the rules by which the C declarations become C++ functions (see also Section 13.1.2). The header summarizes each function, and marks what WFDB-x changes in it.

7.4.1 Conventions

Names and headers. The functions, the types (`WFDB_Sample`, `WFDB_Time`, ...), the structures and the constants (`WFDB_READ`, `WFDB_INVALID_SAMPLE`, `WFDB_HIGHRES`, ...) keep their C names, in namespace `wfdbx`. `<wfdbx/ecgcodes.hpp>` defines the annotation codes (`NORMAL`, `PVC`, ..., Chapter 15), and

<wfdbx/ecgmap.hpp> the macros of <wfdb/ecgmap.h> as inline functions (isann, isqrs, map1, ..., which call wfdb_isann, wfdb_isqrs, wfdb_map1, ...). <wfdbx/record.hpp> includes <wfdbx/wfdb.hpp>.

Arguments. String arguments are const char *, and nullptr is accepted where C accepts NULL: sampfreq(nullptr) returns the frequency of the open record, and getinfo(nullptr) continues with the info strings of the last record. Arrays and structures are allocated by the caller and passed by pointer, as in C: isigopen fills an array of WFDB_Siginfo, getvec an array of WFDB_Sample, getann a WFDB_Annotation. getseginfo takes a const WFDB_Seginfo **.

Strings are returned by value, as std::string, or as std::optional<std::string> where the C function may return NULL (annstr, ecgstr, anndesc, getinfo, wfdbfile, ...); unlike C's pointers to static buffers, they stay valid after later calls. Text is handled as bytes.

Return values and errors. The functions return what the C functions return: an int status (negative on errors), a number of signals, a double, a string or std::nullopt. Errors are reported by these values and by a message on the standard error, as in C, never by exceptions. wfdberror() returns the last message; wfdbquiet() turns the messages off and wfdbverbose() on again.

State. As in C, the library keeps its state in the process: the open record, its input signals, the output signals and the annotators remain open between calls, until wfdbquit() closes them. wfdbquit() leaves the input and output directories and the network setting as they are. The library is not thread-safe. The high-level functions of Section 7.3 (except the metadata functions) call wfdbquit(), which closes anything opened with the library functions.

Additions. getvecs(buffer, n) reads up to n sample vectors at once, with exactly the result of n calls of getvec, but decoding whole blocks of the signal files; getvecs(buffer, n, &status) also gives the status of the last getvec call.

```
long long getvecs(WFDB_Sample *out, long long n, int *status = nullptr);
```

7.4.2 The structures

The structures of the C library have the names and the fields of the C structures, with C's zero values as defaults; they are also available under the shorter names Siginfo, Anninfo, Annotation, Calinfo and Seginfo (Table 7.14).

Table 7.14 The C structures in C++

Structure	Fields
WFDB_Siginfo	fname, desc, units (std::optional<std::string>), gain, initval, group, fmt, spf, bsize, adcres, adczero, baseline, nsamp, cksum (Table 7.12)
WFDB_Seginfo	recname (the name of the segment, a char array), nsamp (its number of samples), samp0 (the number of its first sample in the record)
WFDB_Anninfo	name (the annotator name, a std::string), stat (WFDB_READ or WFDB_WRITE for the MIT format, WFDB_AHA_READ or WFDB_AHA_WRITE for the AHA format)
WFDB_Annotation	time (WFDB_Time, in sample intervals), anntyp (char), subtyp (signed char), chan (unsigned char), num (signed char), aux (std::optional<std::string>: absent, or a string whose first byte is the length of the text that follows, as in C)
WFDB_Calinfo	low, high (the levels of the calibration pulse, in physical units), scale (physical units per cm), sigtype and units (std::optional<std::string>), caltype

WFDB_Time is long long (64 bits), WFDB_Sample is int, WFDB_Date is long, WFDB_Frequency and WFDB_Gain are double, and WFDB_Group, WFDB_Signal and WFDB_Annotator are unsigned int.

7.4.3 The settings functions

The functions of Table 7.15 are additions of WFDB-x to the library, declared in <wfdbx/wfdb.hpp>: they set where files are read and written, network access and legacy mode, for the high-level functions, the library functions and the tools alike (Chapters 3 and 4). Each `set` function returns the previous setting, so that it can be restored. The C function `setwfdb(p)` sets the input directory too (`p` must name at most one directory), and `getwfdb()` returns it (`"."` if none is set); in legacy mode they set and return the WFDB path, as in C.

Table 7.15 The settings functions

Function	Effect and result
`std::optional<std::string> wfdbx_set_input_dir(const char *dir)`	sets the input directory: a local directory, a ~ path or a URL; `nullptr` or `""` restores the default (`WFDBX_INPUT_DIR`, or else the current directory). Returns the previous input directory, or `std::nullopt` if none was set
`std::optional<std::string> wfdbx_get_input_dir()`	returns the input directory, or `std::nullopt` if none is set
`std::optional<std::string> wfdbx_set_output_dir(const char *dir)`	sets the output directory: a local directory or a ~ path (a URL is refused, with a message); `nullptr` or `""` restores the default (`WFDBX_OUTPUT_DIR`, or else the current directory). Returns the previous output directory, or `std::nullopt`
`std::optional<std::string> wfdbx_get_output_dir()`	returns the output directory, or `std::nullopt` if none is set
`int wfdbx_set_network(int on)`	turns network access on (1) or off (0); a negative number restores the default (on if `WFDBX_NETWORK` is a nonzero number). Returns the previous setting, 1 or 0
`int wfdbx_get_network()`	returns 1 if network access is on, 0 otherwise
`int wfdbx_set_legacy(int on)`	turns legacy mode on (1) or off (0), for the whole library. Returns the previous setting
`int wfdbx_get_legacy()`	returns 1 if legacy mode is on, 0 otherwise
`int wfdbx_legacy_args(int &argc, char **&argv)`	if `argv[1]` is `"--legacy"`, removes it from the arguments, turns legacy mode on and returns 1; otherwise returns 0. It also installs the exit check of the tools (an exit status of 0 becomes 1 after a write error, outside legacy mode)
`int wfdbx_time_needs_freq(const char *time_string)`	returns 1 if `strtim(time_string)` needs a sampling frequency, 0 otherwise

Function	Effect and result
`int wfdbx_write_errors()`	returns the number of write errors since the program started (output files that could not be written, flushed or closed)
`void wfdbx_note_write_error()`	counts a write error, for a program that writes a file itself
`std::optional<std::string> wfdbx_input_path(const char *name)`	returns the path under which a program opens a file that it reads itself, where `wfdb_open` finds a file of that name: in normal mode, a relative name under the input directory, a ~ name expanded, an absolute name as given; in legacy mode, the name as given. `""` and `"-"` are returned as given; `std::nullopt` for `nullptr` or a URL (also once the input directory is prefixed)
`std::optional<std::string> wfdbx_output_path(const char *name)`	returns the path under which a program creates a file that it writes itself, where `wfdb_open` creates a file of that name: in normal mode, a relative name under the output directory, a ~ name expanded, an absolute name as given; in legacy mode, any name under the output directory, as a plain prefix. The missing directories are created when the output directory is prefixed. `""` and `"-"` are returned as given; `std::nullopt` for `nullptr` or a URL

7.4.4 Examples

Reading the samples and the annotations of a record, as a C program does (the record `100s` is read from the current directory):

```
#include <cstdio>
#include <vector>
#include <wfdbx/ecgcodes.hpp>
#include <wfdbx/wfdb.hpp>
using namespace wfdbx;

int main() {
    WFDB_Siginfo si[2];                         // receives one WFDB_Siginfo per signal
    int nsig = isigopen("100s", si, 2);         // open 2 signals
    std::printf("%d %s %d %g\n", nsig, si[0].desc->c_str(), si[0].fmt, si[0].gain);
    // 2 MLII 212 200
    std::printf("%g\n", sampfreq(nullptr));   // 360: the frequency of the open record

    WFDB_Sample v[2];                           // receives one sample per signal
    isigsettime(strtim("0:30"));                // skip to 30 seconds
    getvec(v);
```

```
    std::printf("%d %d %g\n", v[0], v[1], aduphys(0, v[0]));    // 947 968 -0.385

    std::vector<WFDB_Sample> frames(360 * 2); // the next second, in one call
    int status;
    long long n = getvecs(frames.data(), 360, &status);
    std::printf("%lld %d\n", n, status);      // 360 2

    WFDB_Anninfo ai{"atr", WFDB_READ};
    WFDB_Annotation a;
    annopen("100s", &ai, 1);
    int count = 0;
    while (getann(0, &a) == 0) {              // 0: an annotation was read; -1: the end
        if (count < 2)
            std::printf("%s %s\n", mstimstr(a.time).c_str(), annstr(a.anntyp)->c_str());
        count++;
    }
    std::printf("%d annotations\n", count);   // 75 annotations

    wfdbquiet();                              // no messages on the standard error
    int stat = isigopen("nosuchrecord", nullptr, 0);
    std::printf("%d %s", stat, wfdberror().c_str());
    // -1 init: can't open header for record nosuchrecord
    wfdbverbose();
    wfdbquit();                               // close everything
}
```

Writing a record and an annotation file, sample by sample and annotation by annotation:

```
#include <cstdio>
#include <wfdbx/ecgcodes.hpp>
#include <wfdbx/wfdb.hpp>
using namespace wfdbx;

int main() {
    WFDB_Siginfo out;
    out.fname = "ramp.dat";
    out.desc = "ramp";
    out.units = "mV";
    out.gain = 200;
    out.fmt = 16;
    out.adcres = 12;
    std::printf("%d\n", osigfopen(&out, 1));    // 1: one output signal
    setsampfreq(250);
    for (WFDB_Sample k = 0; k < 500; k++) putvec(&k);
    std::printf("%d\n", newheader("ramp"));     // 0: ramp.hea is written
    WFDB_Anninfo ai{"qrs", WFDB_WRITE};
    annopen("ramp", &ai, 1);
    for (WFDB_Time t : {100, 350}) {
        WFDB_Annotation a;
        a.time = t;
        a.anntyp = NORMAL;
        putann(0, &a);
    }
    wfdbquit();                                // completes ramp.dat and ramp.qrs
}
```

7.5 The command-line tools in C++

The 33 command-line tools (`rdsamp`, `rdann`, `wrsamp`, `xform`, ...) are separate programs, built in `cpp/build/bin/` (or the `bin` directory of the installation), one per file of `cpp/apps`, and run by their names. Each follows its C program line by line. The tools, their options, their output and their exit status are those of the C tools, and are documented in Chapter 14.

```
export PATH=$PWD/cpp/build/bin:$PATH
export WFDBX_INPUT_DIR=tests/data     # run from the root of the repository
rdsamp -r 100s -t 10 -p     # the first 10 seconds of 100s, in physical units
rdann -r 100s -a atr        # its annotations
rdsamp --legacy -r 100s     # legacy mode: --legacy as the first argument
```

There is no launcher: a tool is run by its own name, and it has no `--version` option, like the C tools. When an output file or the standard output cannot be written (a full disk, for example), a tool reports it ("PROGRAM: write error on standard output") and exits with status 1; with `--legacy`, it keeps the original exit status.

Environment variables. The tools, like the rest of the library, use the environment variables of Table 7.16, also for the files that they read or create themselves: the `-i` file of `wrsamp`, `edf2mit`, `wav2mit`, `a2m` and `ad2m`, the input files of `ahaecg2mit` and the `-S` script of `xform` are read from the input directory, and the WAV file of `mit2wav`, the EDF file of `mit2edf`, the master header of `snip -m` and the `.des`, `.ecg`, `.key` and `.txt` files of `parsescp` are created in the output directory, as the records they read and write are (with `wfdbx_input_path` and `wfdbx_output_path`, Table 7.15). Such a file cannot be read from a URL.

Table 7.16 The environment variables used by the C++ library

Variable	Meaning
`WFDBX_INPUT_DIR`	the input directory (a directory, a ~ path or a URL), when `wfdbx_set_input_dir` does not set one
`WFDBX_OUTPUT_DIR`	the output directory, when `wfdbx_set_output_dir` does not set one
`WFDBX_NETWORK`	network access is on if it is a nonzero number, when `wfdbx_set_network` does not set it
`WFDBANNSORT`	`0` to keep annotation files in the order written (no sorting when they are closed)
`WFDBGVMODE`	the default getvec mode (`setgvmode` with a negative mode)
`WFDBCAL`	the calibration file read by `calopen(nullptr)` (default `wfdbcal`)
`WFDB`	the WFDB path, used in legacy mode only (outside legacy mode, it is ignored, with a message)
`HOME` (`USERPROFILE` on Windows)	the home directory, for ~ paths

Tools that write signal parameters require them as options: `wrsamp` (with `-o`) requires `-F` (frequency), `-G` (gain), `-R` (ADC resolution) and, for calibrated signals, `-U` (units); `wav2mit` requires `-G`, and `-U` unless the gain is 0; `ad2m` and `ahaecg2mit` require `-F`, `-G`, `-R`, and `-U` unless the gain is 0; `a2m`, `rdann`, `ann2rr`, `sortann` and `m2a` require `-F` when the record's header gives no frequency (Chapter 14).

7.6 Testing

The unit tests of the C++ implementation are in `cpp/tests`, built with the library (unless `WFDBX_BUILD_TESTS` is `OFF`) as the program `cpp/build/wfdbx_tests`. From the root of the repository, after building:

```
(cd cpp/build && ctest)                          # the unit tests
cpp/build/wfdbx_tests                            # the same, one line per test
cpp/build/wfdbx_tests metadata_shared_cases      # selected tests, by name
```

The unit tests cover the WFDB library functions, the high-level functions, the metadata functions (with the cases shared by the seven languages, in `tests/metadata`), `check_record` and the FLAC codec. They run in temporary copies of the test records of `tests/data` (the environment variable `WFDBX_TEST_DATA` names another directory). The tests of network access use a local HTTP server on 127.0.0.1 only, and are skipped without `curl`. The tests of `check_record` write the corpus of `python/tests/check_corpus.py` with `python3` (or the program named by the environment variable `WFDBX_PYTHON`) and the Python package of `python/src`, and compare the reports with its `expected.txt`; without Python they are skipped. `cpp/build/wfdbx_check_corpus DIR` prints the reports for a corpus in `DIR`.

The cross-language conformance suite runs every tool and compares its results, byte for byte, with those of the original C programs (1155 cases), and, with `--legacy`, in legacy mode (886 cases). It runs the tools of `cpp/build/bin`:

```
PYTHONPATH=$PWD/python/src python3 tests/conformance/harness.py check --impl cpp -j8
PYTHONPATH=$PWD/python/src python3 tests/conformance/harness.py check --impl cpp -j8 --legacy
```

CHAPTER 8

Rust

The Rust implementation of WFDB-x is the crate `wfdbx`, in the directory `rust` of the repository: a library and the 33 command-line tools, each a separate program of the crate. It is a complete Rust implementation of the WFDB Software Package: it uses only the Rust standard library (no other crate), contains no `unsafe` code, and never calls the C library.

It offers WFDB-x in three ways:

- **The high-level API** (Section 8.3): functions that read and write whole records, headers, annotation files and header metadata as `Vec`s and structures, in one call, with options given as fields of a structure, and that return a `Result` whose error type is `wfdbx::record::Error`. They read signal files in large blocks, and are the fastest way to read a record. They are an addition of WFDB-x: the C package has no equivalent. They are the recommended interface for new code and analysis.
- **The WFDB library functions** under their C names (`isigopen`, `getvec`, `getann`, `putann`, ...), with the behavior of the C library (Section 8.4). They are documented once, for all the languages, in Chapter 13.
- **The command-line tools** (`rdsamp`, `rdann`, `wrsamp`, `xform` and 29 others), with the options, output and exit status of the C tools, one program per tool, run by its name (Section 8.5). They are documented in Chapter 14.

The high-level functions return exactly the samples and annotations that the library functions (and the C library) give.

8.1 Installation and requirements

8.1.1 Requirements

WFDB-x for Rust needs Rust 1.70 or later (the stable toolchain, edition 2021; the crate declares `rust-version = "1.70"`) and Cargo, on Linux, macOS or Windows. It has no dependency beyond the Rust standard library: no WFDB library and no libFLAC (the FLAC formats 508, 516 and 524 are read and written by WFDB-x's own codec). Network access (Section 3.4) runs the program `curl`, which must then be installed. The tests of `check_record` also run `python3`, with NumPy (Section 8.6).

8.1.2 Using the library in a Cargo project

A Cargo project uses the crate as a dependency, from the repository or from a copy of it, in its `Cargo.toml`:

```
[dependencies]
wfdbx = { git = "https://github.com/alphanumericslab/wfdb-x" }
# or, from a copy of the repository:
wfdbx = { path = "../wfdb-x/rust" }
```

The crate is `wfdbx`, version `1.2.3`: its library under the GNU Lesser General Public License, version 2.1 or later, and its tools (the binaries of `src/bin`) under the GNU General Public License, version 3 or later (Section 1.9); its FLAC decoder translates the stream decoder of libFLAC 1.4.3, under the BSD-3-Clause license. Its license expression is therefore `LGPL-2.1-or-later AND GPL-3.0-or-later AND BSD-3-Clause`, and the crate (the directory `rust`) holds the files `LICENSE`, `LICENSE-GPL` and `NOTICE`, copies of those of the repository: `NOTICE` summarizes the licenses and gives the copyright notice and license of libFLAC. Programs that use the crate are built in release mode for speed (`cargo build --release`, `cargo run --release`).

8.1.3 Building and installing the tools

From the root of a copy of the repository, `cargo install` builds the tools in release mode and installs them in `~/.cargo/bin`; `cargo build --release`, in the directory `rust`, builds the library and the tools in `rust/target/release/` without installing them:

```
cargo install --path rust                                  # the 33 tools, in ~/.cargo/bin
cargo build --release --manifest-path rust/Cargo.toml      # in rust/target/release/
cargo build --release --manifest-path rust/Cargo.toml --bin rdsamp   # one tool
```

The tools, like the C tools, have no `--version` option. `cargo doc --open`, in the directory `rust`, builds the documentation of the crate (its doc comments, with examples) and opens it.

8.1.4 The sample record

The repository includes the sample record `100s`, the sample record of the WFDB Software Package: the first minute of record 100 of the MIT-BIH Arrhythmia Database, two ECG signals (`MLII` and `V5`) sampled at 360 Hz, with the reference beat annotations (annotator `atr`). Its files (`100s.hea`, `100s.dat`, `100s.atr`) are in the directory `tests/data` of a copy of the repository, with the other test records of WFDB-x. The crate does not install them: programs name that directory as their input directory (`wfdbx_set_input_dir("tests/data")`, or the option `input_dir`). Work that uses the MIT-BIH Arrhythmia Database cites it as described on PhysioNet.

8.2 Quick start

This program reads the header, ten seconds of the signals and the annotations of the sample record `100s`, from the directory `tests/data` of the repository (the input directory of each call), and computes the mean heart rate. It then writes the first ten seconds as a new record, `first10`, in the current directory, with a metadata entry in its header.

```
use wfdbx::{check_record, isqrs, read_annotations_with, read_header_with, read_metadata,
            read_record, set_metadata, write_record, ReadOptions, WriteOptions};

fn main() -> Result<(), Box<dyn std::error::Error>> {
    // The sample record 100s, in the directory tests/data of the repository.
    let sample = ReadOptions { input_dir: Some("tests/data".into()), ..Default::default() };

    // The header only: no samples are read.
    let hdr = read_header_with("100s", &sample)?;
    println!("{} {} {}", hdr.nsig, hdr.fs, hdr.nsamples); // 2 360 21600

    // Seconds 10 to 20 of both signals, frame by frame, in ADC units.
    let opt = ReadOptions {
        from: Some("0:10".into()),
        to: Some("0:20".into()),
        ..sample.clone()
```

```
    };
    let rec = read_record("100s", &opt)?;
    println!("{} {}", rec.nsamples(), rec.nsig); // 3600 2
    println!("{} {}", rec.at(0, 0), rec.physical(0, 0)?); // 946 -0.39

    // The beat annotations, and the mean heart rate.
    let ann = read_annotations_with("100s", "atr", None, &sample)?;
    let beats: Vec<i64> = (0..ann.len())
        .filter(|&i| isqrs(ann.codes[i]) != 0)
        .map(|i| ann.samples[i])
        .collect();
    let rr = (beats[beats.len() - 1] - beats[0]) as f64 / ann.fs / (beats.len() - 1) as f64;
    println!("{} beats, {:.1} beats per minute", beats.len(), 60.0 / rr);
    // 74 beats, 73.9 beats per minute

    // Write the first 10 seconds as the record "first10" (first10.hea and first10.dat,
    // in the current directory), with the signal specifications of 100s.
    let opt = ReadOptions { to: Some("0:10".into()), ..sample.clone() };
    let x = read_record("100s", &opt)?;
    let w = WriteOptions {
        gains: vec![200.0],
        baselines: vec![1024],
        adczeros: vec![1024],
        adcres: vec![11],
        units: vec![Some("mV".into())],
        names: x.names(),
        metadata: vec![("Source".into(), "MIT-BIH record 100".into())],
        ..Default::default()
    };
    write_record("first10", &x.samples, x.nsig, x.fs, &w)?;

    // Header metadata: "# Key: value" lines of the header (keys and values are bytes).
    set_metadata("first10", "Age", "69")?;
    for (key, value) in read_metadata("first10")? {
        println!("{} = {}", String::from_utf8_lossy(&key), String::from_utf8_lossy(&value));
    }
    // Source = MIT-BIH record 100
    // Age = 69
    println!("{}", check_record("first10", &[])?); // no problems found
    Ok(())
}
```

Run from the root of the repository, as the program `src/main.rs` of a Cargo project that depends on `wfdbx` (`cargo run --release`). The header `100s.hea` gives no units: the program writes `"mV"` explicitly, since WFDB-x never assumes units, a gain, an ADC resolution or a sampling frequency that a record does not give (Chapter 4).

8.3 The high-level API

The high-level API is made of the functions of this section, defined in the modules `wfdbx::record`, `wfdbx::check` and `wfdbx::metadata` and available at the root of the crate (`wfdbx::read_record`, ...), of the structures of their options (Section 8.3.6) and of their results (Section 8.3.7), and of the error type `wfdbx::record::Error` (Section 8.3.8). The doc comments of the crate (`cargo doc`) document each of them.

8.3.1 Conventions

Free functions and methods. The state of the library (the open record, the annotators, the input and output directories, network access, legacy mode) is held by a value of the type `Wfdb`. Each high-level function exists twice: as a method of `Wfdb`, such as `w.read_record("100s", &opt)`, which uses the state of `w`, and as a free function of the crate with the same name and the same parameters, such as `wfdbx::read_record("100s", &opt)`, which uses the default `Wfdb` value of the calling thread (one per thread). The entries of this section document the free functions; the methods take `&mut self` first, and then the same parameters, with the same types and results. Separate `Wfdb` values share no state (except what belongs to the process: the count of write errors, the cache of downloaded remote files and the messages printed once per process), so several records can be read at once, in one thread or in several; a `Wfdb` is `Send`, and is used by one thread at a time.

```rust
use wfdbx::Wfdb;

fn main() -> Result<(), wfdbx::record::Error> {
    let mut w = Wfdb::new(); // a library state of its own
    w.wfdbx_set_input_dir("tests/data");
    let a = w.read_record("100s", &Default::default())?; // on w
    let b = wfdbx::read_record("tests/data/100s", &Default::default())?; // on the default value
    println!("{}", a.samples == b.samples); // true
    Ok(())
}
```

Options. The options of a function are the fields of a structure (`ReadOptions`, `WriteOptions`, `AnnWriteOptions`, `MetadataOptions`, Section 8.3.6), passed by reference as the last argument. Each implements `Default`, so that the fields that are not given keep their defaults: `ReadOptions { highres: true, ..Default::default() }`. Several functions have a short form with no options or with a few arguments (`read_header`, `read_annotations`, `write_annotations`, `check_record`, the metadata functions), and a form whose name ends in `_with`, which takes the options structure. An option that may be absent is an `Option`; `None` means the default or the current setting.

Record names and directories. A record is named as in the other parts of WFDB-x (Chapter 2 and Section 3.2), without the `.hea` extension (the readers, `write_annotations` and `write_record` also accept a trailing `".hea"`, which is removed). A record name is one of:

- a full path, absolute (`"/data/mitdb/100"`) or in the home directory (`"~/Documents/100s"`), used as given;
- a relative path (`"100s"`, `"data/100s"`), read from the input directory if one is set, otherwise from the current directory, and written to the output directory if one is set, otherwise to the current directory (missing directories are created);
- a URL (`"https://physionet.org/files/mitdb/1.0.0/100"`), read only if network access is on, and only with the schemes `http://` and `https://` (Section 3.4); a URL cannot be written.

The input and output directories are set with `wfdbx_set_input_dir` and `wfdbx_set_output_dir` (Section 8.4.3) or with the environment variables `WFDBX_INPUT_DIR` and `WFDBX_OUTPUT_DIR`. The readers take the optional fields `input_dir` and `network` of a `ReadOptions` (or a `MetadataOptions`), and the writers a `dir`: they apply to that one call, and the previous settings are restored when it returns, also when it fails. `input_dir` applies to relative names only, not to full paths or URLs. A writer given a name with a path creates the files in that directory, and the header names the record and its signal file without the path (`rec`, `rec.dat`), so that the record opens under any path that leads to it; in legacy mode, as in the original, the header gives the name with its path.

With `dir`, the name must be a plain name (`"rec"`): the files are created in `dir`, and the record opens from that directory under its plain name. The signal files that a header names are read from the header's directory.

Times. The fields `from` and `to` of `ReadOptions`, and the arguments `from` and `to` of `read_annotations`, are `Option<TimeSpec>` values (Table 8.5): a sample number or a WFDB time string, as `strtim` reads it (`"0:10"`, 10 seconds; `"1:02:03.5"`; `"s3600"`, sample 3600; `"e"`, the end of the record; or a time of day in brackets, `"[14:00:00]"`). A `TimeSpec` converts from integers and strings, so `Some(3600.into())` and `Some("0:10".into())` both work; `None` means that no time is given. A time in seconds needs the sampling frequency; when the record does not give it, such a time is an error.

Samples. Samples are `i32` values (C's `WFDB_Sample`), in ADC units, stored frame by frame in one `Vec<i32>`: `samples[i * nsig + s]` is sample `i` of signal `s`. Invalid samples are `WFDB_INVALID_SAMPLE` (-32768), and NaN in physical units. Sample numbers and times are `i64`.

Per-signal and per-annotation values. The per-signal fields of `WriteOptions` (`gains`, `units`, ...) hold one value for all the signals, or one value per signal; the per-annotation fields of `Annotations` (`symbols`, `chans`, ...) are empty or hold one value per annotation. Any other length is an error, reported before anything is written.

Signal parameters. No sampling frequency, gain, ADC resolution or units is ever assumed. Where one is needed and the record does not give it, it must be given; otherwise the function returns an error. A record whose header gives no sampling frequency has `fs` 0, a signal without an ADC resolution has `adcres` 0, and a signal without units has `units` `None`.

Text. Text read from files (signal descriptions, units, info strings, metadata, aux strings, mnemonics) is given as bytes, `Vec<u8>`, since WFDB files are not UTF-8: every byte sequence round-trips unchanged. Arguments accept `&str`, `&[u8]`, `b"..."` or `Vec<u8>` (`impl AsRef<[u8]>`), and a `Vec<u8>` field is set from a `&str` with `.into()`. `String::from_utf8_lossy` converts bytes for printing. The fields `channels` and `input_dir` of the options and `base_time` of the results are `String`s, and the directories `dir` are `PathBuf`s.

Legacy mode. With `legacy` set to `true`, a function behaves as in the original WFDB package, with its assumed signal parameters and its search for files (Chapter 4); the library's legacy mode is on for the duration of the call only. With `false` (the default), the current setting of `wfdbx_set_legacy` applies, which is off unless it was turned on.

State. The functions that read and write records and annotation files, and `check_record`, call `wfdbquit()` on the `Wfdb` value they use, before and after their work, also when they fail, which closes anything opened with the WFDB library functions on that value. The library's settings (legacy mode, input and output directories, network access, and for these functions also the getvec mode and the time resolution of output annotators, which opening an EDF record sets) are the same after a call as before it (Section 8.3.8). The metadata functions do not call `wfdbquit()`: they open only the header, and leave the library's open files as they are. Results are returned by value.

The examples of this section are the body of the function `fn main() -> Result<(), Box<dyn std::error::Error>>`, followed by `Ok(())`, compiled after these lines, and run with the free functions in a copy of the test records of `tests/data` (among them `100s`), which is the current directory and holds the records that the examples write:

```
use wfdbx::*;
```

```rust
/// Text (bytes) as a String, for printing.
fn s(b: &[u8]) -> String {
    String::from_utf8_lossy(b).into_owned()
}
```

8.3.2 Reading records

read_record Read the signals of a record

```rust
pub fn read_record(record: impl AsRef<[u8]>, opt: &ReadOptions) -> Result<Record, Error>
```

Reads the signals of a record from `opt.from` up to `opt.to` (exclusive), in ADC units, frame by frame, together with the description of the record. The samples are those that `getvec` returns (Chapter 13), read in large blocks with `getvecs`. If the record ends before `opt.to`, the samples up to its end are returned. When the read reaches the end of the record, the samples are compared with the checksums of the header (Section 8.3.8). The getvec mode (`getgvmode()`) and the time resolution of output annotators (`getafreq()`) are the same after the call as before it.

record
: Record name, without the `.hea` extension (a trailing `".hea"` is also accepted): a full path, absolute (`"/data/mitdb/100"`) or in the home directory (`"~/Documents/100s"`), used as given; a relative path (`"100s"`, `"data/100s"`), read from the input directory if one is set, otherwise from the current directory; or a URL (`"https://physionet.org/files/mitdb/1.0.0/100"`), read only if network access is on.

opt
: The options (Table 8.1): the signals to read (`channels`), the interval (`from`, `to`), `highres`, `pad`, `checksum_warning`, `legacy`, and the input directory (under which relative record names are read; it does not apply to full paths or URLs) and network access of the call (`input_dir`, `network`). `&ReadOptions::default()`: every signal of the whole record, with the current settings.

Returns A `Record` (Table 8.6), with the samples in ADC units.

Errors An `Error` if the record cannot be read or has no signals ("can't read record REC"), or its signals cannot be opened ("can't open signals of record REC"); if a requested signal does not exist ("record REC has no signal C"); if `opt.from` is a negative sample number ("read_record: from must not be negative") or cannot be reached ("can't seek to sample N of record REC"); if a time string needs the sampling frequency and the header gives none ("sampling frequency of record REC is not specified"); or if a signal file is shorter than its header says, or cannot be read or decoded ("can't read signal data of record REC (getvec status S)"), also when the checksums do not match either (`getvec` then reports the checksum error, -4, for the incomplete last frame, and S is -3): an incomplete frame is never returned. Data that do not match the checksums of the header are returned, with a warning (see `checksum_warning`). The samples are stored as they are read: at most 2^{26} values are allocated in advance, whatever length the header (or `opt.to`) gives.

```rust
// Seconds 10 to 20 of lead MLII, in physical units.
let opt = ReadOptions {
    channels: vec!["MLII".into()],
    from: Some("0:10".into()),
    to: Some("0:20".into()),
    ..Default::default()
};
```

```rust
let rec = read_record("100s", &opt)?;
let mlii: Vec<f64> = rec.physical_signal(0)?;
println!("{} {} {} {}", mlii.len(), rec.fs, rec.sampfrom, s(&rec.names()[0]));
// 3600 360 3600 MLII

// The first 3 samples of both signals, in ADC units, V5 first.
let first = ReadOptions {
    channels: vec!["1".into(), "0".into()],
    to: Some(3.into()),
    ..Default::default()
};
let rec = read_record("100s", &first)?;
for i in 0..rec.nsamples() {
    println!("{} {}", rec.at(i, 0), rec.at(i, 1)); // 1011 995, 3 times
}
println!("{} {}", rec.siginfo[0].gain, rec.siginfo[0].baseline); // 200 1024

// The whole record: the checksums are verified.
let rec = read_record("100s", &ReadOptions::default())?;
println!("{} {}", rec.nsamples(), rec.checksum_ok); // 21600 true
```

A multi-frequency record (here the test record `mfreq`, whose signals have 1, 2 and 4 samples per frame at a frame frequency of 100 Hz) is read at its frame frequency, or at its highest sampling frequency:

```rust
let rec = read_record("mfreq", &ReadOptions::default())?;
println!("{} {}", rec.fs, rec.nsamples()); // 100 600
let opt = ReadOptions { highres: true, ..Default::default() };
let rec = read_record("mfreq", &opt)?;
println!("{} {}", rec.fs, rec.nsamples()); // 400 2400
let spf: Vec<i32> = rec.siginfo.iter().map(|si| si.spf).collect();
println!("{spf:?}"); // [1, 2, 4]
```

`read_header` — Read the header of a record

```rust
pub fn read_header(record: impl AsRef<[u8]>, legacy: bool) -> Result<Header, Error>
```

Reads the header of a record, without opening its signal files. The result describes the record and each of its signals, and holds the info strings and the metadata entries of the header (from the `.hea` file only, as `read_metadata` reads them, with its warning for a key given more than once). For an EDF record, which has no `.hea` file, and for a header read from the standard input (`"-"`), the metadata is empty, without an error. For a multi-segment record, the master header is read, and `nsamples` is the length of the whole record.

`record`
: Record name, as for `read_record`: a full path (absolute or ~), used as given; a relative path, read from the input directory if one is set, otherwise from the current directory; or a URL, read only if network access is on.

`legacy`
: `true` to read the header as the original WFDB package does; normally `false`.

Returns A `Header` (Table 8.7).

Errors An `Error` if the header cannot be read (“can’t read header for record REC”).

```rust
let hdr = read_header("100s", false)?;
println!("{} {} {} {}", s(&hdr.name), hdr.fs, hdr.nsig, hdr.nsamples); // 100s 360 2 21600
for si in &hdr.siginfo {
    let desc = si.desc.as_deref().unwrap_or(b"");
    let fname = si.fname.as_deref().unwrap_or(b"");
    println!("{} {} {} {} {} {:?}", s(desc), s(fname), si.fmt, si.gain, si.adcres, si.units);
}
// MLII 100s.dat 212 200 11 None
// V5 100s.dat 212 200 11 None
println!("[{}]", s(&hdr.comments[0])); // [ 69 M 1085 1629 x1]
println!("{}", hdr.base_time.is_none()); // true: the header gives no start time
let (key, value) = &hdr.metadata[0];
println!("[{}] [{}]", s(key), s(value));
// [Produced by xform from record 100, beginning at 0] [0]
```

read_header_with

Read the header of a record, with options

```rust
pub fn read_header_with(record: impl AsRef<[u8]>, opt: &ReadOptions) -> Result<Header, Error>
```

As `read_header`, with the options `input_dir`, `network` and `legacy` of `opt`, for this call only. The other fields of `opt` are not used.

record
: Record name, as for `read_header`.

opt
: The options (Table 8.1); only `input_dir` (optional: the input directory under which relative record names are read; it does not apply to full paths or URLs), `network` and `legacy` are used.

Returns A `Header` (Table 8.7).

Errors As `read_header`.

```rust
// A copy of the header 100s.hea in the directory "in".
std::fs::create_dir_all("in")?;
std::fs::copy("100s.hea", "in/100s.hea")?;
let opt = ReadOptions { input_dir: Some("in".into()), ..Default::default() };
let hdr = read_header_with("100s", &opt)?; // reads in/100s.hea
println!("{} {}", hdr.nsig, hdr.fs); // 2 360
println!("{}", read_header("100s", false).is_ok()); // true: the input directory is restored
```

read_annotations

Read an annotation file

```rust
pub fn read_annotations(
    record: impl AsRef<[u8]>,
    annotator: impl AsRef<[u8]>,
    from: Option<TimeSpec>,
    to: Option<TimeSpec>,
    fs: Option<f64>,
    legacy: bool,
) -> Result<Annotations, Error>
```

Reads the annotations of an annotation file, in file order, as parallel vectors. The sampling frequency, needed to convert times in seconds, is that of the record's header, or, for a record without a header or whose header gives none, the argument `fs`, as the option `-F` of `rdann`.

`record`
: Record name, as for `read_record`: a full path (absolute or ~), used as given; a relative path, read from the input directory if one is set, otherwise from the current directory; or a URL, read only if network access is on. Its header, if there is one, gives the sampling frequency.

`annotator`
: Annotator name: the suffix of the annotation file. `"atr"` reads the file `RECORD.atr`.

`from`
: The first time to read: a sample number or a time string. `None`: the beginning of the file.

`to` The time at which to stop: annotations at or after it are not read. `None`: the end of the file.

`fs` The sampling frequency, in Hz (> 0), to use when the record has no header or its header gives none; it is ignored otherwise. `None`: none.

`legacy`
: `true` to behave as the original WFDB package (which assumes 250 Hz for a record without a frequency); normally `false`.

Returns An `Annotations` (Table 8.8). Its `fs` is the frequency in which the times are counted: the one written in the annotation file, if there is one, or else the sampling frequency; it is 0 if none of them is known.

Errors An `Error` if `from` is a negative sample number ("read_annotations: from must not be negative", checked first, before any file is opened); if `fs` is given and is not > 0 ("fs must be > 0"); if the annotation file cannot be opened ("can't read annotator ANN for record REC"); if a time string needs the sampling frequency and none is known ("sampling frequency of record REC is not specified"); if `from` cannot be reached ("can't seek in annotator ANN"); or if the file ends early or cannot be read before `to` is reached ("can't read annotation file ANN of record REC (getann status S)", where S is the status that `getann` returned, such as -3 for an unexpected end of file). Whatever happens, the annotation file is closed before the function returns.

```
let ann = read_annotations("100s", "atr", Some("0:10".into()), Some("0:30".into()), None, false)?;
println!("{} {}", ann.len(), ann.fs); // 24 360
println!("{} {}", ann.samples[0], s(&ann.symbols[0])); // 3862 N

// The whole file: the first annotation is a rhythm change, with an aux string.
let ann = read_annotations("100s", "atr", None, None, None, false)?;
let aux = ann.aux[0].as_deref().unwrap_or(b"");
println!("{} {} {}", s(&ann.symbols[0]), ann.codes[0] == RHYTHM, s(aux)); // + true (N
```

For a record without a header, `fs` gives the frequency in which times in seconds are converted (the library reports on the standard error that it cannot open the header):

```
let beats = Annotations {
    samples: vec![100, 460, 820],
    symbols: vec!["N".into(), "N".into(), "N".into()],
    ..Default::default()
};
write_annotations("beats", "qrs", &beats, 0.0, None)?; // no header
let ann = read_annotations("beats", "qrs", None, Some("0:02".into()), Some(360.0), false)?;
println!("{} {}", ann.len(), ann.fs); // 2 360
```

read_annotations_with Read an annotation file, with options

```
pub fn read_annotations_with(
    record: impl AsRef<[u8]>,
    annotator: impl AsRef<[u8]>,
    fs: Option<f64>,
    opt: &ReadOptions,
) -> Result<Annotations, Error>
```

As `read_annotations`, with the options `from`, `to`, `input_dir`, `network` and `legacy` of `opt` (the last three for this call only). The other fields of `opt` are not used.

record
: Record name, as for `read_annotations`.

annotator
: Annotator name, as for `read_annotations`.

fs
: The sampling frequency to use when the record has no header or its header gives none, as for `read_annotations`.

opt
: The options (Table 8.1); only `from`, `to`, `input_dir` (optional: the input directory under which relative record names are read; it does not apply to full paths or URLs), `network` and `legacy` are used.

Returns An `Annotations` (Table 8.8).

Errors As `read_annotations`.

```
// Seconds 10 to 30 of the annotations of 100s, read from the directory "in".
std::fs::create_dir_all("in")?;
for f in ["100s.hea", "100s.atr"] {
    std::fs::copy(f, format!("in/{f}"))?;
}
let opt = ReadOptions {
    from: Some("0:10".into()),
    to: Some("0:30".into()),
    input_dir: Some("in".into()),
    ..Default::default()
};
let ann = read_annotations_with("100s", "atr", None, &opt)?;
println!("{} {}", ann.len(), ann.samples[0]); // 24 3862
```

8.3.3 Writing records

write_record Write a header and a signal file

```
pub fn write_record(
    record: impl AsRef<[u8]>,
    samples: &[i32],
    nsig: usize,
    fs: f64,
    opt: &WriteOptions,
) -> Result<(), Error>
```

Writes samples in ADC units as a new record: the header `RECORD.hea` and one signal file, `RECORD.dat`, which holds all the signals (so they share one format). Existing files of these names are replaced. The sampling frequency, the gain and ADC resolution of each signal, and the units of each calibrated signal must be given: there are no defaults. Every check except those of the writing itself comes before any file is created or changed. Both files are written to temporary files in the directories of their destinations (`.NAME.XXXXXXXX.tmp`), which replace the existing files (keeping their permissions; a symbolic link is followed, and a destination that is not a regular file, such as a device, is written directly) only when both are complete, so that an error leaves the existing files as they were. Whatever happens, the output signals are closed and no temporary file is left before the function returns, and the getvec mode and the time resolution of output annotators are left as they were. (Text is bytes and ADC values are `i32`: the checks of the other languages for header text that is not Latin-1, and for ADC values that are not whole numbers within the range of a 32-bit integer, have no counterpart.)

`record`
: Record name (not empty), without the `.hea` extension (a trailing `".hea"` is removed, as by the readers): a plain name (`"rec"`) or a relative path (`"data/rec"`), created in the output directory if one is set, otherwise in the current directory, missing directories included; or a full path, absolute or in the home directory (`"~/Documents/rec"`), used as given. With a path, the files are created in that directory, and the header names the record and its signal file without the path (`rec`, `rec.dat`), so that the record opens under any path that leads to it (in legacy mode, as in the original, the header gives the name with its path). A URL cannot be written.

`samples`
: The samples, in ADC units, frame by frame: `nsig` values per frame, or, for a multi-frequency record, the sum of `opt.samps_per_frame` (the samples of signal 0, then those of signal 1, and so on). `WFDB_INVALID_SAMPLE` marks invalid samples. The number of samples must be a multiple of the frame size.

`nsig`
: The number of signals (> 0).

`fs`
: The sampling frequency, in Hz: a finite number > 0 (for a multi-frequency record, the frame frequency).

`opt`
: The options (Table 8.2): the format, the signal specifications (`gains`, `adcres` and `units` are required), the start time, the comments and metadata, the directory `dir` (which must exist) in which to create the files for this call only, `record` being then a plain name, `legacy`, and `invalid_warning`.

Returns `Ok(())`.

Errors An `Error` with `opt.dir`, if the directory does not exist ("directory D does not exist") or the record name contains one ("record name must not contain a directory when dir is given"); then, in this order: if the record name is empty ("write_record: the record name must not be empty"); if `nsig` is 0 ("write_record: at least one signal is required"); if a `samps_per_frame` value is less than 1 ("write_record: samps_per_frame must hold integers >= 1"); if a per-signal field of `opt` has more than one value but not one per signal ("write_record: OPTION must have one value per signal"); if `samples.len()` is not a multiple of the frame size ("write_record: the number of samples is not a multiple of the frame size"); if the gains, the ADC resolutions (in normal mode, each at least 1) or the units of a calibrated signal are missing ("write_record: gains must be specified", "write_record: ADC resolution(s) must be specified", "write_record: units must be specified"); if `fs` is not > 0 or not finite ("write_record: the sampling frequency must be > 0", "…must be finite"); if a gain is not finite ("write_record: gains must be finite"); if a FLAC format

is given for more than 8 signals (“write_record: format 516 (FLAC) stores at most 8 signals in a file; the record has 12”); if a sample does not fit the format (“write_record: signal K has a sample (V) outside the range of format F (LO to HI)”; in format 8, “write_record: signal K changes by D between two samples, more than format 8 can store (-128 to 127)” or “write_record: signal K has an invalid sample, which format 8 cannot store”), the samples of each signal being checked in order (for a multi-frequency signal, its samples of each frame, frame after frame); if an entry of `opt.metadata` is not valid (“write_record: metadata key ’K’ is also given in comments”, for example); if `opt.base_time` is not a valid time (“write_record: incorrect base time ’T’”); then, if the files cannot be created (“can’t create signal file for record REC”, “can’t create header for record REC”); or if they cannot be written completely, for example on a full disk (“write error on record REC”); the existing files are then left as they were.

```
// A ramp of 1000 samples of one signal, in millivolts, at 100 Hz: the record "ramp"
// (ramp.hea and ramp.dat).
let x: Vec<i32> = (0..1000).collect();
let opt = WriteOptions {
    gains: vec![200.0],
    adcres: vec![12],
    units: vec![Some("mV".into())],
    names: vec!["ramp".into()],
    base_time: Some("14:00:03 21/06/2024".into()),
    comments: vec!["a ramp".into()],
    metadata: vec![("Generator".into(), "loop".into())],
    ..Default::default()
};
write_record("ramp", &x, 1, 100.0, &opt)?;
let rec = read_record("ramp", &ReadOptions::default())?;
println!("{} {} {}", rec.nsamples(), rec.base_time.as_deref().unwrap_or(""), rec.physical(10, 0)?)
    ↪ ;
// 1000 [14:00:03 21/06/2024] 0.05
for c in &read_header("ramp", false)?.comments {
    println!("[{}]", s(c));
}
// [a ramp]
// [ Generator: loop]
```

A multi-frequency record: signal `fast` has 2 samples per frame, signal `slow` one; each frame holds the two samples of `fast` and then the sample of `slow`:

```
let mut frames = Vec::new();
for i in 0..100 {
    // 1 second at 100 frames per second
    frames.push(2 * i); // fast: 200 samples per second
    frames.push(2 * i + 1);
    frames.push(500); // slow: 100 samples per second
}
let opt = WriteOptions {
    gains: vec![200.0],
    adcres: vec![12],
    units: vec![Some("mV".into())],
    names: vec!["fast".into(), "slow".into()],
    samps_per_frame: vec![2, 1],
    ..Default::default()
};
write_record("mf", &frames, 2, 100.0, &opt)?;
let hi = ReadOptions { highres: true, ..Default::default() };
```

```rust
let rec = read_record("mf", &hi)?;
println!("{} {} {} {} {}", rec.fs, rec.at(0, 0), rec.at(0, 1), rec.at(1, 0), rec.at(1, 1));
// 200 0 500 1 500
```

---

write_record_physical

Write a record from samples in physical units

```rust
pub fn write_record_physical(
    record: impl AsRef<[u8]>,
    samples: &[f64],
    nsig: usize,
    fs: f64,
    opt: &WriteOptions,
) -> Result<(), Error>
```

As `write_record`, with the samples in physical units. Each sample v is stored as $\mathrm{round}(v \cdot gain) + baseline$ (halves rounded away from zero), and NaN is stored as an invalid sample. Every signal must have a gain other than 0.

record
: Record name, as for `write_record`.

samples
: The samples, in physical units, frame by frame (as for `write_record`).

nsig
: The number of signals (> 0).

fs
: The sampling frequency, in Hz: a finite number > 0.

opt
: The options, as for `write_record` (Table 8.2).

Returns `Ok(())`.

Errors As `write_record`; also an `Error` if a gain is 0 ("write_record: signal N is uncalibrated", checked after the units), or if a sample is infinite or its ADC value is outside the range of `i32` ("write_record: sample I is infinite or beyond the range of 32-bit ADC units", checked after the gains, before the metadata). Nothing is written then.

```rust
// Two sine waves, in millivolts, at 360 Hz, frame by frame: the record "sine".
let fs = 360.0;
let mut x = Vec::new();
for i in 0..3600 {
    let t = 2.0 * std::f64::consts::PI * i as f64 / fs;
    x.push(t.sin()); // signal 0
    x.push(0.5 * t.cos()); // signal 1
}
let opt = WriteOptions {
    gains: vec![1000.0],
    adcres: vec![16],
    units: vec![Some("mV".into())],
    names: vec!["A".into(), "B".into()],
    ..Default::default()
};
write_record_physical("sine", &x, 2, fs, &opt)?;
let rec = read_record("sine", &ReadOptions::default())?;
let mut err: f64 = 0.0;
for (i, v) in x.iter().enumerate() {
```

```
    err = err.max((rec.physical(i / 2, i % 2)? - v).abs());
}
println!("{} {}", rec.nsamples(), err <= 0.0005); // 3600 true: rounded to 1/1000 mV
```

---

write_annotations

Write an annotation file

```
pub fn write_annotations(
    record: impl AsRef<[u8]>,
    annotator: impl AsRef<[u8]>,
    ann: &Annotations,
    fs: f64,
    dir: Option<&Path>,
) -> Result<(), Error>
```

Writes an annotation file, `RECORD.ANNOTATOR`, in the MIT format; an existing file of that name is replaced. The record need not have a header. The type of each annotation is its `symbols` entry (`b"N"`, `b"V"`, ...; an unknown mnemonic gives `NOTQRS`), or, if `symbols` is empty or the entry is empty, its `codes` entry (Chapter 15). Both may be given (as `read_annotations` returns them) if they agree: the code of each non-empty symbol (`strann`) is the code given for that annotation. `subtypes`, `chans` and `nums` are stored in one byte each, as by `putann` (other values wrap around: the subtype and number to -128 to 127, the channel to 0 to 255), and `aux` texts are cut to 255 bytes. If the annotations are not in time order, the file is sorted when it is written, unless the environment variable `WFDBANNSORT` is `0` (`write_annotations_with` decides this for one call). The times -2^{63} and $2^{63} - 1$ (`i64::MIN`, `i64::MAX`) are reserved by the file format, and the codes are 0 to 58 (59 to 63 are its control codes). The arguments are checked before the file is created or changed. The file is written to a temporary file in the directory of its destination (`.NAME.XXXXXXXX.tmp`), which replaces an existing file (keeping its permissions; a symbolic link is followed) only when it is complete (and, if needed, sorted), so that an error leaves the existing file as it was. Whatever happens, the annotator is closed and no temporary file is left before the function returns, and the getvec mode and the time resolution of output annotators (`getafreq()`) are left as they were (`fs` applies to this file only).

record
: Record name, without the `.hea` extension (a trailing `".hea"` is also accepted): a plain name or a relative path (`"data/100s"`), created in the output directory if one is set, otherwise in the current directory; or a full path, absolute or in the home directory (`"~/Documents/100s"`), used as given. The file is `RECORD.ANNOTATOR` at that place (`data/100s.qrs`). A URL cannot be written.

annotator
: Annotator name: the suffix of the file. `"qrs"` writes `RECORD.qrs`.

ann
: The annotations (Table 8.8): `samples`, and `symbols` or `codes` (required when there is at least one annotation; with no annotations, an empty file is written); `subtypes`, `chans`, `nums` and `aux` are optional. Each of these vectors is empty (`subtypes`, `chans` and `nums` are then 0, and there is no aux) or has one entry per annotation. `ann.record`, `ann.annotator` and `ann.fs` are not used.

fs
: 0 (none) or a finite frequency > 0, in which the times are counted, recorded in the file; it applies to this file only.

dir
: The directory (which must exist) in which to create the file, for this call only. `record` must then be a plain name, without a directory: the file is `DIR/RECORD.ANNOTATOR`. `None`: the output directory.

Returns `Ok(())`.

Errors An `Error` with `dir`, if the directory does not exist ("directory D does not exist") or `record` contains a directory ("record name must not contain a directory when dir is given"); then, in this order: if a per-annotation field has another length than 0 or the number of annotations ("write_annotations: symbols must have one entry per annotation (N), or none", for example); if there is at least one annotation and neither `symbols` nor `codes` is given ("write_annotations: symbols or codes must be given"); if `fs` is not 0 and not a finite frequency > 0 ("write_annotations: fs must be finite and > 0 (or 0: none)"); if a time is `i64::MIN` or `i64::MAX` ("write_annotations: annotation time T cannot be written (annotation times must be integers from -9223372036854775807 to 9223372036854775806)", for the first such time); if a code is not from 0 to 58 ("write_annotations: annotation code C is reserved (59 to 63 are the control codes of the annotation file format)", or "…annotation code C is out of range (0 to 58)", for the first such code); if `symbols` and `codes` are both given and disagree ("write_annotations: symbols and codes disagree at annotation K", K counted from 0); then, if the file cannot be created ("can't create annotator ANN for record REC"); or if it cannot be written completely, or `putann` reports an error while writing it ("write error on annotator ANN for record REC"); the existing file is then left as it was.

```
// Three beats and a rhythm change, by symbol, with an aux string for the rhythm.
let a = Annotations {
    samples: vec![18, 77, 370, 662],
    symbols: vec!["+".into(), "N".into(), "V".into(), "N".into()],
    aux: vec![Some("(N".into()), None, None, None],
    ..Default::default()
};
write_annotations("100s", "test", &a, 0.0, None)?;
let b = read_annotations("100s", "test", None, None, None, false)?;
let aux = b.aux[0].as_deref().unwrap_or(b"");
println!("{} {} {}", s(&b.symbols[0]), s(&b.symbols[2]), s(aux)); // + V (N

// A copy of the reference annotations, as read_annotations returns them.
let atr = read_annotations("100s", "atr", None, None, None, false)?;
write_annotations("100s", "copy", &atr, 0.0, None)?;
let c = read_annotations("100s", "copy", None, None, None, false)?;
println!("{}", c.samples == atr.samples && c.aux == atr.aux); // true
```

`write_annotations_with` — Write an annotation file, with options

```
pub fn write_annotations_with(
    record: impl AsRef<[u8]>,
    annotator: impl AsRef<[u8]>,
    ann: &Annotations,
    fs: f64,
    opt: &AnnWriteOptions,
) -> Result<(), Error>
```

As `write_annotations`, with the output directory and the sorting of the file given by `opt`, for this call only.

`record`
: Record name, as for `write_annotations`.

`annotator`
: Annotator name, as for `write_annotations`.

ann The annotations, as for `write_annotations`.

fs 0 or the frequency recorded in the file, as for `write_annotations`.

opt The options (Table 8.3): `dir`, as the argument `dir` of `write_annotations`, and `sort`: `Some(true)` always sorts annotations that are not in time order, `Some(false)` keeps the given order (without the library's advice to run `sortann`), and `None` lets `WFDBANNSORT` decide (sorting is on unless it is `0`). The environment is not changed.

Returns `Ok(())`.

Errors As `write_annotations`.

```
let u = Annotations {
    samples: vec![500, 100],
    codes: vec![NORMAL, NORMAL],
    ..Default::default()
};
// Sorted when the file is written (the library reports it on the standard error).
write_annotations_with("100s", "sorted", &u, 0.0, &AnnWriteOptions::default())?;
let a = read_annotations("100s", "sorted", None, None, None, false)?;
println!("{:?}", a.samples); // [100, 500]
// Kept in the order given, in the existing directory "out".
std::fs::create_dir_all("out")?;
let opt = AnnWriteOptions { dir: Some("out".into()), sort: Some(false) };
write_annotations_with("100s", "unsorted", &u, 0.0, &opt)?;
println!("{}", std::path::Path::new("out/100s.unsorted").exists()); // true
```

8.3.4 Checking records

check_record — Check a record and its annotation files

```
pub fn check_record(record: impl AsRef<[u8]>, annotators: &[&str]) -> Result<CheckReport, Error>
```

Checks the header of a record, its signal specifications, every sample of its signal files, each segment of a multi-segment record, and the annotation files named, and reports each problem found. A problem in the record is never an error of the function: it is reported, with a severity (`Severity::Error`: the record is damaged or does not follow the WFDB format, and programs will read wrong data or fail; `Severity::Warning`: the record can be read, but something is missing or unusual, and may give wrong results; `Severity::Info`: a fact worth knowing, such as invalid samples). The problems, their codes and their messages are the same in the seven languages (Table 6.7). The checks run in normal mode, even if legacy mode is on, and the library prints no messages during the call (the previous settings are restored afterwards, the getvec mode and the time resolution of output annotators among them). They are similar to those of Benjamin Moody's `wfdbcheck`.

record
Record name, as for `read_record`: a full path (absolute or ~), used as given; a relative path, read from the input directory if one is set, otherwise from the current directory; or a URL, read only if network access is on.

annotators
The annotator names whose files are checked, such as `&["atr"]`; `&[]`: none.

Returns A `CheckReport` (Table 8.9): `report.ok` is `true` if no problem is an error, `report.issues` lists the problems (`Issue` values, Table 8.10), and `report.lines()` (or `Display`) gives one line per problem, or `no problems found`.

Errors An `Error` if `record` is empty ("record must not be empty"). A record that cannot be read is reported (code `header_unreadable`), not returned as an error.

```
let report = check_record("100s", &["atr"])?;
println!("{}", report.ok); // true: no errors
println!("{report}");
// warning: signal 0: the units are not specified
// warning: signal 1: the units are not specified

// A damaged record: one sample of a 12-bit signal is overwritten with 2048.
let x: Vec<i32> = (0..100).collect();
let opt = WriteOptions {
    gains: vec![100.0],
    adcres: vec![12],
    units: vec![Some("mV".into())],
    names: vec!["ramp".into()],
    ..Default::default()
};
write_record("damaged", &x, 1, 100.0, &opt)?;
let mut dat = std::fs::read("damaged.dat")?;
dat[20] = 0x00; // sample 10 (2 bytes per sample in format 16):
dat[21] = 0x08; // 0x0800 = 2048, little-endian
std::fs::write("damaged.dat", &dat)?;
let report = check_record("damaged", &[])?;
println!("{}", report.ok); // false
for i in &report.issues {
    println!("{} {} {}", i.severity, i.code, i.r#where);
}
// error sample_range signal 0
// error checksum signal 0
println!("{}", report.lines()[0]);
// error: signal 0: 1 sample is outside the ADC range (-2048 to 2047): sample 10 (value 2048)
```

`check_record_with` — Check a record, with options

```
pub fn check_record_with(
    record: impl AsRef<[u8]>,
    annotators: &[&str],
    opt: &ReadOptions,
) -> Result<CheckReport, Error>
```

As `check_record`, with the options `input_dir` and `network` of `opt`, for this call only. The other fields of `opt` are not used.

`record`
: Record name, as for `check_record`.

`annotators`
: The annotator names whose files are checked, as for `check_record`.

`opt`
: The options (Table 8.1); only `input_dir` (optional: the input directory under which relative record names are read; it does not apply to full paths or URLs) and `network` are used.

Returns A `CheckReport` (Table 8.9).

Errors As `check_record`.

```
std::fs::create_dir_all("in")?;
for f in ["100s.hea", "100s.dat", "100s.atr"] {
    std::fs::copy(f, format!("in/{f}"))?;
}
let opt = ReadOptions { input_dir: Some("in".into()), ..Default::default() };
let report = check_record_with("100s", &["atr"], &opt)?;
println!("{} {}", report.ok, report.issues.len()); // true 2
```

8.3.5 Header metadata

These functions read and change the metadata of a header: its info strings of the form `# Key: value`, such as `# Age: 63` (Chapter 5). In short: keys are compared without regard to letter case (A-Z and a-z only); a key must not be empty and must not contain `:`, `#`, CR, LF or NUL, and a value must not contain CR, LF or NUL; spaces, tabs, VT, FF and CR at either end of keys and values are removed; keys and values are bytes (arguments `impl AsRef<[u8]>`, results `Vec<u8>`). A change rewrites the header in one step, through a temporary file in the directory of the destination, which then replaces it, keeping its permissions (on Unix) and every other byte of it; the temporary file is removed on any failure. All the arguments are checked first, so an error leaves the header unchanged, and nothing is written if nothing changes (unless `dir` is given). Only the `.hea` file is read (not a `.info` file); for a multi-segment record, it is the master header, unless a segment is named. The header is opened as the library opens a header (`wfdb_open`), so in legacy mode it is searched for in the WFDB path. The entries are returned as a `Metadata`:

```
pub type Metadata = Vec<(Vec<u8>, Vec<u8>)>;
```

Each function has a form whose name ends in `_with`, which takes a `MetadataOptions` (Table 8.4): the input directory and network access for reading the header, and, for the functions that change it, the directory `dir` in which to write it. These functions do not call `wfdbquit()`. Metadata often holds information about patients: WFDB-x stores it as plain text, exactly as given, and does not de-identify it.

> **Patient data** A header can hold identifying data about a patient (a name, a date of birth, a medical record number). The metadata functions read and write such data as given. Records that are shared must be de-identified first.

`read_metadata` Read the metadata of a record

```
pub fn read_metadata(record: impl AsRef<[u8]>) -> Result<Metadata, Error>
```

Reads the metadata entries of the header of a record, in file order. The info strings are the comment lines after the record line, which is found as the header parser finds it, in every mode: the first line that holds something other than spaces, tabs and CRs and does not begin with `#` (an empty or blank line before it, and the comments before it, are not read). An info string is an entry if it contains a colon: the key is the text before the first colon, the value the rest of the line, both without spaces at either end. Info strings without a colon, or whose key is empty or contains `#`, are free comments, which are not returned.

`record`
: Record name, without the `.hea` extension (a trailing `".hea"` is also accepted): a full path,

absolute (`"/data/mitdb/100"`) or in the home directory (`"~/Documents/100s"`), used as given; a relative path (`"100s"`, `"data/100s"`), read from the input directory if one is set, otherwise from the current directory; or a URL (`"https://physionet.org/files/mitdb/1.0.0/100"`), read only if network access is on.

Returns The entries, as (key, value) pairs, in file order. A key that appears more than once (ignoring letter case) is returned once, with its first value, with a warning (Section 8.3.8).

Errors An `Error` if the header cannot be found or read ("can't read header for record REC"), or if `record` is empty ("record name must not be empty") or `"-"` ("the metadata of a header read from the standard input cannot be read or changed").

```
let opt = WriteOptions {
    gains: vec![200.0],
    adcres: vec![12],
    units: vec![Some("mV".into())],
    names: vec!["ECG".into()],
    comments: vec!["recorded at rest".into()],
    metadata: vec![("Age".into(), "63".into()), ("Sex".into(), "F".into())],
    ..Default::default()
};
write_record("pat", &[0; 360], 1, 360.0, &opt)?;
for (key, value) in read_metadata("pat")? {
    println!("{} = {}", s(&key), s(&value));
}
// Age = 63
// Sex = F
for c in read_header("pat", false)?.comments {
    println!("[{}]", s(&c));
}
// [recorded at rest]: the free comment is not an entry
// [ Age: 63]
// [ Sex: F]
```

`read_metadata_with` — Read the metadata of a record, with options

```
pub fn read_metadata_with(
    record: impl AsRef<[u8]>,
    opt: &MetadataOptions,
) -> Result<Metadata, Error>
```

As `read_metadata`, with the input directory and network access of `opt`, for this call only. `opt.dir` is not used.

`record`
: Record name, as for `read_metadata`.

`opt` The options (Table 8.4); only `input_dir` and `network` are used.

Returns The entries, as for `read_metadata`.

Errors As `read_metadata`.

```
std::fs::create_dir_all("in")?;
std::fs::copy("100s.hea", "in/100s.hea")?;
let opt = MetadataOptions { input_dir: Some("in".into()), ..Default::default() };
```

```rust
let md = read_metadata_with("100s", &opt)?; // reads in/100s.hea
println!("{} {}", md.len(), s(&md[0].1)); // 1 0
```

get_metadata

Return the value of a metadata entry

```rust
pub fn get_metadata(
    record: impl AsRef<[u8]>,
    key: impl AsRef<[u8]>,
) -> Result<Option<Vec<u8>>, Error>
```

Returns the value of one metadata entry of the header of a record.

record
: Record name (as for `read_metadata`).

key The key, such as `"Age"`. Letter case is ignored, and spaces at either end are removed; it must be a valid key (see `set_metadata`), which is checked before the header is read.

Returns `Some(value)` (empty for an entry with an empty value), or `None` if the header has no entry for `key`. If it has more than one, the first is returned, with the warning of `read_metadata`.

Errors As `read_metadata`; also an `Error` if `key` is not valid ("metadata key 'a#b' must not contain '#'", for example).

```rust
let opt = WriteOptions {
    gains: vec![200.0],
    adcres: vec![12],
    units: vec![Some("mV".into())],
    metadata: vec![("Age".into(), "63".into()), ("Notes".into(), "".into())],
    ..Default::default()
};
write_record("pat", &[0; 360], 1, 360.0, &opt)?;
println!("{:?}", get_metadata("pat", "age")?.map(|v| s(&v))); // Some("63")
println!("{:?}", get_metadata("pat", " NOTES ")?.map(|v| s(&v))); // Some(""): an empty value
println!("{:?}", get_metadata("pat", "Weight")?); // None: no such entry
```

get_metadata_with

Return the value of a metadata entry, with options

```rust
pub fn get_metadata_with(
    record: impl AsRef<[u8]>,
    key: impl AsRef<[u8]>,
    opt: &MetadataOptions,
) -> Result<Option<Vec<u8>>, Error>
```

As `get_metadata`, with the input directory and network access of `opt`, for this call only. `opt.dir` is not used.

record
: Record name (as for `read_metadata`).

key The key (as for `get_metadata`).

opt The options (Table 8.4); only `input_dir` and `network` are used.

Returns As `get_metadata`.

Errors As `get_metadata`.

```
let opt = WriteOptions {
    gains: vec![200.0],
    adcres: vec![12],
    units: vec![Some("mV".into())],
    metadata: vec![("Age".into(), "63".into())],
    ..Default::default()
};
write_record("pat", &[0; 360], 1, 360.0, &opt)?;
std::fs::create_dir_all("in")?;
std::fs::copy("pat.hea", "in/pat.hea")?;
let opt = MetadataOptions { input_dir: Some("in".into()), ..Default::default() };
let age = get_metadata_with("pat", "Age", &opt)?.unwrap_or_default();
println!("{}", s(&age)); // 63
```

`set_metadata` — Set the value of a metadata entry

```
pub fn set_metadata(
    record: impl AsRef<[u8]>,
    key: impl AsRef<[u8]>,
    value: impl AsRef<[u8]>,
) -> Result<(), Error>
```

Sets the value of a metadata entry in the header of a record. If the header has an entry for `key` (ignoring letter case), its line is replaced by `# Key: value`, at the same position and with its line ending, with the spelling of `key` given here (even if the value is the same). Otherwise the entry is added after the last info string of the header (or, if it has none, after its last line that is not blank), with the line ending of the record line. Every other byte of the header is kept.

`record`
: Record name (as for `read_metadata`). The header is changed where it was read, unless `dir` is given (`set_metadata_with`); a header read from a URL can only be written elsewhere, with `set_metadata_with` and `dir`.

`key` The key, such as `"Age"` or `"Date of birth"`. Spaces at either end are removed; it must not then be empty, and must not contain `:`, `#`, CR, LF or NUL.

`value`
: The value, which may be a sentence. Spaces at either end are removed; it must not contain CR, LF or NUL, and may be empty (it is then written `# Key:`).

Returns `Ok(())`.

Errors An `Error` if `key` or `value` is not valid (“metadata key must not be empty”, “metadata key ’KEY’ must not contain C”, “value of metadata key ’KEY’ must not contain C”, where C is `':'`, `'#'`, “a line break” or “a NUL character”); if `record` is empty or `"-"`; if the header cannot be read (“can’t read header for record REC”); if it has more than one entry for `key` (“header of record REC has more than one entry for key KEY (lines L1, L2); remove them first”); if it has no record line (“header of record REC has no record line”); if it was read from a URL (“header of record REC was read from a URL, and cannot be changed (give dir to write the changed header to a local directory)”); or if the changed header cannot be written (“can’t write header for record REC”).

```rust
let opt = WriteOptions {
    gains: vec![200.0],
    adcres: vec![12],
    units: vec![Some("mV".into())],
    metadata: vec![("Age".into(), "63".into())],
    ..Default::default()
};
write_record("pat", &[0; 360], 1, 360.0, &opt)?;
set_metadata("pat", "AGE", "64")?; // replaces the line: "# AGE: 64"
set_metadata("pat", "Date of birth", "1961-02-03")?; // adds "# Date of birth: 1961-02-03"
for (key, value) in read_metadata("pat")? {
    println!("{} = {}", s(&key), s(&value));
}
// AGE = 64
// Date of birth = 1961-02-03
if let Err(e) = set_metadata("pat", "a:b", "1") {
    println!("{e}"); // metadata key 'a:b' must not contain ':'
}
```

---

set_metadata_with

Set the value of a metadata entry, with options

```rust
pub fn set_metadata_with(
    record: impl AsRef<[u8]>,
    key: impl AsRef<[u8]>,
    value: impl AsRef<[u8]>,
    opt: &MetadataOptions,
) -> Result<(), Error>
```

As `set_metadata`, with the input directory and network access for reading the header, and the directory `dir` in which to write the changed header, given by `opt`, for this call only. With `dir`, the header that was read is unchanged, and the changed header is written to the file of the same name in `dir`, even if nothing changes.

record
: Record name (as for `set_metadata`). A header read from a URL can be changed only with `opt.dir`.

key
: The key (as for `set_metadata`).

value
: The value (as for `set_metadata`).

opt
: The options (Table 8.4): `input_dir`, `network` and `dir`.

Returns `Ok(())`.

Errors As `set_metadata`; also an `Error` if `opt.dir` does not exist (“directory D does not exist”).

```rust
// The header of 100s (in the current directory) is unchanged; the changed copy is
// written to the directory "copy" (which must exist).
std::fs::create_dir_all("copy")?;
let opt = MetadataOptions { dir: Some("copy".into()), ..Default::default() };
set_metadata_with("100s", "Age", "69", &opt)?;
let age = get_metadata("copy/100s", "Age")?.unwrap_or_default();
println!("{} {:?}", s(&age), get_metadata("100s", "Age")?); // 69 None
```

remove_metadata

Remove a metadata entry

```
pub fn remove_metadata(record: impl AsRef<[u8]>, key: impl AsRef<[u8]>) -> Result<usize, Error>
```

Removes a metadata entry from the header of a record. Every entry for `key` (ignoring letter case) is removed, with its line ending, so this also repairs a header that has more than one. Every other byte is kept. Nothing is written if the header has no entry for `key`.

record
: Record name (as for `set_metadata`).

key The key (as for `set_metadata`).

Returns The number of lines removed (0 if the header has no entry for `key`).

Errors As `set_metadata`, except that more than one entry for `key` is not an error.

```
let opt = WriteOptions {
    gains: vec![200.0],
    adcres: vec![12],
    units: vec![Some("mV".into())],
    metadata: vec![("Age".into(), "63".into()), ("Sex".into(), "F".into())],
    ..Default::default()
};
write_record("pat", &[0; 360], 1, 360.0, &opt)?;
println!("{}", remove_metadata("pat", "sex")?); // 1
println!("{}", remove_metadata("pat", "Weight")?); // 0: nothing is written
println!("{}", read_metadata("pat")?.len()); // 1
```

remove_metadata_with

Remove a metadata entry, with options

```
pub fn remove_metadata_with(
    record: impl AsRef<[u8]>,
    key: impl AsRef<[u8]>,
    opt: &MetadataOptions,
) -> Result<usize, Error>
```

As `remove_metadata`, with the options of `opt`, for this call only (as for `set_metadata_with`). With `dir`, the header is written to `dir` even if nothing is removed.

record
: Record name (as for `set_metadata`).

key The key (as for `set_metadata`).

opt The options (Table 8.4): `input_dir`, `network` and `dir`.

Returns The number of lines removed.

Errors As `remove_metadata`; also an `Error` if `opt.dir` does not exist.

```
let opt = WriteOptions {
    gains: vec![200.0],
    adcres: vec![12],
    units: vec![Some("mV".into())],
    metadata: vec![("Age".into(), "63".into()), ("Sex".into(), "F".into())],
```

```
    ..Default::default()
};
write_record("pat", &[0; 360], 1, 360.0, &opt)?;
std::fs::create_dir_all("out")?;
let opt = MetadataOptions { dir: Some("out".into()), ..Default::default() };
println!("{}", remove_metadata_with("pat", "Sex", &opt)?); // 1
println!("{} {}", read_metadata("pat")?.len(), read_metadata("out/pat")?.len()); // 2 1
```

---

update_metadata

Make several changes to the metadata

```
pub fn update_metadata<K: AsRef<[u8]>, V: AsRef<[u8]>>(
    record: impl AsRef<[u8]>,
    changes: &[(K, Option<V>)],
) -> Result<(), Error>
```

Makes several changes to the metadata of a record in one write. Each (key, value) of `changes` is applied in order, as `set_metadata` does for `Some(value)`, or as `remove_metadata` does for `None`. All the changes are checked first: if one is not valid, none is made.

record
: Record name (as for `set_metadata`).

changes
: The changes, as (key, value) pairs; a value of `None` removes the entry. No two keys may differ only in letter case. A list of removals only needs the type of its values: `&[("Dx", None::<&str>)]`. Empty `changes` change nothing (and write nothing).

Returns `Ok(())`.

Errors As `set_metadata`; also an `Error` if two keys differ only in letter case ("metadata key 'K' is given more than once (keys are compared without regard to letter case)").

```
let opt = WriteOptions {
    gains: vec![200.0],
    adcres: vec![12],
    units: vec![Some("mV".into())],
    metadata: vec![("Age".into(), "63".into()), ("Dx".into(), "164889003".into())],
    ..Default::default()
};
write_record("pat", &[0; 360], 1, 360.0, &opt)?;
update_metadata("pat", &[("Age", Some("64")), ("Dx", None), ("Sex", Some("F"))])?;
for (key, value) in read_metadata("pat")? {
    println!("{} = {}", s(&key), s(&value));
}
// Age = 64
// Sex = F
update_metadata("pat", &[("Sex", None::<&str>)])?; // removals only: the type of the values
println!("{}", read_metadata("pat")?.len()); // 1
```

---

update_metadata_with

Make several changes to the metadata, with options

```
pub fn update_metadata_with<K: AsRef<[u8]>, V: AsRef<[u8]>>(
```

```
    record: impl AsRef<[u8]>,
    changes: &[(K, Option<V>)],
    opt: &MetadataOptions,
) -> Result<(), Error>
```

As `update_metadata`, with the options of `opt`, for this call only (as for `set_metadata_with`). With `dir`, the header is written to `dir` even if `changes` is empty.

`record`
: Record name (as for `set_metadata`).

`changes`
: The changes (as for `update_metadata`).

`opt` The options (Table 8.4): `input_dir`, `network` and `dir`.

Returns `Ok(())`.

Errors As `update_metadata`; also an `Error` if `opt.dir` does not exist.

```
let opt = WriteOptions {
    gains: vec![200.0],
    adcres: vec![12],
    units: vec![Some("mV".into())],
    metadata: vec![("Age".into(), "64".into())],
    ..Default::default()
};
write_record("pat", &[0; 360], 1, 360.0, &opt)?;
// The header "pat" in the directory "in" is unchanged; the changed copy is written to
// the directory "out" (which must exist).
std::fs::create_dir_all("in")?;
std::fs::create_dir_all("out")?;
std::fs::copy("pat.hea", "in/pat.hea")?;
let opt = MetadataOptions {
    input_dir: Some("in".into()),
    dir: Some("out".into()),
    ..Default::default()
};
update_metadata_with("pat", &[("Age", Some("65"))], &opt)?;
let a = get_metadata("in/pat", "Age")?.unwrap_or_default();
let b = get_metadata("out/pat", "Age")?.unwrap_or_default();
println!("{} {}", s(&a), s(&b)); // 64 65
```

8.3.6 The options

The options of the high-level functions are the fields of the structures of Tables 8.1 to 8.4, all public, and all with a `Default` implementation; the times are `TimeSpec` values (Table 8.5). `ReadOptions`, `WriteOptions`, `AnnWriteOptions` and `TimeSpec` are defined in `wfdbx::record`, `MetadataOptions` in `wfdbx::metadata`; all are available at the root of the crate.

Table 8.1 The fields of `ReadOptions`, the options of `read_record` (all), `read_header_with` (`input_dir`, `network`, `legacy`), `read_annotations_with` (`from`, `to`, `input_dir`, `network`, `legacy`) and `check_record_with` (`input_dir`, `network`)

Field	Meaning (default)
`channels: Vec<String>`	the signals to return, in this order, by number (`"0"`, `"1"`, ...) or by name (`"V5"`, as in `Record::names()`, then as `findsig` finds it). Default empty: all the signals
`from: Option<TimeSpec>`	the first sample: a sample number (≥ 0) or a time string. Default `None`: 0
`to: Option<TimeSpec>`	the sample after the last one: a sample number or a time string. Default `None`: the end of the record
`highres: bool`	`true` to read a multi-frequency record at its highest sampling frequency, repeating the samples of the slower signals; `false` averages the samples of the faster signals to one per frame. Default `false`
`pad: bool`	`true` to replace each invalid sample by the previous valid sample of the same signal. Default `false`
`legacy: bool`	behave as the original WFDB package, for this call only. Default `false`
`input_dir: Option<String>`	the input directory for this call only (optional; see `wfdbx_set_input_dir`): a local directory, a ~ path or the URL of a directory, under which relative record names are read. It does not apply to full paths or URLs. `""` means the default (`WFDBX_INPUT_DIR`, or the current directory). Default `None`: the current setting
`network: Option<bool>`	`Some(true)` to allow opening URLs for this call only, `Some(false)` to forbid it (see `wfdbx_set_network`). Default `None`: the current setting
`checksum_warning: bool`	`true` to print a warning on the standard error when the samples do not match the checksums of the header (Section 8.3.8); the samples are returned either way. Default `true`

Table 8.2 The fields of `WriteOptions`, the options of `write_record` and `write_record_physical`. Each per-signal vector holds one value for all the signals, or one value per signal (or none, for those that have a default)

Field	Meaning (default)
`fmt: i32`	the storage format, the same for all the signals, such as `16`, `212`, `24` or `516` (FLAC) (Chapter 15). Every sample must fit the format: within its range (Table 15.1; -2048 to 2047 for format 212, for example), and, in format 8, within -128 to 127 of the sample before it. The smallest value of a format is the value of an invalid (missing) sample: a sample equal to it, or to `WFDB_INVALID_SAMPLE`, is read back as invalid (see `invalid_warning`). A FLAC file holds at most 8 signals. Default `16`
`gains: Vec<f64>`	the gains, in ADC units per physical unit: required, and finite. A gain of 0 marks a signal as uncalibrated, which only `write_record` (ADC units) allows. Negative gains are allowed
`baselines: Vec<i32>`	the ADC values that correspond to 0 physical units. Default: the `adczeros` values
`adczeros: Vec<i32>`	the ADC output for an input of 0 V. Default `vec![0]`
`adcres: Vec<i32>`	the ADC resolutions, in bits: required, each at least 1
`units: Vec<Option<Vec<u8>>>`	the physical units, such as `Some("mV".into())`: required for every signal whose gain is not 0; `None` for an uncalibrated signal
`names: Vec<Vec<u8>>`	the signal descriptions. Default: empty descriptions

Field	Meaning (default)
`samps_per_frame: Vec<i32>`	the number of samples per frame of each signal, for a multi-frequency record (each ≥ 1). Default: 1
`base_time: Option<Vec<u8>>`	the start time and date, such as `"12:00:00 01/01/2020"` (see `setbasetime`). Default `None`: none
`comments: Vec<Vec<u8>>`	the info strings, written to the header as comment lines (`"#"` followed by the string). Default: none
`metadata: Vec<(Vec<u8>, Vec<u8>)>`	metadata entries (Chapter 5) written to the header after the comments, as `# Key: value` (through `putinfo`), in this order. The keys and values follow the rules of `set_metadata`; no two keys may differ only in letter case, and none may be the key of an entry in `comments` (an error, before any file is created). Default: none
`legacy: bool`	behave as the original WFDB package, for this call only: the gain defaults to 200, and the ADC resolution to one based on the format. Default `false`
`dir: Option<PathBuf>`	the directory (which must exist) in which the files are created, for this call only; the record name must then be a plain name, without a directory, and the record opens from that directory under its plain name. Default `None` (or an empty path): the output directory
`invalid_warning: bool`	`true` to print a warning on the standard error when samples will be read back as invalid (missing) samples (Section 8.3.8): those equal to the invalid value of the format or to `WFDB_INVALID_SAMPLE`, other than the NaN samples of `write_record_physical`, which mark missing samples deliberately; `false` for no warning, when such samples are deliberate. Default `true`

Table 8.3 The fields of `AnnWriteOptions`, the options of `write_annotations_with`

Field	Meaning (default)
`dir: Option<PathBuf>`	the directory (which must exist) in which to create the file, for this call only; the record name must then be a plain name, without a directory: the file is `DIR/RECORD.ANNOTATOR`. Default `None`: the output directory
`sort: Option<bool>`	`Some(true)` always sorts annotations that are not in time order, `Some(false)` keeps the given order, for this call only. Default `None`: `WFDBANNSORT` decides (sorting is on unless it is `0`)

Table 8.5 `TimeSpec`, a sample number or a time string

Item	Meaning
`TimeSpec::Sample(i64)`	a sample number
`TimeSpec::Str(Vec<u8>)`	a time string, converted with `strtim` for the open record
`From<i64>`, `From<i32>`, `From<u32>`, `From<usize>`	the sample number: `3600.into()`
`From<&str>`, `From<String>`, `From<&[u8]>`	the time string: `"0:10".into()`
`fn resolve(&self, w: &mut Wfdb) -> i64`	the sample number; a time string is converted with `strtim` for the record open in `w`, and a time of day gives the elapsed time from the start of the record (the absolute value of what `strtim` returns)
`fn needs_freq(&self, w: &mut Wfdb) -> bool`	`true` if the time is a string that needs a sampling frequency to be converted (see `wfdbx_time_needs_freq`)

Table 8.4 The fields of `MetadataOptions`, the options of the metadata functions whose names end in `_with`

Field	Meaning (default)
`input_dir: Option<String>`	the input directory for reading the header, for this call only (optional), as in `ReadOptions`: it applies to relative record names only. Default `None`: the current setting
`network: Option<bool>`	network access for this call only, as in `ReadOptions`. Default `None`: the current setting
`dir: Option<PathBuf>`	for `set_metadata_with`, `remove_metadata_with` and `update_metadata_with`: the directory (which must exist; a ~ path is expanded) in which the changed header is written, under the file name of the header that was read, instead of changing that header. The header is then written even if nothing changes. The signal files are not copied, and the output directory is not used. Not used by `read_metadata_with` and `get_metadata_with`. Default `None` (or an empty path): the header is changed where it was read

8.3.7 The result types

The readers return structures of the types `Record`, `Header` and `Annotations`, whose fields and methods are listed in Tables 8.6 to 8.8; `check_record` returns a `CheckReport`, which holds `Issue` values (Tables 8.9 and 8.10). The signals of a record are described by `Siginfo` structures (Table 8.11), which are those of the WFDB library (Section 8.4). Their fields are public, and may be changed. `Record`, `Header` and `Annotations` implement `Clone`, `Debug`, `Default` and `PartialEq`; `CheckReport` and `Issue` implement `Clone`, `Debug`, `PartialEq` and `Eq`.

Table 8.6 The fields and methods of `Record`, the result of `read_record`

Member	Contents
`name: Vec<u8>`	the record name, as given to `read_record`
`fs: f64`	the sampling frequency of the returned samples, in Hz, or 0 if the header gives none; with `highres`, the highest sampling frequency of a multi-frequency record
`nsig: usize`	the number of signals returned
`signals: Vec<usize>`	the number, in the record, of each signal returned
`samples: Vec<i32>`	the samples in ADC units, frame by frame: `samples[i * nsig + s]` is sample `i` of signal `s`; invalid samples are `WFDB_INVALID_SAMPLE`
`siginfo: Vec<Siginfo>`	one per signal returned, in the order of the signals; `desc` has no white space at either end
`base_time: Option<String>`	the start time and date as text, to the second, such as `"[14:00:03 21/06/2024]"`; `None` if the header gives none
`base_time_ms: Option<i64>`	the start time in milliseconds after midnight (`"14:00:03.500"` gives `50403500`); `Some` exactly when `base_time` is
`base_date: Option<[i32; 3]>`	the start date as `[year, month, day]`; `None` if the header gives no date
`sampfrom: i64`	the number of the first sample returned

Member	Contents
`comments: Vec<Vec<u8>>`	the info strings of the record: the comments of the header after its record line (the first line that holds something other than spaces, tabs and CRs and does not begin with `#`, as the header parser finds it: an empty or blank first line is not the record line; in legacy mode, the first line that does not begin with `#`, as C's `getinfo` finds it), and the lines of its `.info` file if it has one (see `getinfo`)
`checksum_ok: bool`	`false` if the samples do not match the checksums of the header; checksums are checked only when a read reaches the end of the record, so a shorter read gives `true`
`legacy: bool`	`true` if the record was read in legacy mode; `physical()` then uses a gain of 200 for uncalibrated signals
`fn nsamples(&self) -> usize`	the number of samples per signal (the number of frames)
`fn at(&self, i: usize, s: usize) -> i32`	sample `i` of signal `s`, in ADC units; panics if `i` or `s` is out of range
`fn physical(&self, i: usize, s: usize)`	a `Result<f64, Error>`: sample `i` of signal `s` in physical units, $(value - baseline)/gain$; NaN if the sample is invalid. An `Error` ("signal N of record REC is uncalibrated") if the gain of the signal is 0
`fn physical_signal(&self, s: usize)`	a `Result<Vec<f64>, Error>`: all the samples of signal `s` in physical units (the same error)
`fn names(&self) -> Vec<Vec<u8>>`	the signal descriptions (empty if absent), without the white space that a header may put at either end of them

Table 8.7 The fields of `Header`, the result of `read_header` and `read_header_with`

Field	Contents
`name: Vec<u8>`	the record name, as given
`fs: f64`	the sampling frequency in Hz, as `sampfreq` gives it; 0 if the header gives none
`counter_freq: f64`	the counter frequency (`getcfreq`)
`base_counter: f64`	the base counter value (`getbasecount`)
`nsamples: i64`	the number of samples per signal, from the record line of the header (0 if the header does not give it)
`nsig: usize`	the number of signals
`siginfo: Vec<Siginfo>`	the signal specifications, one per signal; `desc` has no white space at either end
`base_time`, `base_time_ms`, `base_date`	the start time and date, as in `Record`
`comments: Vec<Vec<u8>>`	the info strings of the record, as in `Record`
`metadata: Vec<(Vec<u8>, Vec<u8>)>`	the metadata entries of the header, in file order, as `read_metadata` returns them: from the `.hea` file only; empty, without an error, for an EDF record and for a header read from the standard input

Table 8.8 The fields and methods of `Annotations`, the result of `read_annotations` and the argument of `write_annotations`: parallel vectors, annotation `i` being `samples[i]`, `codes[i]`, `symbols[i]`, …, in file order

Member	Contents
`record: Vec<u8>`, `annotator: Vec<u8>`	the record and annotator names, such as `b"100s"` and `b"atr"`
`fs: f64`	the frequency in which `samples` are counted, normally the sampling frequency of the record; 0 if unknown. `write_annotations` does not use it
`samples: Vec<i64>`	the annotation times, in samples
`codes: Vec<i32>`	the annotation codes, such as `NORMAL` or `PVC`
`symbols: Vec<Vec<u8>>`	the mnemonics of the codes, as `annstr` returns them, such as `b"N"`, `b"V"` or `b"+"`
`subtypes: Vec<i32>`, `chans: Vec<i32>`, `nums: Vec<i32>`	the subtype, channel number and annotator number of each annotation
`aux: Vec<Option<Vec<u8>>>`	the auxiliary text of each annotation, without its length byte and up to its first NUL byte, or `None` where there is none
`fn len(&self) -> usize`	the number of annotations
`fn is_empty(&self) -> bool`	`true` if there are no annotations

Table 8.9 The fields and methods of `CheckReport`, the result of `check_record`

Member	Contents
`record: Vec<u8>`	the record name, as given
`ok: bool`	`true` if no issue has the severity `Severity::Error`
`issues: Vec<Issue>`	the problems, in the order of the checks
`fn lines(&self) -> Vec<String>`	the text form of the report: one line per issue (the `Display` form of the `Issue`), or the one line `no problems found`
`Display`	the lines of `lines()`, joined by line feeds (no final line feed)

Table 8.10 The fields of `Issue`, a problem found by `check_record`, and of `Severity`

Member	Contents
`severity: Severity`	`Severity::Error`, `Severity::Warning` or `Severity::Info`; `as_str()` and `Display` give `"error"`, `"warning"` or `"info"`
`code: &'static str`	a stable identifier of the kind of problem, such as `"checksum"` or `"gain_missing"` (Table 6.7)
`r#where: String`	where the problem is: `""` (the record), `"signal N"`, `"segment SEG"`, `"segment SEG, signal N"` or `"annotator ANN"` (the field is named `where`, a keyword of Rust, written `r#where`)
`message: String`	the description of the problem
`Display`	`SEVERITY: WHERE: MESSAGE`, or `SEVERITY: MESSAGE` when `where` is empty

Table 8.11 The fields of `Siginfo` (C's `WFDB_Siginfo`), the specification of a signal

Field	Contents
`fname: Option<Vec<u8>>`	the name of the signal file, as the header gives it
`desc: Option<Vec<u8>>`	the signal description, such as `b"MLII"`
`units: Option<Vec<u8>>`	the physical units, such as `b"mV"`; `None` if not specified
`gain: f64`	the gain, in ADC units per physical unit (0: uncalibrated)
`initval: i32`	the value of the first sample
`group: u32`	the signal group number (the signals of one group share one file)
`fmt: i32`	the storage format, such as `16`, `212` or `516`
`spf: i32`	the number of samples per frame (more than 1 for the faster signals of a multi-frequency record)
`bsize: i32`	the block size, in bytes (0 for ordinary files)
`adcres: i32`	the ADC resolution, in bits (0: not specified)
`adczero: i32`	the ADC output for an input of 0 V
`baseline: i32`	the ADC value that corresponds to 0 physical units
`nsamp: i64`	the number of samples (0: not specified)
`cksum: i32`	the 16-bit checksum of the samples

8.3.8 Warnings and errors

Errors. Each high-level function returns a `Result`; its error type is `wfdbx::record::Error`, the one public item of the high-level API that is not available at the root of the crate. `e.message()` and `Display` give the message, which describes the failed operation, such as "can't read record REC"; the type implements `std::error::Error`, `Clone`, `Debug`, `PartialEq` and `Eq`, so it converts to `Box<dyn std::error::Error>` with the `?` operator. There is one error type: the messages tell the cases apart (Table 8.12). Each function checks its arguments before it creates or changes a file, so an invalid argument leaves the files as they were; `write_record` and `write_annotations` also write through temporary files, so that a failure while writing (a full disk, for example) leaves the existing files as they were. A signal file that ends before the length its header gives is a read error even when the checksums do not match either ("... (getvec status -3)"): `read_record` never returns an incomplete frame, and `check_record` reports the short file (`length_short`). The library may also print a message of its own on the standard error when it cannot open a file (as in `init: can't open header for record REC`); `wfdberror()` returns the last such message, and `wfdbquiet()` turns these messages off (Section 8.4). `Record::at`, like the indexing of the result vectors, panics on an index out of range.

Warnings. Warnings are printed on the standard error with the library's messages, through `wfdb_error`: `wfdbquiet()` silences them, and `wfdberror()` returns the last one. There is no callback.

Settings. Whether a function returns a result or an error, the library's settings are those it had before the call: the input and output directories, network access and legacy mode (set for the call by the options), and, for `read_record`, `read_header`, `read_annotations`, `write_record`, `write_record_physical`, `write_annotations`, their `_with` forms and `check_record`, the getvec mode (`getgvmode()`) and the time resolution of output annotators (`getafreq()`), which opening an EDF record sets (the `WFDB_HIGHRES` bit, and the record's sampling frequency when no resolution is set) and `wfdbquit()` does not reset. A later `write_annotations` therefore records no frequency that an EDF record read before it left behind; its own `fs` applies to its file only.

Table 8.12 The kinds of errors of the high-level API

Kind	Messages
files	a record, header or annotation file cannot be found, read, created or written completely (for example on a full disk): “can’t read record REC”, “can’t read header for record REC”, “can’t write header for record REC”, “write error on record REC”, “directory D does not exist”, . . . ; a signal or annotation file ends before its header or its end-of-file marker says (“. . . (getvec status S)”, “. . . (getann status S)”)
arguments	an invalid argument or option, reported before anything is written (most messages of `write_record` and `write_annotations` begin with the function’s name); for the metadata functions, an invalid key or value, a record name that is empty or `"-"`, a key that the header has more than once, a header without a record line, or a header read from a URL to be changed without `dir`
missing values	the operation needs a value that the record does not give: a time in seconds without a sampling frequency (“sampling frequency of record REC is not specified”), physical units without a gain (“signal N of record REC is uncalibrated”)

The checksum warning. When `read_record` reads to the end of a record whose samples do not match the checksums of its header, it returns all the samples and prints the warning “WFDB-x: checksum error in record REC: the signal data do not match the checksums in its header”. `checksum_warning: false` turns the warning off; `checksum_ok` of the result is `false` either way.

The invalid-sample warning. When `write_record` or `write_record_physical` writes samples that will be read back as invalid (missing) samples (those equal to the smallest value of the format or to `WFDB_INVALID_SAMPLE`, other than the NaN samples of physical units), it prints, for each such signal, the warning “WFDB-x: N sample(s) of signal K will be read back as invalid (missing) samples (format F stores an invalid sample as LO)”. `invalid_warning: false` in `WriteOptions` turns the warning off.

The duplicate-key warning. When a header holds two metadata entries whose keys differ only in letter case (written by other software), `read_metadata`, `get_metadata` and `read_header` use the first and print the warning “WFDB-x: header of record REC has more than one entry for key KEY (lines L1, L2); the first is used” (KEY spelled as in its first entry). `set_metadata` and `update_metadata` then refuse to change that key (an `Error`), and `remove_metadata` removes all of its entries.

```
// A record whose last sample is changed after it was written: the checksum fails.
let x: Vec<i32> = (0..100).collect();
let opt = WriteOptions {
    gains: vec![100.0],
    adcres: vec![12],
    units: vec![Some("mV".into())],
    ..Default::default()
};
write_record("ck", &x, 1, 100.0, &opt)?;
let mut dat = std::fs::read("ck.dat")?;
let n = dat.len();
dat[n - 2] = 0;
dat[n - 1] = 0;
std::fs::write("ck.dat", &dat)?;
let rec = read_record("ck", &ReadOptions::default())?; // prints the checksum warning
println!("{} {}", rec.nsamples(), rec.checksum_ok); // 100 false: all the samples
let quiet = ReadOptions { checksum_warning: false, ..Default::default() };
let rec = read_record("ck", &quiet)?; // no warning
```

```rust
// Two entries for one key: the first is used, with a warning.
let dup = WriteOptions {
    gains: vec![0.0],
    adcres: vec![12],
    comments: vec![" Age: 63".into(), " AGE: 64".into()],
    ..Default::default()
};
write_record("dup", &[0; 10], 1, 100.0, &dup)?;
let age = get_metadata("dup", "age")?.unwrap_or_default(); // with a warning
println!("{}", s(&age)); // 63
if let Err(e) = set_metadata("dup", "Age", "65") {
    println!("{e}");
    // header of record dup has more than one entry for key Age (lines 3, 4); remove
    // them first
}
println!("{}", remove_metadata("dup", "age")?); // 2

match read_record("nosuchrecord", &ReadOptions::default()) {
    Err(e) => println!("{}", e.message()), // can't read record nosuchrecord
    Ok(_) => {}
}
```

8.4 The WFDB library functions in Rust

Every function of the WFDB library is a method of `Wfdb` and a free function of the crate, under its C name, with the same arguments, in the same order, and the same return values, so that programs and documentation written for the C library carry over directly. The functions, their arguments and their results are documented once, for all the languages, in Chapter 13; this section gives the rules by which the C declarations become Rust functions (see also Section 13.1.3). The doc comment of each function summarizes it, and says what WFDB-x changes in it.

8.4.1 Conventions

The library state. The state of the C library (the open record, its signals, the annotators, the WFDB path, the time base, ...) is held by a `Wfdb` value (`Wfdb::new()`), and each C function is a method of it with the C name: `w.isigopen(...)`, `w.getvec(...)`. Each method is also a free function with the same name and arguments (`wfdbx::isigopen(...)`), which uses the default `Wfdb` value of the calling thread; a free function must not be called from inside a closure given to `wfdbx::with_default`, which borrows that value. Separate `Wfdb` values share no state, unlike the C library. The functions that use no state are free functions only (`wfdbldflags`, `wfdbcflags`, `wfdbx_write_errors`, `isann`, ...; Section 13.1.3).

Names and modules. The functions, the constants (`WFDB_READ`, `WFDB_INVALID_SAMPLE`, `WFDB_HIGHRES`, ...), the structures and the annotation codes (`NORMAL`, `PVC`, ..., Chapter 15) keep their C names, and are available at the root of the crate (`use wfdbx::*;`). They are defined in the modules of Table 8.13. The macros of `<wfdb/ecgmap.h>` are functions (`isann`, `isqrs`, `map1`, ..., and `wfdb_isann`, `wfdb_isqrs`, ...).

Arguments. String arguments are byte strings: they accept `&str`, `&[u8]` or `b"..."`, and a byte string ends at its first NUL byte, as a C string does. Where C accepts `NULL`, they are `impl OptBytes`, which also accepts `Option<&[u8]>`, `Option<&str>` and `&Option<Vec<u8>>`; `wfdbx::NULL` is a typed `None`: `sampfreq(NULL)` returns the frequency of the open record, and `getinfo(NULL)` continues with the info strings of the last record. Where C fills an array, the caller passes a mutable slice (`isigopen(record, &mut si, n)`, `getvec(&mut v)`); an empty slice stands for C's `NULL`

Table 8.13 The modules of the crate `wfdbx`

Module	Contents
`record`	the high-level functions of reading and writing, their option and result types, and `Error`
`check`	`check_record`, `check_record_with`, `CheckReport`, `Issue`, `Severity`
`metadata`	the metadata functions, `MetadataOptions`, `Metadata`
`signal`	signals and headers (port of `lib/signal.c`)
`annot`	annotations (`lib/annot.c`)
`calib`	the calibration list (`lib/calib.c`)
`wfdbio`	file names, the input and output directories, network access, error messages, `WfdbFile` (`lib/wfdbio.c`)
`wfdbinit`	`wfdbinit`, `wfdbquit`, `wfdbflush` (`lib/wfdbinit.c`)
`ecgcodes`	the annotation codes and the functions of `<wfdb/ecgmap.h>`
`types`	the types, constants and structures of `<wfdb/wfdb.h>`
`wfdb`	the type `Wfdb`, `with_default`, `OptBytes`, `NULL`
`cstd`	the C standard functions that the tools use, such as a `printf` that formats numbers exactly as the GNU C library does (`cformat!`, `cprintf!`)

(`isigopen(record, &mut [], 0)`); `osigopen(record, &mut [], 0)` closes the open output signals (unless the record name begins with `+`), reads the header, opens nothing and returns 0. Where C fills a structure, the caller passes a mutable reference (`getann(0, &mut annot)`). `getseginfo` takes an `&mut Option<Vec<Seginfo>>`, which receives a copy of the segment table.

Strings are returned by value, as `Vec<u8>`, or as `Option<Vec<u8>>` where the C function may return `NULL` (`anndesc`, `getinfo`, `wfdbfile`, ...); unlike C's pointers to static buffers, they stay valid after later calls. Text is handled as bytes.

Return values and errors. The functions return what the C functions return: an `i32` status (negative on errors), a number of signals, an `f64`, a string or `None`. Errors are reported by these values and by a message on the standard error, as in C, never by `Result` or panics. `wfdberror()` returns the last message; `wfdbquiet()` turns the messages off and `wfdbverbose()` on again. `wfdb_error(msg)` takes a message already formatted, with the macro `cformat!` (which formats as C's `printf`), and `wfdb_error_fmt(fmt, args)` formats it.

State. As in C, a `Wfdb` value keeps its state between calls: the open record, its input signals, the output signals and the annotators remain open until `wfdbquit()` closes them. `wfdbquit()` leaves the input and output directories and the network setting as they are, and so, as in C, the time resolution of output annotators (`getafreq()`), which opening an EDF record sets to its sampling frequency when none is set, and the getvec mode (`getgvmode()`), to which it adds `WFDB_HIGHRES`. Dropping a `Wfdb` value flushes its output buffers, as C's `exit` does, but does not close its records (no annotation end marks, no header updates): a program calls `wfdbquit()` before it ends. The high-level functions of Section 8.3 (except the metadata functions) call `wfdbquit()` on the `Wfdb` value they use, which closes anything opened with the library functions on that value, and leave those two settings as they were.

Additions. `getvecs(buffer, n)` reads up to `n` sample vectors at once, with exactly the result of `n` calls of `getvec`, but decoding whole blocks of the signal files; `getvecs_status(buffer, n)` also gives the status of the last `getvec` call (Section 13.11):

```
pub fn getvecs(out: &mut [i32], n: i64) -> i64
pub fn getvecs_status(out: &mut [i32], n: i64) -> (i64, i32)
```

`WfdbFile` is C's `FILE *` (or `WFDB_FILE`): `wfdb_open` and `wfdb_fopen` return an `Option<WfdbFile>`, whose methods (`getc`, `read`, `fgets`, `write`, `seek`, `close`, ...) are those of the C standard library.

8.4.2 The structures

The structures of the C library have the names of the C structures without the prefix `WFDB_`, and the C field names, with C's zero values as `Default` (Table 8.14).

Table 8.14 The C structures in Rust

Structure	Fields
`Siginfo`	`fname`, `desc`, `units` (`Option<Vec<u8>>`), `gain`, `initval`, `group`, `fmt`, `spf`, `bsize`, `adcres`, `adczero`, `baseline`, `nsamp`, `cksum` (Table 8.11)
`Seginfo`	`recname` (the name of the segment, `Vec<u8>`), `nsamp` (its number of samples, `i64`), `samp0` (the number of its first sample in the record, `i64`)
`Anninfo`	`name` (the annotator name, `Vec<u8>`), `stat` (`i32`: `WFDB_READ` or `WFDB_WRITE` for the MIT format, `WFDB_AHA_READ` or `WFDB_AHA_WRITE` for the AHA format)
`Annotation`	`time` (`i64`, in sample intervals), `anntyp` (`i8`), `subtyp` (`i8`), `chan` (`u8`), `num` (`i8`), `aux` (`Option<Vec<u8>>`: absent, or bytes whose first byte is the length of the text that follows, as in C)
`Calinfo`	`low`, `high` (the levels of the calibration pulse, in physical units, `f64`), `scale` (physical units per cm), `sigtype` and `units` (`Option<Vec<u8>>`), `caltype` (`i32`)

The integer types follow the C reference platform: `WFDB_Sample` and C's `int` are `i32`, `WFDB_Time`, `WFDB_Date` and C's `long` are `i64`, `WFDB_Frequency` and `WFDB_Gain` are `f64`, C's `char` is `i8`, and `WFDB_Signal`, `WFDB_Group` and `WFDB_Annotator` are `u32`. C's wrap-around arithmetic is reproduced.

8.4.3 The settings functions

The functions of Table 8.15 are additions of WFDB-x to the library: they set where files are read and written, network access and legacy mode, for the high-level functions, the library functions and the tools alike (Chapters 3 and 4), and give the path under which a program opens a file that it reads or creates itself, as the tools do (`wfdbx_input_path`, `wfdbx_output_path`; Section 3.3.2). Like the other library functions, each is a method of `Wfdb` and a free function, except `wfdbx_write_errors` and `wfdbx_note_write_error` (free functions only: the count belongs to the process). Each `set` function returns the previous setting, so that it can be restored. The C function `setwfdb(p)` sets the input directory too (`p` must name at most one directory), and `getwfdb()` returns it (`"."` if none is set); in legacy mode they set and return the WFDB path, as in C.

Table 8.15 The settings functions (the signatures of the free functions)

Function	Effect and result
`wfdbx_set_input_dir(dir: impl OptBytes) -> Option<Vec<u8>>`	sets the input directory: a local directory, a ~ path or a URL; `NULL` or `""` restores the default (`WFDBX_INPUT_DIR`, or else the current directory). Returns the previous input directory, or `None` if none was set
`wfdbx_get_input_dir() -> Option<Vec<u8>>`	returns the input directory, or `None` if none is set

Function	Effect and result
`wfdbx_set_output_dir(dir: impl OptBytes) -> Option<Vec<u8>>`	sets the output directory: a local directory or a ~ path (a URL is refused, with a message, and the setting is unchanged); `NULL` or `""` restores the default (`WFDBX_OUTPUT_DIR`, or else the current directory). Returns the previous output directory, or `None`
`wfdbx_get_output_dir() -> Option<Vec<u8>>`	returns the output directory, or `None` if none is set
`wfdbx_set_network(on: i32) -> i32`	turns network access on (> 0) or off (0); a negative number restores the default (on if `WFDBX_NETWORK` is a nonzero number). Returns the previous setting, 1 or 0
`wfdbx_get_network() -> i32`	returns 1 if network access is on, 0 otherwise
`wfdbx_set_legacy(on: i32) -> i32`	turns legacy mode on (nonzero) or off (0). Returns the previous setting
`wfdbx_get_legacy() -> i32`	returns 1 if legacy mode is on, 0 otherwise
`wfdbx_legacy_args(argv: &mut Vec<Vec<u8>>) -> i32`	if `argv[1]` is `"--legacy"`, removes it from the arguments, turns legacy mode on and returns 1; otherwise returns 0. It also enables the exit check of the tools in `cstd::exit` (an exit status of 0 becomes 1 after a write error, outside legacy mode)
`wfdbx_time_needs_freq(string: impl AsRef<[u8]>) -> i32`	returns 1 if `strtim(string)` needs a sampling frequency, 0 otherwise
`wfdbx_write_errors() -> i32`	returns the number of write errors since the program started (output files that could not be written, flushed or closed)
`wfdbx_note_write_error()`	counts a write error, for a program that writes a file itself
`wfdbx_input_path(name: impl OptBytes) -> Option<Vec<u8>>`	returns the path under which a program opens a file that it reads itself: in normal mode, a relative name under the input directory, a ~ name expanded, an absolute name as given; in legacy mode, the name as given. `NULL`, `""` and `"-"` are returned as given; `None` for a URL, or a name that becomes one under the input directory

`wfdbx_output_path(name: impl OptBytes) -> Option<Vec<u8>>`	returns the path under which a program creates a file that it writes itself: in normal mode, a relative name under the output directory (whose missing directories are then created), a ~ name expanded, an absolute name as given; in legacy mode, any name under the output directory, as a plain prefix. `NULL`, `""` and `"-"` are returned as given; `None` for a URL

8.4.4 Examples

Reading the samples and the annotations of a record, as a C program does (the record `100s` is read from the current directory):

```
use wfdbx::*;

fn main() {
    let mut si = vec![Siginfo::default(); 2]; // receives one Siginfo per signal
    let nsig = isigopen("100s", &mut si, 2); // open 2 signals
    let desc = si[0].desc.clone().unwrap_or_default();
    println!("{} {} {} {}", nsig, String::from_utf8_lossy(&desc), si[0].fmt, si[0].gain);
    // 2 MLII 212 200
    println!("{}", sampfreq(NULL)); // 360: the frequency of the open record

    let mut v = [0i32; 2]; // receives one sample per signal
    isigsettime(strtim("0:30")); // skip to 30 seconds
    getvec(&mut v);
    println!("{} {} {}", v[0], v[1], aduphys(0, v[0])); // 947 968 -0.385

    let mut frames = vec![0i32; 360 * 2]; // the next second, in one call
    let (n, status) = getvecs_status(&mut frames, 360);
    println!("{n} {status}"); // 360 2

    let ai = Anninfo { name: b"atr".to_vec(), stat: WFDB_READ };
    annopen("100s", &[ai], 1);
    let mut a = Annotation::default();
    let mut count = 0;
    while getann(0, &mut a) == 0 {
        // 0: an annotation was read; -1: the end
        if count < 2 {
            let t = mstimstr(a.time);
            println!("{} {}", String::from_utf8_lossy(&t), String::from_utf8_lossy(&annstr(a.
    ↪ anntyp as i32)));
        }
        count += 1;
    }
    println!("{count} annotations"); // 75 annotations

    wfdbquiet(); // no messages on the standard error
    let stat = isigopen("nosuchrecord", &mut [], 0);
    print!("{} {}", stat, String::from_utf8_lossy(&wfdberror()));
    // -1 init: can't open header for record nosuchrecord
    wfdbverbose();
    wfdbquit(); // close everything
}
```

Writing a record and an annotation file, sample by sample and annotation by annotation:

```
use wfdbx::*;

fn main() {
    let out = Siginfo {
        fname: Some(b"ramp.dat".to_vec()),
        desc: Some(b"ramp".to_vec()),
        units: Some(b"mV".to_vec()),
        gain: 200.0,
        fmt: 16,
        adcres: 12,
        ..Default::default()
    };
    println!("{}", osigfopen(&[out], 1)); // 1: one output signal
    setsampfreq(250.0);
    for k in 0..500 {
        putvec(&[k]);
    }
    println!("{}", newheader("ramp")); // 0: ramp.hea is written
    let ai = Anninfo { name: b"qrs".to_vec(), stat: WFDB_WRITE };
    annopen("ramp", &[ai], 1);
    for t in [100, 350] {
        let a = Annotation { time: t, anntyp: NORMAL as i8, ..Default::default() };
        putann(0, &a);
    }
    wfdbquit(); // completes ramp.dat and ramp.qrs
}
```

8.5 The command-line tools in Rust

The 33 command-line tools (`rdsamp`, `rdann`, `wrsamp`, `xform`, ...) are separate programs of the crate, one per file of `rust/src/bin`, built in `rust/target/release/` (or installed in `~/.cargo/bin` by `cargo install`), and run by their names. Each follows its C program line by line. The tools, their options, their output and their exit status are those of the C tools, and are documented in Chapter 14.

```
export PATH=$PWD/rust/target/release:$PATH
export WFDBX_INPUT_DIR=tests/data     # run from the root of the repository
rdsamp -r 100s -t 10 -p     # the first 10 seconds of 100s, in physical units
rdann -r 100s -a atr        # its annotations
rdsamp --legacy -r 100s     # legacy mode: --legacy as the first argument
```

There is no launcher: a tool is run by its own name, and it has no `--version` option, like the C tools. The tools have the same names as the original WFDB tools: if both are installed, the one that comes first in the `PATH` runs. When an output file or the standard output cannot be written (a full disk, for example), a tool reports it ("PROGRAM: write error on standard output") and exits with status 1; with `--legacy`, it keeps the original exit status. A tool whose standard output is a closed pipe ends with status 141, as a C program ended by `SIGPIPE`.

Environment variables. The tools, like the rest of the library, use the environment variables of Table 8.16, also for the files that they read or create themselves (Section 3.3.2): the `-i` file of `wrsamp`, `edf2mit`, `wav2mit`, `a2m` and `ad2m`, the input files of `ahaecg2mit` and the `-S` script of `xform` are read from the input directory, and the WAV file of `mit2wav`, the EDF file of `mit2edf`, the master header of `snip -m` and the `.des`, `.ecg`, `.key` and `.txt` files of `parsescp` are created in the output

directory. Such a file cannot be read from a URL. In legacy mode, the files that a tool reads itself are opened as named, and the output directory, if set, is a plain prefix of every output name.

Table 8.16 The environment variables used by the Rust library

Variable	Meaning
`WFDBX_INPUT_DIR`	the input directory (a directory, a ~ path or a URL), when `wfdbx_set_input_dir` does not set one
`WFDBX_OUTPUT_DIR`	the output directory, when `wfdbx_set_output_dir` does not set one
`WFDBX_NETWORK`	network access is on if it is a nonzero number, when `wfdbx_set_network` does not set it
`WFDBANNSORT`	`0` to keep annotation files in the order written (no sorting when they are closed)
`WFDBGVMODE`	the default getvec mode (`setgvmode` with a negative mode)
`WFDBCAL`	the calibration file read by `calopen(NULL)` (default `wfdbcal`)
`WFDB`	the WFDB path, used in legacy mode only (outside legacy mode, it is ignored, with a message)
`HOME` (`USERPROFILE` on Windows)	the home directory, for ~ paths

Tools that write signal parameters require them as options: `wrsamp` (with `-o`) requires `-F` (frequency), `-G` (gain), `-R` (ADC resolution) and, for calibrated signals, `-U` (units); `wav2mit` requires `-G`, and `-U` unless the gain is 0; `ad2m` and `ahaecg2mit` require `-F`, `-G`, `-R`, and `-U` unless the gain is 0; `a2m`, `rdann`, `ann2rr`, `sortann` and `m2a` require `-F` when the record's header gives no frequency (Chapter 14).

8.6 Testing

The tests of the Rust implementation are the unit tests of the modules, the integration tests of `rust/tests` and the examples of the doc comments (which read the sample record `100s` from `tests/data`). From the directory `rust`:

```
cargo test --release                      # all the tests, and the doc examples
cargo test --release --test metadata      # one file of rust/tests
cargo test --release shared_cases         # selected tests, by name
cargo clippy --release --all-targets      # lints
```

The integration tests cover the WFDB library functions (`signals.rs`, `annotations.rs`, `wfdbio.rs`, `bulk.rs`, the C standard functions in `cstd.rs`), finding files and the settings (`files.rs`), network access (`network.rs`), the high-level functions of reading and writing (in `signals.rs`), the metadata functions (`metadata.rs`, with the cases shared by the seven languages, in `tests/metadata`) and `check_record` (`check.rs`). They run in temporary copies of the test records of `tests/data`. The tests of network access use a local HTTP server on 127.0.0.1 only, and need `curl`. The tests of `check_record` write the corpus of `python/tests/check_corpus.py` with `python3` (or the program named by the environment variable `PYTHON`), NumPy and the Python package of `python/src`, and compare the reports with its `expected.txt`. `cargo run --release --example check_cases DIR` prints the reports for a corpus in `DIR`.

The cross-language conformance suite runs every tool and compares its results, byte for byte, with those of the original C programs (1155 cases), and, with `--legacy`, in legacy mode (886 cases). It runs the tools of `rust/target/release`:

```
cargo build --release --manifest-path rust/Cargo.toml
PYTHONPATH=$PWD/python/src python3 tests/conformance/harness.py check --impl rust -j8
```

```
PYTHONPATH=$PWD/python/src python3 tests/conformance/harness.py check --impl rust -j8 --legacy
```

CHAPTER 9

Julia

The Julia implementation of WFDB-x is the package `WFDBX` (module `WFDBX`), in the directory `julia` of the repository: a library and the 33 command-line tools, run through one driver script. It is a complete Julia implementation of the WFDB Software Package: it uses only Julia and its standard libraries (no other package), and never calls the C library.

It offers WFDB-x in three ways:

- **The high-level API** (Section 9.3): functions that read and write whole records, headers, annotation files and header metadata as Julia matrices, vectors and structures, in one call, with keyword arguments, and that throw a `WfdbError` (or, for some invalid arguments, an `ArgumentError`) on errors. They read signal files in large blocks, and are the fastest way to read a record. They are an addition of WFDB-x: the C package has no equivalent. They are the recommended interface for new code, analysis and notebooks.
- **The WFDB library functions** under their C names (`isigopen`, `getvec`, `getann`, `putann`, `...`), with the behavior of the C library (Section 9.4). They are documented once, for all the languages, in Chapter 13.
- **The command-line tools** (`rdsamp`, `rdann`, `wrsamp`, `xform` and 29 others), with the options, output and exit status of the C tools, run through the driver script `julia/bin/wfdbx.jl` (Section 9.5). They are documented in Chapter 14.

The high-level functions return exactly the samples and annotations that the library functions (and the C library) give.

9.1 Installation and requirements

9.1.1 Requirements

WFDB-x for Julia needs Julia 1.9 or later, on Linux, macOS or Windows (the Windows code paths are not tested). It has no dependency beyond Julia's standard libraries: no WFDB library and no libFLAC (the FLAC formats 508, 516 and 524 are read and written by WFDB-x's own codec). Network access (Section 3.4) uses the standard library `Downloads`, which is loaded only when the first URL is read. The unit tests use the standard libraries `Test` and `Sockets`, and the tests of `check_record` also run `python3`, with NumPy (Section 9.6).

9.1.2 Installing the package

The package is installed from a copy of the repository, with the package manager:

```
using Pkg
Pkg.develop(path="/path/to/wfdb-x/julia")
```

In the package REPL, `] dev /path/to/wfdb-x/julia` does the same. The package is `WFDBX`, version `1.2.3` (its library under the GNU Lesser General Public License, version 2.1 or later, its tools under the GNU General Public License, version 3 or later: Section 1.9), and is loaded with `using WFDBX`. The package is the directory `julia`, which holds copies of the license files (`LICENSE`, `LICENSE-GPL`)

and of the file NOTICE of the repository (the licenses of WFDB-x and the copyright notice and BSD-3-Clause license of libFLAC 1.4.3, whose stream decoder the FLAC decoder translates), so that a copy of that directory carries them. The first `using WFDBX` precompiles the package, running a workload of every part of the library and of every tool, so that later sessions and tool runs load it without compiling anything. Changing a source file of the package makes the next session precompile it again.

9.1.3 The command-line tools

The tools are run through the driver script `julia/bin/wfdbx.jl`: its first argument names the tool, and the other arguments go to the tool. A small shell script, saved as `wfdbx` in a directory of the PATH, makes the command shorter (Section 9.5):

```
#!/bin/sh
# wfdbx: run a WFDB-x tool, as in `wfdbx rdsamp -r 100s -t 10`
exec julia --startup-file=no --project=/path/to/wfdb-x/julia \
    /path/to/wfdb-x/julia/bin/wfdbx.jl "$@"
```

`wfdbx --version` then prints the version of WFDB-x, `WFDB-x 1.2.3`. The docstrings of the package document each function (`?read_record` in the REPL), with examples.

9.1.4 The sample record

The repository includes the sample record `100s`, the sample record of the WFDB Software Package: the first minute of record 100 of the MIT-BIH Arrhythmia Database, two ECG signals (`MLII` and `V5`) sampled at 360 Hz, with the reference beat annotations (annotator `atr`). Its files (`100s.hea`, `100s.dat`, `100s.atr`) are in the directory `tests/data` of a copy of the repository, with the other test records of WFDB-x. The package does not install them: programs name that directory as their input directory (`wfdbx_set_input_dir("/path/to/wfdb-x/tests/data")`, or the keyword argument `input_dir`). Work that uses the MIT-BIH Arrhythmia Database cites it as described on PhysioNet.

9.2 Quick start

This program reads the header, ten seconds of the signals and the annotations of the sample record `100s`, from the directory `tests/data` of the repository (the input directory of each call), and computes the mean heart rate. It then writes the first ten seconds as a new record, `first10`, in the current directory, with two metadata entries in its header.

```
using WFDBX

# The sample record 100s, in the directory tests/data of the repository.
const DATA = "tests/data"

# The header only: no samples are read.
hdr = read_header("100s"; input_dir=DATA)
println(hdr.nsig, " ", hdr.fs, " ", hdr.nsamples)     # 2 360.0 21600

# Seconds 10 to 20 of both signals, in ADC units (one column per signal).
rec = read_record("100s"; from="0:10", to="0:20", input_dir=DATA)
println(size(rec.samples))                            # (3600, 2)
x = physical(rec)                                     # in physical units
println(rec.samples[1, 1], " ", x[1, 1])              # 946 -0.39

# The beat annotations, and the mean heart rate.
ann = read_annotations("100s", "atr"; input_dir=DATA)
```

```
beats = ann.samples[[isqrs(c) != 0 for c in ann.codes]]
rr = (beats[end] - beats[1]) / ann.fs / (length(beats) - 1)
println(length(beats), " beats, ", round(60 / rr; digits=1), " beats per minute")
# 74 beats, 73.9 beats per minute

# Write the first 10 seconds as the record "first10" (first10.hea and first10.dat, in
# the current directory, the output directory), with the signal specifications of 100s.
r10 = read_record("100s"; to="0:10", input_dir=DATA)
write_record("first10", r10.samples, r10.fs; gains=200.0, baselines=1024, adczeros=1024,
             adcres=11, units="mV", names=signal_names(r10),
             metadata=["Source" => "MIT-BIH record 100"])

# Header metadata: "# Key: value" lines of the header (keys and values are bytes).
set_metadata("first10", "Age", "69")
for (key, value) in read_metadata("first10")
    println(String(key), " = ", String(value))
end
# Source = MIT-BIH record 100
# Age = 69
println(check_record("first10"))                    # no problems found
```

The program runs from the root of the repository. The input directory is given to each call that reads `100s` (`input_dir`), so the record `first10`, written to the current directory, is then read from it under its plain name. The header `100s.hea` gives no units: the program writes `"mV"` explicitly, since WFDB-x never assumes units, a gain, an ADC resolution or a sampling frequency that a record does not give (Chapter 4).

9.3 The high-level API

The high-level API is made of the functions of this section, defined in the files `highlevel.jl`, `check.jl` and `metadata.jl` of `julia/src` and exported by the module `WFDBX`, of the types of their results (Section 9.3.6), and of the exception type `WfdbError` (Section 9.3.7). The docstrings of the package document each of them.

9.3.1 Conventions

The library state: `[w::Wfdb]`. The state of the library (the open record, the annotators, the input and output directories, network access, legacy mode) is held by an object of the abstract type `Wfdb`; `Wfdb()` creates a new one, and `lib()` returns the default object of the process. Each high-level function that uses the state (all but `physical`, `signal_names` and `lines`) has two methods: one that takes a `Wfdb` object `w` as its first argument, such as `read_record(w, "100s")`, which uses the state of `w`, and one without it, with the same other arguments, such as `read_record("100s")`, which uses `lib()`. The docstrings write this optional first argument `[w::Wfdb]`. The entries of this section show the method that takes `w`, as it is defined; the method without it is defined as `read_record(record::ByteStr; kw...) = read_record(lib(), record; kw...)`. Separate `Wfdb` objects share no state (except what belongs to the process: the count of write errors and the standard streams), so several records can be read at once.

```
w = Wfdb()                              # a library state of its own
wfdbx_set_input_dir(w, "in")            # (the input directory of w only)
mkpath("in"); cp("100s.hea", "in/100s.hea"); cp("100s.dat", "in/100s.dat")
a = read_record(w, "100s")              # reads in/100s.hea, on w
b = read_record("100s")                 # reads ./100s.hea, on lib()
println(a.samples == b.samples)         # true
```

Keyword arguments. The options of a function are keyword arguments, after a semicolon, such as `read_record("100s"; from="0:10", highres=true)`; each has a default, given in the signature. An option whose default is `nothing` means "not given": the default behavior or the current setting.

Record names and directories. A record is named as in the other parts of WFDB-x (Chapters 2 and 3), without the `.hea` extension (the readers, `write_annotations` and `write_record` also accept a trailing `.hea`, which is removed). A record name is a full path, absolute (`"/data/mitdb/100"`) or in the home directory (`"~/Documents/100s"`), used as given; a relative path (`"100s"`, `"data/100s"`), read from the input directory if one is set, otherwise from the current directory, and written to the output directory if one is set, otherwise to the current directory (missing directories are created); or a URL (`"https://physionet.org/files/mitdb/1.0.0/100"`), read only if network access is on (a URL cannot be written). Only `http://` and `https://` URLs are read: a name that begins with another URL scheme (`ftp://`, `file://`, ...) is refused, with or without network access (Section 3.4). Nothing is searched for (legacy mode searches the WFDB path). So a record written under a plain name is written to the output directory and read again from the input directory: the two are the same unless one of them is set, with `wfdbx_set_input_dir` and `wfdbx_set_output_dir` (Section 9.4.3) or with the environment variables `WFDBX_INPUT_DIR` and `WFDBX_OUTPUT_DIR`. The readers take the keyword arguments `input_dir` and `network`, and the writers `dir`: they apply to that one call, and the previous settings are restored when it returns, also when it throws. The `input_dir` option applies to relative names only, not to full paths or URLs. A writer given a name with a path creates the files in that directory, and the header names the record and its signal file without the path (`rec`, `rec.dat`), so that the record opens under any path that leads to it; in legacy mode, as in the original, the header gives the name with its path. With the `dir` option, the name must be a plain name: the files are created in `dir`, and the record opens from that directory under its plain name. The signal files that a header names are read from the header's directory.

Times. The keyword arguments `from` and `to` of `read_record` and `read_annotations` are sample numbers (whole numbers, such as `3600` or `3600.0`) or WFDB time strings, as `strtim` reads them (`"0:10"`, 10 seconds; `"1:02:03.5"`; `"s3600"`, sample 3600; `"e"`, the end of the record; or a time of day in brackets, `"[14:00:00]"`). A time in seconds needs the sampling frequency; when the record does not give it, such a time is an error.

Samples. Samples are `Int32` values (C's `WFDB_Sample`), in ADC units, in a `Matrix{Int32}` with one row per sample and one column per signal: `samples[i, s]` is sample `i` of signal `s` (both counted from 1). Invalid samples are `WFDB_INVALID_SAMPLE` (-32768), and `NaN` in physical units. Sample numbers and times are `Int64`, counted from 0 as in C. Signal numbers are counted from 0 where the API gives them as numbers (`channels`, and the `channels` field of a `Record`), as in C and the tools.

Per-signal and per-annotation values. The per-signal keyword arguments of `write_record` (`gains`, `units`, ...) take one value for all the signals, or a vector with one value per signal; the per-annotation fields of `Annotations` (`symbols`, `chans`, ...) are empty or hold one value per annotation. Any other length is an error, reported before anything is written.

Signal parameters. No sampling frequency, gain, ADC resolution or units is ever assumed. Where one is needed and the record does not give it, it must be given; otherwise the function throws a `WfdbError`. A record whose header gives no sampling frequency has `fs` 0, a signal without an ADC resolution has `adcres` 0, and a signal without units has `units` `nothing`.

Text. Text read from files (signal descriptions, units, info strings, metadata, aux strings, mnemonics) is given as bytes, `Vector{UInt8}`, since WFDB files are not UTF-8: every byte sequence round-

trips unchanged. `String(x)` (or `String.(v)` for a vector) converts bytes for printing; `String(x)` takes the vector it converts, so `String(copy(x))` keeps `x`. Arguments accept `String`s or bytes (the type `ByteStr`, `Union{AbstractString, AbstractVector{UInt8}}`; `OptByteStr` also accepts `nothing`); a `String` is taken as its bytes (UTF-8), as given. The names of records (`Record.name`, `Header.name`, ...) and the time strings are `String`s.

Legacy mode. With `legacy=true`, a function behaves as in the original WFDB package, with its assumed signal parameters and its search for files (Chapter 4); the library's legacy mode is on for the duration of the call only. With `false` (the default), the current setting of `wfdbx_set_legacy` applies, which is off unless it was turned on.

State. The functions that read and write records and annotation files call `wfdbquit()` on the `Wfdb` object they use, before and after their work, also when they throw, which closes anything opened with the WFDB library functions on that object. The library's settings (legacy mode, getvec mode, time resolution of output annotators, input and output directories, network access) are the same after a call as before it, whether it returns or throws: the high-level functions and `check_record` save `getgvmode()` and `getafreq()` when they start and restore them when they return, since opening an EDF record adds `WFDB_HIGHRES` to the getvec mode and sets the time resolution to the record's sampling frequency (when none is set), as in C, and `wfdbquit()` resets neither; the `fs` of `write_annotations` applies to its file only. The C-style functions (`isigopen`, `wfdbquit`, ...) keep the behavior of C. The metadata functions do not call `wfdbquit()`: they open only the header, and leave the library's open files as they are.

The examples of this section run, each in a new module after `using WFDBX`, in a copy of the test records of `tests/data` (among them `100s`), which is the current directory and holds the records that the examples write; the input and output directories are not set. The quick start (Section 9.2) runs instead from the root of the repository.

9.3.2 Reading records

`read_record` — Read the signals of a record

```
read_record(w::Wfdb, record::ByteStr; channels=nothing, from=0, to=nothing,
            highres::Bool=false, pad::Bool=false, input_dir::OptByteStr=nothing,
            network::Union{Nothing,Bool}=nothing, legacy::Bool=false,
            checksum_warning::Bool=true)
```

Reads the signals of a record from `from` up to `to` (exclusive), in ADC units, together with the description of the record. The samples are those that `getvec` returns (Chapter 13), read in large blocks with `getvecs`. If the record ends before `to`, the samples up to its end are returned. When the read reaches the end of the record, the samples are compared with the checksums of the header (Section 9.3.7). The getvec mode (`getgvmode()`) and the time resolution of output annotators (`getafreq()`) are the same after the call as before it: `highres` and `pad` apply to this call only.

`record`

Record name, without the `.hea` extension (a trailing `".hea"` is also accepted): a full path, absolute (`"/data/mitdb/100"`) or in the home directory (`"~/Documents/100s"`), used as given; a relative path (`"100s"`, `"data/100s"`), read from the input directory if one is set, otherwise from the current directory; or a URL (`"https://physionet.org/files/mitdb/1.0.0/100"`), read only if network access is on.

channels
: The signals to read, in the order given: a vector of signal numbers (`0` is the first signal) or names, such as `["V5"]` (as `signal_names` gives them). `nothing`: every signal.

from
: The first sample: a sample number (≥ 0) or a time string. Default 0.

to
: The sample after the last one: a sample number or a time string. `nothing`: the end of the record.

highres
: `true` to read a multi-frequency record at its highest sampling frequency (`fs` of the result is then that frequency); `false`: at its frame frequency, as `getvec` does in low-resolution mode.

pad
: `true` to replace each invalid sample by the previous valid sample of the same signal (getvec mode `WFDB_GVPAD`).

input_dir
: The input directory for this call only (optional): a local directory, a ~ path or a URL, under which relative record names are read. It does not apply to full paths or URLs. `nothing`: the current setting.

network
: `true` to allow reading URLs for this call only, `false` to forbid it. `nothing`: the current setting.

legacy
: `true` to behave as the original WFDB package, for this call only.

checksum_warning
: `true` (the default) to log a warning when the data do not match the checksums of the header; `false`: no warning.

Returns A `Record` (Table 9.1), with the samples in ADC units.

Errors A `WfdbError` if the record cannot be read or has no signals ("can't read record REC"), or its signals cannot be opened ("can't open signals of record REC"); if a requested signal does not exist ("record REC has no signal C"); if `from` is negative ("read_record: from must not be negative"), if `from` or `to` is a number that is not whole ("read_record: from must be a whole number") or beyond the range of a 64-bit integer, or is neither a number nor a string ("read_record: from must be a sample number or a time string"); if `from` cannot be reached ("can't seek to sample N of record REC"); if a time string needs the sampling frequency and the header gives none ("sampling frequency of record REC is not specified"); or if a signal file is shorter than its header says, or cannot be read or decoded ("can't read signal data of record REC (getvec status S)"), also when the checksums do not match either (the C-style `getvec` then returns the checksum error, -4, for the incomplete last frame): an incomplete frame is never returned. Data that do not match the checksums of the header are returned, with a warning (see `checksum_warning`). At most 2^{26} samples are allocated in advance, whatever length the header gives; the result grows as the samples are read.

```
# Seconds 10 to 20 of lead MLII, in physical units.
rec = read_record("100s"; channels=["MLII"], from="0:10", to="0:20")
mlii = physical(rec)[:, 1]
println(length(mlii), " ", rec.fs, " ", rec.sampfrom, " ", String(signal_names(rec)[1]))
# 3600 360.0 3600 MLII

# The first 3 samples of both signals, in ADC units, V5 first.
rec = read_record("100s"; channels=[1, 0], to=3)
println(rec.samples)                                # Int32[1011 995; 1011 995; 1011 995]
println(rec.channels, " ", rec.siginfo[1].gain, " ", rec.siginfo[1].baseline)
# [1, 0] 200.0 1024
```

```
# The whole record: the checksums are verified.
rec = read_record("100s")
println(size(rec.samples, 1), " ", rec.checksum_ok)  # 21600 true
```

A multi-frequency record (here the test record `mfreq`, whose signals have 1, 2 and 4 samples per frame at a frame frequency of 100 Hz) is read at its frame frequency, or at its highest sampling frequency:

```
rec = read_record("mfreq")
println(rec.fs, " ", size(rec.samples))             # 100.0 (600, 3)
rec = read_record("mfreq"; highres=true)
println(rec.fs, " ", size(rec.samples))             # 400.0 (2400, 3)
println([si.spf for si in rec.siginfo])             # Int32[1, 2, 4]
```

---

read_header

Read the header of a record

```
read_header(w::Wfdb, record::ByteStr; input_dir::OptByteStr=nothing,
            network::Union{Nothing,Bool}=nothing, legacy::Bool=false)
```

Reads the header of a record, without opening its signal files. The result describes the record and each of its signals, and holds the info strings and the metadata entries of the header (from the `.hea` file only, as `read_metadata` reads them, with its warning for a key given more than once). For an EDF record, which has no `.hea` file, and for a header read from the standard input (`"-"`), the metadata is empty, without an error. For a multi-segment record, the master header is read, and `nsamples` is the length of the whole record.

record
: Record name, without the `.hea` extension (a trailing `".hea"` is also accepted): a full path, absolute (`"/data/mitdb/100"`) or in the home directory (`"~/Documents/100s"`), used as given; a relative path (`"100s"`, `"data/100s"`), read from the input directory if one is set, otherwise from the current directory; or a URL (`"https://physionet.org/files/mitdb/1.0.0/100"`), read only if network access is on.

input_dir
: The input directory for this call only (optional), as for `read_record`: relative record names are read under it. It does not apply to full paths or URLs.

network
: Network access for this call only, as for `read_record`.

legacy
: `true` to read the header as the original WFDB package does, for this call only.

Returns A `Header` (Table 9.2).

Errors A `WfdbError` if the header cannot be read (“can’t read header for record REC”).

```
hdr = read_header("100s")
println(hdr.name, " ", hdr.fs, " ", hdr.nsig, " ", hdr.nsamples)   # 100s 360.0 2 21600
for si in hdr.siginfo
    println(String(si.desc), " ", String(si.fname), " ", si.fmt, " ", si.gain, " ",
            si.adcres, " ", si.units)
end
# MLII 100s.dat 212 200.0 11 nothing
```

```
# V5 100s.dat 212 200.0 11 nothing
println("[", String(hdr.comments[1]), "]")          # [ 69 M 1085 1629 x1]
println(hdr.base_time === nothing)                   # true: the header gives no start time
key, value = hdr.metadata[1]
println("[", String(key), "] [", String(value), "]")
# [Produced by xform from record 100, beginning at 0] [0]

# A copy of the header in the directory "in", read for this call only.
mkpath("in"); cp("100s.hea", "in/100s.hea")
println(read_header("100s"; input_dir="in").nsig)   # 2
```

---

read_annotations

Read an annotation file

```
read_annotations(w::Wfdb, record::ByteStr, annotator::ByteStr; from=nothing,
                 to=nothing, fs::Union{Nothing,Real}=nothing,
                 input_dir::OptByteStr=nothing, network::Union{Nothing,Bool}=nothing,
                 legacy::Bool=false)
```

Reads the annotations of an annotation file, in file order, as parallel vectors. The sampling frequency, needed to convert times in seconds, is that of the record's header, or, for a record without a header or whose header gives none, the keyword argument `fs`, as the option `-F` of `rdann`.

record
: Record name, without the `.hea` extension (a trailing `".hea"` is also accepted): a full path, absolute (`"/data/mitdb/100"`) or in the home directory (`"~/Documents/100s"`), used as given; a relative path (`"100s"`, `"data/100s"`), read from the input directory if one is set, otherwise from the current directory; or a URL (`"https://physionet.org/files/mitdb/1.0.0/100"`), read only if network access is on. Its header, if there is one, gives the sampling frequency.

annotator
: Annotator name: the suffix of the annotation file. `"atr"` reads the file `RECORD.atr`.

from
: The first time to read: a sample number (a whole number ≥ 0, such as `100` or `100.0`) or a time string. `nothing`: the beginning of the file.

to
: The time at which to stop: a sample number (a whole number) or a time string. Annotations at or after it are not read. `nothing`: the end of the file.

fs
: The sampling frequency, in Hz (> 0), to use when the record has no header or its header gives none. `nothing`: none.

input_dir
: The input directory for this call only (optional), as for `read_record`: relative record names are read under it. It does not apply to full paths or URLs.

network
: Network access for this call only, as for `read_record`.

legacy
: `true` to behave as the original WFDB package (which assumes 250 Hz for a record without a frequency), for this call only.

Returns An `Annotations` (Table 9.3). Its `fs` is the frequency in which the times are counted: the one written in the annotation file, if there is one, or else the sampling frequency; it is 0 if none of them is known.

Errors A `WfdbError`, before any file is opened, if `from` is negative ("read_annotations: from must not be negative"), if `from` or `to` is a number that is not whole ("read_annotations: from must be a whole number") or beyond the range of a 64-bit integer, or is neither a number nor a string ("read_annotations: from must be a sample number or a time string"), or if `fs` is given and is not > 0 ("read_annotations: fs must be > 0"); if the annotation file cannot be opened ("can't read annotator ANN for record REC"); if a time string needs the sampling frequency and none is known ("sampling frequency of record REC is not specified"); if `from` cannot be reached ("can't seek in annotator ANN"); or if the file ends early or cannot be read before `to` is reached ("can't read annotation file ANN of record REC (getann status S)", where S is the status that `getann` returned, such as -3 for an unexpected end of file). Whatever happens, the annotation file is closed (`wfdbquit`) before the function returns.

```
ann = read_annotations("100s", "atr"; from="0:10", to="0:30")
println(length(ann), " ", ann.fs)                   # 24 360.0
println(ann.samples[1], " ", String(ann.symbols[1])) # 3862 N

# The whole file: the first annotation is a rhythm change, with an aux string.
ann = read_annotations("100s", "atr")
println(String(ann.symbols[1]), " ", ann.codes[1] == WFDBX.RHYTHM, " ", String(ann.aux[1]))
# + true (N

# A record without a header: fs gives the frequency of times in seconds (the library
# reports on the standard error that it cannot open the header).
write_annotations("beats", "qrs", Annotations(samples=[100, 460, 820],
                                              symbols=[b"N", b"N", b"N"]))
ann = read_annotations("beats", "qrs"; to="0:02", fs=360)
println(length(ann), " ", ann.fs)                   # 2 360.0
```

---

physical

Convert the samples of a record to physical units

```
physical(rec::Record; legacy::Bool=false)
```

Returns the samples of a `Record` in physical units, computed as $(adc - baseline)/gain$ for each signal, with the gain and baseline of its `siginfo`. Invalid samples become `NaN`. The result has the shape of `rec.samples`. It uses no library state.

rec The record, as `read_record` returns it.

legacy
: `true` to behave as the original WFDB package, which converts the samples of an uncalibrated signal (gain 0) with a gain of 200. With `false`, the legacy mode of the default library object (`wfdbx_set_legacy`) decides.

Returns A `Matrix{Float64}`, one row per sample and one column per signal.

Errors A `WfdbError` if the gain of a signal is 0 ("signal N of record REC is uncalibrated", where N is the number of the signal in the record, counted from 0), except in legacy mode.

```
rec = read_record("100s"; to=2)
println(physical(rec))          # [-0.145 -0.065; -0.145 -0.065]
```

---

signal_names

Return the signal descriptions of a record

```
signal_names(rec::Record)
```

Returns the description of each signal of a `Record`, such as `"MLII"`, as bytes, without the white space that a header may put at either end of it (empty for a signal without a description). These are the names that `channels` of `read_record` accepts. It uses no library state.

`rec` The record, as `read_record` returns it.

Returns A `Vector{Vector{UInt8}}`, one element per signal of `rec`.

Errors Nothing.

```
rec = read_record("100s"; to=1)
println(String.(signal_names(rec)))     # ["MLII", "V5"]
```

9.3.3 Writing records

`write_record` Write a header and a signal file

```
write_record(w::Wfdb, record::ByteStr, samples::AbstractMatrix{<:Integer}, fs::Real;
             fmt::Integer=16, gains=nothing, baselines=nothing, adczeros=0,
             adcres=nothing, units=nothing, names=nothing, samps_per_frame=nothing,
             base_time=nothing, comments=(), metadata=nothing, dir=nothing,
             legacy::Bool=false, invalid_warning::Bool=true)
```

Writes samples in ADC units as a new record: the header `RECORD.hea` and one signal file, `RECORD.dat`, which holds all the signals (so they share one format). Existing files of these names are replaced. The sampling frequency, the gain and ADC resolution of each signal, and the units of each calibrated signal must be given: there are no defaults. Every check except those of the writing itself comes before any file is touched, so an invalid argument writes nothing. The files are written as `putvec` and `newheader` write them, followed by the info strings (`putinfo`), to temporary files in the directories of the destinations (`.NAME.XXXXXXXX.tmp`), which replace the existing files only when both are complete: a failure (a full disk, for example) leaves the existing files as they were, and no temporary file is left. The files replaced keep their permissions; for a symbolic link, the file it leads to is replaced; a destination that is not a regular file (a device) is written directly. Text (the record name, `names`, `units`, `comments`) is written as its bytes: a `String` as its UTF-8 bytes.

`record`
: Record name, without the `.hea` extension (a trailing `".hea"` is removed, as by the readers): a plain name (`"rec"`) or a relative path (`"data/rec"`), created in the output directory if one is set, otherwise in the current directory, missing directories included; or a full path, absolute or in the home directory (`"~/Documents/rec"`), used as given. With a path, the files are created in that directory, and the header names the record and its signal file without the path (`"rec"`, `"rec.dat"`), so that the record opens under any path that leads to it (in legacy mode, as in the original, the header gives the name with its path). A URL cannot be written. It must not be empty.

`samples`
: The samples in ADC units, one row per sample (per frame, with `samps_per_frame`) and one column per signal: integers within the range of `Int32`. `WFDB_INVALID_SAMPLE` is written as an invalid sample.

`fs` The sampling frequency in Hz: finite and > 0 (the frame frequency of a multi-frequency record).

fmt The storage format of all the signals, such as 16, 212, 24 or 516 (FLAC) (Chapter 15). Every sample must fit the format: within its range (Table 15.1; -2048 to 2047 for format 212, for example), and, in format 8, within -128 to 127 of the sample before it. The smallest value of a format is the value of an invalid (missing) sample: a sample equal to it, or to `WFDB_INVALID_SAMPLE`, is read back as invalid (see `invalid_warning`). A FLAC file holds at most 8 signals. Default 16.

gains
The gain of each signal, in ADC units per physical unit (required): finite; 0 marks an uncalibrated signal, and negative gains are allowed.

baselines
The ADC value of 0 physical units of each signal, a whole number within the range of `Int32` (an integer, or a finite float with a whole value). `nothing`: the signal's `adczero`.

adczeros
The ADC value of 0 V of each signal, a whole number within the range of `Int32`. Default 0.

adcres
The ADC resolution of each signal, in bits: a whole number, at least 1 (required).

units
The physical units of each signal, such as `"mV"` (required for each calibrated signal; `nothing` for an uncalibrated one).

names
The description of each signal. `nothing`: none.

samps_per_frame
The number of samples per frame of each signal, integers ≥ 1, for a multi-frequency record: each row of `samples` then holds `samps_per_frame[s]` consecutive columns for signal `s`. `nothing`: 1 for every signal.

base_time
The start time and date, as `"hh:mm:ss dd/mm/yyyy"` (or a part of it, as `setbasetime` reads it). `nothing`: none.

comments
The info strings to write at the end of the header, without their `#`. Default: none.

metadata
`Key: value` entries, written after the comments as `# Key: value` (`# Key:` for an empty value): a vector of `key => value` pairs (or `(key, value)` tuples), written in that order, or a dictionary. Keys and values follow the rules of `set_metadata`; no two keys may differ only in letter case, none may be the key of an entry in `comments`, and a value must not be `nothing`. `nothing`: none.

dir The directory (which must exist) in which to create the files, for this call only (the output directory of the call). `record` must then be a plain name, without a directory, and the record opens from that directory under its plain name. `nothing`: the output directory.

legacy
`true` to behave as the original WFDB package, which assumes a gain of 200, an ADC resolution based on the format and no units when they are not given, for this call only.

invalid_warning
`true` to log a warning (`@warn`) when samples will be read back as invalid (missing) samples ("N sample(s) of signal K will be read back as invalid (missing) samples (format F stores an invalid sample as V)"): those equal to the invalid value of the format or to `WFDB_INVALID_SAMPLE`, other than the `NaN` samples of `write_record_physical`, which mark missing samples deliberately. `false` for no warning, when such samples are deliberate. Default `true`.

Returns `nothing`.

Errors A `WfdbError`, before any file is touched, if `record` is empty ("write_record: the record name must not be empty"); if there are no signals ("write_record: at least one signal is required"); if `gains`, `adcres` or `units` is missing ("write_record: gains must be specified", "write_record: ADC resolution(s) must be specified", "write_record: units must be specified"); if an ADC resolution, an ADC zero or a baseline is not a whole number within the range of `Int32` ("write_record: adcres must be whole numbers within the range of a 32-bit integer", and the same for `adczeros` and `baselines`; also in legacy mode), or an ADC resolution is less than 1 ("write_record: ADC resolution(s) must be specified"); if a `samps_per_frame` value is not an integer ≥ 1 ("write_record: samps_per_frame must hold integers >= 1"), or the number of columns of `samples` is not their sum ("write_record: signals has N columns, but samps_per_frame adds up to M"); if `fs` is not finite and > 0; if a gain is not finite ("write_record: gains must be finite"); if a vector of per-signal values has neither one value nor one per signal ("write_record: OPTION must have one value per signal"); if the number of columns does not match the signals; if a sample lies outside the range of `Int32`; if a sample does not fit the format ("write_record: signal K has a sample (V) outside the range of format F (LO to HI)"); in format 8, "write_record: signal K changes by D between two samples, more than format 8 can store (-128 to 127)" or "write_record: signal K has an invalid sample, which format 8 cannot store"); if a FLAC format is given for more than 8 signals ("write_record: format 516 (FLAC) stores at most 8 signals in a file; the record has 12"); if `base_time` is not a valid time ("write_record: incorrect base time 'T'"); if an entry of `metadata` is not valid ("write_record: metadata key "a:b" must not contain ':'", ...), or its key is also that of an entry in `comments`; or if `dir` does not exist ("directory D does not exist") or `record` contains a directory while `dir` is given. A `WfdbError` also if a file cannot be created ("can't create signal file for record REC", "can't create header for record REC"), or cannot be written completely, for example on a full disk ("write error on record REC").

```
x = Int32[0 100; 10 110; 20 120; 30 130]            # 4 samples of 2 signals
write_record("two", x, 250; gains=[100, 200], adcres=12, units="mV",
             names=["a", "b"], base_time="12:00:00 01/01/2020",
             comments=["recorded at rest"], metadata=["Age" => "63"])
print(read("two.hea", String))
# two 2 250 4 12:00:00 01/01/2020
# two.dat 16 100/mV 12 0 0 60 0 a
# two.dat 16 200/mV 12 0 100 460 0 b
# #recorded at rest
# # Age: 63
println(read_record("two").samples == x)            # true

# A multi-frequency record: signal 1 has 2 samples per frame.
y = Int32[1 2 3; 4 5 6]                             # 2 frames: a, then b twice
write_record("mf", y, 100; gains=200, adcres=12, units="mV", samps_per_frame=[1, 2])
println(size(read_record("mf"; highres=true).samples))   # (4, 2)
```

`write_record_physical` — Write a record from samples in physical units

```
write_record_physical(w::Wfdb, record::ByteStr, samples::AbstractMatrix{<:Real}, fs::Real;
                      fmt::Integer=16, gains=nothing, baselines=nothing,
                      adczeros=0, adcres=nothing, units=nothing, names=nothing,
                      samps_per_frame=nothing, base_time=nothing, comments=(),
                      metadata=nothing, dir=nothing, legacy::Bool=false,
                      invalid_warning::Bool=true)
```

Writes a record as `write_record` does, from samples in physical units. Each value `v` is stored as $\mathrm{round}(v \cdot gain) + baseline$, with halfway cases rounded away from zero; `NaN` is stored as an invalid sample. The gain of every signal must be nonzero.

`record`
: Record name, as for `write_record`.

`samples`
: The samples in physical units, one row per sample (or frame) and one column per signal: real numbers; `NaN` for an invalid sample.

`fs` The sampling frequency in Hz, as for `write_record`.

`fmt` The storage format, as for `write_record`. Default 16.

`gains`
: The gain of each signal (required), finite and nonzero.

`baselines`
: The baseline of each signal, as for `write_record`.

`adczeros`
: The ADC zero of each signal, as for `write_record`.

`adcres`
: The ADC resolution of each signal (required), as for `write_record`.

`units`
: The physical units of each signal (required), as for `write_record`.

`names`
: The signal descriptions, as for `write_record`.

`samps_per_frame`
: The samples per frame of each signal, as for `write_record`.

`base_time`
: The start time and date, as for `write_record`.

`comments`
: The info strings, as for `write_record`.

`metadata`
: The metadata entries, as for `write_record`.

`dir` The directory in which to write the files, as for `write_record`.

`legacy`
: `true` to behave as the original WFDB package (a gain of 0 is then taken as 200), for this call only.

`invalid_warning`
: `true` to log a warning for the samples that will be read back as invalid, as for `write_record`; the `NaN` samples are not counted. Default `true`.

Returns `nothing`.

Errors A `WfdbError` in the cases of `write_record`; if a gain is 0 ("write_record: signal N is uncalibrated", N counted from 0); if a value is infinite, or its ADC value lies outside the range of `Int32` ("write_record: signal N has a sample that is infinite or beyond the range of 32-bit ADC units", N counting the columns from 0); or if an ADC value does not fit the format (in format 8, a `NaN` too: "write_record: signal 0 has an invalid sample, which format 8 cannot store"). Nothing is written then.

```
x = [0.0 1.0; 0.5 -1.0; NaN 0.25]                  # in mV; NaN: an invalid sample
write_record_physical("phys", x, 100; gains=200, adcres=12, units="mV", names=["a", "b"])
```

```
rec = read_record("phys")
println(rec.samples)            # Int32[0 200; 100 -200; -32768 50]
println(physical(rec))          # [0.0 1.0; 0.5 -1.0; NaN 0.25]
```

---

write_annotations

Write an annotation file

```
write_annotations(w::Wfdb, record::ByteStr, annotator::ByteStr, ann::Annotations;
                  fs::Real=0.0, sort::Union{Nothing,Bool}=nothing, dir=nothing,
                  legacy::Bool=false)
```

Writes the annotations of an `Annotations` object to the annotation file `RECORD.ANNOTATOR`, in the output directory or in `dir`, as `putann` writes them, to a temporary file in the directory of the destination, which replaces an existing file only when it is complete (and, if needed, sorted): a failure leaves the existing file as it was. The type of each annotation is its mnemonic in `ann.symbols` if that is not empty, and its code in `ann.codes` otherwise; both may be given (as `read_annotations` returns them) if they agree. As in C, the subtype, channel and annotator number are written in one byte each (other values wrap around), and auxiliary text longer than 255 bytes is truncated.

record
: Record name, without the `.hea` extension (a trailing `".hea"` is also accepted): the record of the annotations (its header is not needed). A plain name or a relative path (`"data/100s"`) is created in the output directory if one is set, otherwise in the current directory; a full path, absolute or in the home directory (`"~/Documents/100s"`), is used as given. The file is `RECORD.ANNOTATOR` at that place (`data/100s.qrs`). A URL cannot be written.

annotator
: Annotator name: the suffix of the file, such as `"qrs"`.

ann
: The annotations (Table 9.3): `samples` (the times), and `symbols` or `codes`; `subtypes`, `chans`, `nums` and `aux` are optional. Each field is empty or has one element per annotation. `ann.record`, `ann.annotator` and `ann.fs` are not used.

fs
: 0 (the default), or a finite frequency > 0, which is written in the file as the frequency in which the annotation times are counted. With 0, readers use the record's sampling frequency. It applies to this file only: the time resolution of output annotators (`getafreq()`) is left as it was.

sort
: If the annotations are not in time order, the file is sorted when it is written: `true` always sorts, and `false` keeps the given order (without the library's advice to run `sortann`), for this call only. `nothing`: the environment variable `WFDBANNSORT` decides (sorting is on unless it is `0`).

dir
: The directory (which must exist) in which to create the file, for this call only. `record` must then be a plain name, without a directory: the file is `DIR/RECORD.ANNOTATOR`.

legacy
: `true` to behave as the original WFDB package, for this call only.

Returns `nothing`.

Errors A `WfdbError`, before the file is touched, if a field of `ann` is neither empty nor has one element per annotation ("write_annotations: FIELD must have one entry per annotation (N), or none"); if there is at least one annotation and both `ann.symbols` and `ann.codes` are empty ("write_annotations: symbols or codes must be given"; with no annotations, an empty file is

written); if a time is `typemin(Int64)` or `typemax(Int64)`, which the file format reserves ("write_annotations: annotation time T cannot be written (annotation times must be integers from -9223372036854775807 to 9223372036854775806)"); if a code is not from 0 to 58 ("write_annotations: annotation code C is reserved (59 to 63 are the control codes of the annotation file format)" for 59 to 63, "write_annotations: annotation code C is out of range (0 to 58)" otherwise); if `ann.symbols` and `ann.codes` are both given and disagree ("write_annotations: symbols and codes disagree at annotation K", K counted from 0); if `fs` is neither 0 nor a finite frequency > 0; or if `dir` does not exist or `record` contains a directory while `dir` is given. A `WfdbError` also if the file cannot be created ("can't create annotator ANN for record REC") or cannot be written completely, or `putann` reports an error ("write error on annotator ANN for record REC").

```
ann = Annotations(samples=[100, 460, 820], symbols=[b"N", b"N", b"V"])
write_annotations("100s", "qrs", ann; fs=360)       # writes 100s.qrs
a = read_annotations("100s", "qrs")
println(a.samples, " ", String.(a.symbols))         # [100, 460, 820] ["N", "N", "V"]
println(a.codes == [WFDBX.NORMAL, WFDBX.NORMAL, WFDBX.PVC])   # true

# Out of order: sorted when the file is written (the library prints a message
# "Rearranging annotations for output annotator srt ...done!").
write_annotations("100s", "srt", Annotations(samples=[460, 100], codes=Int32[1, 1]); sort=true)
println(read_annotations("100s", "srt").samples)    # [100, 460]
```

9.3.4 Checking records

---

check_record — Check a record and its annotation files

```
check_record(w::Wfdb, record::ByteStr; annotators=String[],
             input_dir::OptByteStr=nothing, network::Union{Nothing,Bool}=nothing)
```

Checks the header of a record, its signal specifications, every sample of its signal files, each segment of a multi-segment record, and the annotation files named, and reports each problem found. A problem in the record is never an error of the function: it is reported, with a severity (`"error"`: the record is damaged or does not follow the WFDB format, and programs will read wrong data or fail; `"warning"`: the record can be read, but something is missing or unusual, and may give wrong results; `"info"`: a fact worth knowing, such as invalid samples). The problems, their codes and their messages are the same in the seven languages (Table 6.7). The checks run in normal mode, even if legacy mode is on, and the library prints no messages during the call (the previous settings are restored afterwards). They are similar to those of Benjamin Moody's `wfdbcheck`.

record
: Record name, without the `.hea` extension (a trailing `".hea"` is also accepted): a full path, absolute (`"/data/mitdb/100"`) or in the home directory (`"~/Documents/100s"`), used as given; a relative path (`"100s"`, `"data/100s"`), read from the input directory if one is set, otherwise from the current directory; or a URL (`"https://physionet.org/files/mitdb/1.0.0/100"`), read only if network access is on.

annotators
: The annotator names whose files are checked, such as `["atr"]` (one name may be given as a string). Default: none.

input_dir
: The input directory for this call only (optional), as for `read_record`: relative record names are read under it. It does not apply to full paths or URLs.

network
: Network access for this call only, as for `read_record`.

Returns A `CheckReport` (Table 9.4): `report.ok` is `true` if no problem is an error, `report.issues` lists the problems (`Issue` objects, Table 9.5), and `lines(report)` (or `print(report)`) gives one line per problem, or `no problems found`.

Errors An `ArgumentError` if `record` is empty ("record must not be empty") or an annotator name is not a string ("annotator names must be strings"). A record that cannot be read is reported (code `header_unreadable`), not thrown as an error.

```
report = check_record("100s"; annotators=["atr"])
println(report.ok)                                  # true: no errors
println(report)
# warning: signal 0: the units are not specified
# warning: signal 1: the units are not specified

# A damaged record: one sample of a 12-bit signal is overwritten with 2048.
write_record("damaged", reshape(Int32.(0:99), :, 1), 100; gains=100, adcres=12,
             units="mV", names=["ramp"])
dat = read("damaged.dat")
dat[21] = 0x00; dat[22] = 0x08      # sample 10 (2 bytes per sample in format 16): 2048
write("damaged.dat", dat)
report = check_record("damaged")
println(report.ok)                                  # false
for i in report.issues
    println(i.severity, " ", i.code, " ", i.where)
end
# error sample_range signal 0
# error checksum signal 0
println(lines(report)[1])
# error: signal 0: 1 sample is outside the ADC range (-2048 to 2047): sample 10 (value 2048)
```

---

lines

The text form of a check report

```
lines(r::CheckReport)
```

Returns the lines of a report of `check_record`: one line per issue, in the order of the issues (`"SEVERITY: WHERE: MESSAGE"`, without `"WHERE: "` for the record itself), or the one line `"no problems found"`. `print(report)` and `show(report)` write these lines, joined by line feeds. The lines are the same in the seven languages.

r The report, as `check_record` returns it.

Returns A `Vector{String}`, with at least one element.

Errors Nothing.

```
for l in lines(check_record("100s"))
    println(l)
end
```

```
# warning: signal 0: the units are not specified
# warning: signal 1: the units are not specified
x = zeros(Int32, 10, 1)
write_record("clean", x, 100; gains=200, adcres=12, units="mV", names=["a"])
println(lines(check_record("clean")))              # ["no problems found"]
```

9.3.5 Header metadata

These functions read and change the metadata of a header: its info strings of the form `# Key: value`, such as `# Age: 63` (Chapter 5). In short: keys are compared without regard to letter case (A-Z and a-z only); a key must not be empty and must not contain `:`, `#`, CR, LF or NUL, and a value must not contain CR, LF or NUL; spaces, tabs, VT, FF and CR at either end of keys and values are removed; keys and values are bytes (arguments `ByteStr`: a `String` is taken as its bytes; results `Vector{UInt8}`). A change rewrites the header in one step, through a temporary file in the directory of the destination, which then replaces it, keeping its permissions and every other byte of it; the temporary file is removed on any failure. All the arguments are checked first, so an error leaves the header unchanged, and nothing is written if nothing changes (unless `dir` is given). Only the `.hea` file is read (not a `.info` file); for a multi-segment record, it is the master header, unless a segment is named. The header is opened as the library opens a header (`wfdb_open`), so in legacy mode it is searched for in the WFDB path. The entries are returned as `MetadataEntry` pairs:

```
const MetadataEntry = Pair{Vector{UInt8},Vector{UInt8}}     # key => value
```

The functions take the keyword arguments `input_dir` and `network` for reading the header, and the functions that change it also `dir`, the directory in which to write it. `dir nothing` or `""` changes the header that was read. These functions do not call `wfdbquit()`. Invalid arguments throw an `ArgumentError`, and a header that cannot be read or written a `WfdbError` (Section 9.3.7). Metadata often holds information about patients: WFDB-x stores it as plain text, exactly as given, and does not de-identify it.

> **Patient data** A header can hold identifying data about a patient (a name, a date of birth, a medical record number). The metadata functions read and write such data as given. Records that are shared must be de-identified first.

`read_metadata` — Read the metadata of a record

```
read_metadata(w::Wfdb, record::ByteStr; input_dir::OptByteStr=nothing,
              network::Union{Nothing,Bool}=nothing)
```

Reads the metadata entries of the header of a record, in file order. The info strings are the lines that begin with `#` after the record line, which is found as the header parser finds it, in every mode: the first line that holds something other than spaces, tabs and line endings and does not begin with `#` (an empty or blank line before it, and the comments before it, are not read). An info string is an entry if it contains a colon: the key is the text before the first colon, the value the rest of the line, both without spaces at either end. Info strings without a colon, or whose key is empty or contains `#`, are free comments, which are not returned.

`record`

Record name, without the `.hea` extension (a trailing `".hea"` is also accepted): a full path, absolute (`"/data/mitdb/100"`) or in the home directory (`"~/Documents/100s"`), used as given; a relative path (`"100s"`, `"data/100s"`), read from the input directory if one is set, otherwise

from the current directory; or a URL (`"https://physionet.org/files/mitdb/1.0.0/100"`), read only if network access is on.

input_dir
The input directory for this call only (optional), as for `read_record`: relative record names are read under it. It does not apply to full paths or URLs.

network
Network access for this call only, as for `read_record`.

Returns The entries, as `key => value` pairs of bytes, in file order. A key that appears more than once (ignoring letter case) is returned once, with its first value, and a warning is logged (Section 9.3.7).

Errors A `WfdbError` if the header cannot be found or read ("can't read header for record REC"); an `ArgumentError` if `record` is empty ("record name must not be empty") or `"-"` ("the metadata of a header read from the standard input cannot be read or changed").

```
write_record("pat", zeros(Int32, 360, 1), 360; gains=200, adcres=12, units="mV",
             names=["ECG"], comments=["recorded at rest"],
             metadata=["Age" => "63", "Sex" => "F"])
for (key, value) in read_metadata("pat")
    println(String(key), " = ", String(value))
end
# Age = 63
# Sex = F
for c in read_header("pat").comments
    println("[", String(c), "]")
end
# [recorded at rest]: the free comment is not an entry
# [ Age: 63]
# [ Sex: F]
mkpath("in"); cp("100s.hea", "in/100s.hea")
md = read_metadata("100s"; input_dir="in")          # reads in/100s.hea
println(length(md), " ", String(md[1].second))      # 1 0
```

---

get_metadata

Return the value of a metadata entry

```
get_metadata(w::Wfdb, record::ByteStr, key::ByteStr; input_dir::OptByteStr=nothing,
             network::Union{Nothing,Bool}=nothing)
```

Returns the value of one metadata entry of a record. Letter case is ignored, and the spaces at either end of `key` are removed: `"age"` finds `Age`. If the header has more than one entry for `key`, the first is returned; as in `read_metadata`, a warning is logged for each key of the header that has more than one entry.

record
Record name, as for `read_metadata`.

key The key, such as `"Age"` (see `set_metadata` for the valid keys).

input_dir
The input directory for this call only (optional), as for `read_record`: relative record names are read under it. It does not apply to full paths or URLs.

network
Network access for this call only, as for `read_record`.

Returns The value, as bytes (an empty vector for an entry with an empty value), or `nothing` if the header has no entry for `key`.

Errors An `ArgumentError` if `key` is not a valid key (checked before the header is read), or as `read_metadata`.

```
write_record("pat", zeros(Int32, 10, 1), 100; gains=200, adcres=12, units="mV",
             metadata=["Age" => "63", "Notes" => ""])
println(String(get_metadata("pat", "age")))         # 63
println(get_metadata("pat", "Notes"))               # UInt8[]
println(get_metadata("pat", "Weight") === nothing)  # true
```

---

set_metadata

Set the value of a metadata entry

```
set_metadata(w::Wfdb, record::ByteStr, key::ByteStr, value::ByteStr;
             input_dir::OptByteStr=nothing, network::Union{Nothing,Bool}=nothing,
             dir::OptByteStr=nothing)
```

Sets the value of a metadata entry in the header of a record. If the header has an entry for `key` (ignoring letter case), its line is replaced by `# Key: value` (`# Key:` if the value is empty), at the same position and with the same line ending, with the spelling of `key` given here. Otherwise the entry is added after the last info string of the header (or, if it has none, after its last line that is not blank), ending as the record line does. Every other byte of the header is kept.

`record`
: Record name, as for `read_metadata`. The header is changed where it was read, unless `dir` is given; a header read from a URL can only be written elsewhere, with `dir`.

`key` The key, such as `"Age"` or `"Date of birth"`. Spaces at either end are removed. It must not be empty, and must not contain `:`, `#`, a line break or a NUL byte.

`value`
: The value, which may be a sentence. Spaces at either end are removed. It must not contain a line break or a NUL byte; it may be empty.

`input_dir`
: The input directory for reading the header, for this call only.

`network`
: Network access for reading the header, for this call only.

`dir` An existing directory in which to write the changed header, under the same file name, instead of changing the header itself (the signal files are not copied). `nothing` or `""`: change the header that was read.

Returns `nothing`.

Errors An `ArgumentError` if `record` is empty or `"-"`, if `key` or `value` is not valid ("metadata key "a:b" must not contain ':'", "value of metadata key "Age" must not contain a line break", ...), if the header has more than one entry for `key` ("header of record REC has more than one entry for key KEY (lines N, M); remove them first"), if it has no record line, or if it was read from a URL and `dir` is not given. A `WfdbError` if the header cannot be read ("can't read header for record REC"), if the changed header cannot be written ("can't write header for record REC"), or if `dir` does not exist ("directory D does not exist").

```
write_record("pat", zeros(Int32, 10, 1), 100; gains=200, adcres=12, units="mV",
             metadata=["Age" => "63"])
```

```
set_metadata("pat", "age", "64")                    # replaces the line "# Age: 63"
set_metadata("pat", "Date of birth", "1960-01-01")  # adds a line
for l in eachline("pat.hea")
    startswith(l, "#") && println(l)
end
# # age: 64
# # Date of birth: 1960-01-01
mkpath("copy")
set_metadata("pat", "Age", "65"; dir="copy")        # writes copy/pat.hea
println(String(get_metadata("copy/pat", "Age")), " ", String(get_metadata("pat", "Age")))
# 65 64
```

---

remove_metadata

Remove a metadata entry

```
remove_metadata(w::Wfdb, record::ByteStr, key::ByteStr; input_dir::OptByteStr=nothing,
                network::Union{Nothing,Bool}=nothing, dir::OptByteStr=nothing)
```

Removes a metadata entry from the header of a record. Every entry for `key` (ignoring letter case) is removed, with its line ending, so this also repairs a header that has more than one. Every other byte of the header is kept. Nothing is written if the header has no entry for `key` (unless `dir` is given).

record
: Record name, as for `set_metadata`.

key
: The key, as for `set_metadata`.

input_dir
: The input directory for reading the header, for this call only.

network
: Network access for reading the header, for this call only.

dir
: The directory in which to write the changed header, as for `set_metadata`.

Returns The number of lines removed (0 if the header has no entry for `key`).

Errors As `set_metadata` (a key that the header has more than once is not an error here).

```
write_record("pat", zeros(Int32, 10, 1), 100; gains=200, adcres=12, units="mV",
             metadata=["Age" => "63", "Sex" => "F"])
println(remove_metadata("pat", "SEX"))              # 1
println(remove_metadata("pat", "Weight"))           # 0: nothing is written
println(length(read_metadata("pat")))               # 1
```

---

update_metadata

Make several changes to the metadata

```
update_metadata(w::Wfdb, record::ByteStr, changes; input_dir::OptByteStr=nothing,
                network::Union{Nothing,Bool}=nothing, dir::OptByteStr=nothing)
```

Makes several changes to the metadata of a record in one write. Each change is applied in order, as `set_metadata` does, or, if its value is `nothing`, as `remove_metadata` does. All the changes are checked first: if one is not valid, none is made.

record
Record name, as for `set_metadata`.

changes
A vector (or tuple) of `key => value` pairs (or `(key, value)` tuples), or a dictionary; a value of `nothing` removes the entry. No two keys may differ only in letter case. The changes of a `Dict` are applied in its iteration order, which is not defined: a vector fixes the order in which new entries are added.

input_dir
The input directory for reading the header, for this call only.

network
Network access for reading the header, for this call only.

dir The directory in which to write the changed header, as for `set_metadata`.

Returns `nothing`.

Errors As `set_metadata`; an `ArgumentError` also if `changes` is not a vector, a tuple or a dictionary of pairs ("metadata must be a dictionary, or a vector of key => value pairs"), if a value is neither a string nor `nothing`, or if a key is given more than once ("metadata key "AGE" is given more than once (keys are compared without regard to letter case)").

```
write_record("pat", zeros(Int32, 10, 1), 100; gains=200, adcres=12, units="mV",
             metadata=["Age" => "63", "Dx" => "164889003"])
update_metadata("pat", ["Age" => "64", "Dx" => nothing, "Sex" => "M"])
println([String(k) => String(v) for (k, v) in read_metadata("pat")])
# ["Age" => "64", "Sex" => "M"]
```

9.3.6 The result types

The readers return objects of the types `Record`, `Header` and `Annotations`, whose fields are listed in Tables 9.1 to 9.3; `check_record` returns a `CheckReport`, which holds `Issue` objects (Tables 9.4 and 9.5). The signals of a record are described by `Siginfo` structures (Table 9.6), which are those of the WFDB library (Section 9.4). `Record`, `Header`, `CheckReport` and `Issue` are immutable structures (the vectors and matrices they hold can be changed in place); `Annotations` and `Siginfo` are mutable, with keyword constructors whose defaults are empty vectors or C's zero values, such as `Annotations(samples=[100], symbols=[b"N"])`. `length(ann)` is the number of annotations.

Table 9.1 The fields of `Record`, the result of `read_record`

Field	Contents
`name::String`	the record name, as given to `read_record`
`fs::Float64`	the sampling frequency of the returned samples, in Hz, or 0 if the header gives none; with `highres`, the highest sampling frequency of a multi-frequency record
`samples::Matrix{Int32}`	the samples in ADC units, one row per sample and one column per signal returned; invalid samples are `WFDB_INVALID_SAMPLE`
`siginfo::Vector{Siginfo}`	one per signal returned, in the order of the columns; `desc` has no white space at either end
`base_time::Union{Nothing,String}`	the start time and date as text, to the second, such as `"[14:00:03 21/06/2024]"`; `nothing` if the header gives none
`base_time_ms::Union{Nothing,Int64}`	the start time in milliseconds after midnight (`"14:00:03.500"` gives `50403500`); `nothing` exactly when `base_time` is

Field	Contents
`base_date::Union{Nothing,NTuple{3,Int}}`	the start date as `(year, month, day)`; `nothing` if the header gives no date
`sampfrom::Int64`	the number of the first sample returned
`comments::Vector{Vector{UInt8}}`	the info strings of the record: the comments of the header after its record line (found as the header parser finds it: an empty or blank first line is not the record line; in legacy mode, as the original `getinfo` finds it), and the lines of its `.info` file if it has one (see `getinfo`)
`channels::Vector{Int}`	the number, in the record, of each signal returned (`0` is the first signal)
`checksum_ok::Bool`	`false` if the samples do not match the checksums of the header; checksums are checked only when a read reaches the end of the record, so a shorter read gives `true`

Table 9.2 The fields of `Header`, the result of `read_header`

Field	Contents
`name::String`	the record name, as given
`fs::Float64`	the sampling frequency in Hz, as `sampfreq` gives it; 0 if the header gives none
`counter_freq::Float64`	the counter frequency (`getcfreq`)
`base_counter::Float64`	the base counter value (`getbasecount`)
`nsamples::Int64`	the number of samples per signal, from the record line of the header (0 if the header does not give it)
`nsig::Int`	the number of signals
`siginfo::Vector{Siginfo}`	the signal specifications, one per signal; `desc` has no white space at either end
`base_time`, `base_time_ms`, `base_date`	the start time and date, as in `Record`
`comments::Vector{Vector{UInt8}}`	the info strings of the record, as in `Record`
`metadata::Vector{MetadataEntry}`	the metadata entries of the header, in file order, as `read_metadata` returns them: from the `.hea` file only; empty, without an error, for an EDF record and for a header read from the standard input

Table 9.3 The fields of `Annotations`, the result of `read_annotations` and the argument of `write_annotations`: parallel vectors, annotation `k` being `samples[k]`, `codes[k]`, `symbols[k]`, …, in file order

Field	Contents
`record::String`, `annotator::String`	the record and annotator names that were read, such as `"100s"` and `"atr"`; not used by `write_annotations`
`fs::Float64`	the frequency in which `samples` are counted, normally the sampling frequency of the record; 0 if unknown. `write_annotations` does not use it
`samples::Vector{Int64}`	the annotation times, in samples
`codes::Vector{Int32}`	the annotation codes, such as `WFDBX.NORMAL` or `WFDBX.PVC`

Field	Contents
`symbols::Vector{Vector{UInt8}}`	the mnemonics of the codes, as `annstr` returns them, such as `b"N"`, `b"V"` or `b"+"`; when writing, an empty mnemonic means "use `codes`"
`subtypes`, `chans`, `nums` (`Vector{Int32}`)	the subtype, channel number and annotator number of each annotation
`aux::Vector{Union{Nothing,Vector{UInt8}}}`	the auxiliary text of each annotation, without its length byte, or `nothing` where there is none

Table 9.4 The fields of `CheckReport`, the result of `check_record`

Member	Contents
`record::String`	the record name, as given
`ok::Bool`	`true` if no issue has the severity `"error"`
`issues::Vector{Issue}`	the problems, in the order of the checks
`lines(report)`	the text form of the report: one line per issue, or the one line `no problems found`
`print`, `show`	write the lines of `lines(report)`, joined by line feeds (no final line feed)

Table 9.5 The fields of `Issue`, a problem found by `check_record`

Member	Contents
`severity::String`	`"error"`, `"warning"` or `"info"`
`code::String`	a stable identifier of the kind of problem, such as `"checksum"` or `"gain_missing"` (Table 6.7)
`where::String`	where the problem is: `""` (the record), `"signal N"`, `"segment SEG"`, `"segment SEG, signal N"` or `"annotator ANN"`
`message::String`	the description of the problem
`string(issue)`	`SEVERITY: WHERE: MESSAGE`, or `SEVERITY: MESSAGE` when `where` is empty

Table 9.6 The fields of `Siginfo` (C's `WFDB_Siginfo`), the specification of a signal

Field	Contents
`fname::Union{Nothing,Vector{UInt8}}`	the name of the signal file, as the header gives it
`desc::Union{Nothing,Vector{UInt8}}`	the signal description, such as `b"MLII"`
`units::Union{Nothing,Vector{UInt8}}`	the physical units, such as `b"mV"`; `nothing` if not specified
`gain::Float64`	the gain, in ADC units per physical unit (0: uncalibrated)
`initval::Int32`	the value of the first sample
`group::UInt32`	the signal group number (the signals of one group share one file)
`fmt::Int32`	the storage format, such as `16`, `212` or `516`
`spf::Int32`	the number of samples per frame (more than 1 for the faster signals of a multi-frequency record)
`bsize::Int32`	the block size, in bytes (0 for ordinary files)
`adcres::Int32`	the ADC resolution, in bits (0: not specified)
`adczero::Int32`	the ADC output for an input of 0 V
`baseline::Int32`	the ADC value that corresponds to 0 physical units
`nsamp::Int64`	the number of samples (0: not specified)

Field	Contents
`cksum::Int32`	the 16-bit checksum of the samples

9.3.7 Warnings and errors

Exceptions. The high-level functions report errors by exceptions (Table 9.7). `WfdbError` is the exception type of WFDB-x; its field `msg` (a `String`) describes the failed operation, such as "can't read record REC", and `showerror` prints `WfdbError: MESSAGE`. The metadata functions and `check_record` throw Julia's `ArgumentError` (whose field is `msg` too) for an invalid argument, and a `MethodError` is thrown, as for any Julia function, for an argument of a type that no method accepts (such as a number as the value of `set_metadata`). Each function checks its arguments before it creates or changes a file, so an invalid argument leaves the files as they were; the writers also write through temporary files, so that a failure while writing leaves the existing files as they were. A signal file that ends before its header says, or cannot be read, is an error of `read_record` even in the frame at which a checksum is verified (whose checksum error the C-style `getvec` reports instead), and `check_record` reports it as a short file (`length_short`) or a read error (`read_error`), not as a checksum error. The library may also print a message of its own on the standard error when it cannot open a file (as in `init: can't open header for record REC`); `wfdberror()` returns the last such message, and `wfdbquiet()` turns these messages off (Section 9.4).

Warnings are logged with Julia's logging macro `@warn`, at the level `Warn`, so the current logger decides where they go (the standard error by default); `Test.@test_logs` checks them, and a logger such as `Base.CoreLogging.NullLogger()` silences them.

The checksum warning. When `read_record` reads to the end of a record whose samples do not match the checksums of its header, it returns all the samples and logs the warning "checksum error in record REC: the signal data do not match the checksums in its header". `checksum_warning=false` turns the warning off; `checksum_ok` of the result is `false` either way.

The duplicate-key warning. When a header holds two metadata entries whose keys differ only in letter case (written by other software), `read_metadata`, `get_metadata` and `read_header` use the first and log the warning "header of record REC has more than one entry for key KEY (lines L1, L2); the first is used" (KEY spelled as in its first entry). `set_metadata` and `update_metadata` then refuse to change that key (an `ArgumentError`), and `remove_metadata` removes all of its entries.

```
# A record whose last sample is changed after it was written: the checksum fails.
write_record("ck", reshape(Int32.(0:99), :, 1), 100; gains=100, adcres=12, units="mV")
dat = read("ck.dat"); dat[end-1:end] .= 0x00; write("ck.dat", dat)
rec = read_record("ck")          # logs the checksum warning
println(size(rec.samples, 1), " ", rec.checksum_ok)  # 100 false: all the samples
rec = read_record("ck"; checksum_warning=false)     # no warning

# Two entries for one key: the first is used, with a warning.
write_record("dup", zeros(Int32, 10, 1), 100; gains=0, adcres=12,
             comments=[" Age: 63", " AGE: 64"])
println(String(get_metadata("dup", "age")))         # 63 (with a warning)
try
    set_metadata("dup", "Age", "65")
catch e
    println(e.msg)
    # header of record dup has more than one entry for key Age (lines 3, 4); remove them first
end
println(remove_metadata("dup", "age"))              # 2
```

Table 9.7 The exceptions of the high-level API

Exception	Cases
`WfdbError`	a record, header or annotation file cannot be found, read, created or written completely (for example on a full disk): "can't read record REC", "can't read header for record REC", "can't write header for record REC", "write error on record REC", "directory D does not exist", ...; a signal or annotation file ends before its header or its end-of-file marker says ("... (getvec status S)", "... (getann status S)"); an invalid argument of the readers and writers (most messages of `write_record` and `write_annotations` begin with the function's name, including those of an invalid `metadata` argument of `write_record`); the operation needs a value that the record does not give: a time in seconds without a sampling frequency ("sampling frequency of record REC is not specified"), physical units without a gain ("signal N of record REC is uncalibrated")
`ArgumentError`	an invalid argument of the metadata functions: an invalid key or value, a record name that is empty or `"-"`, a key given more than once in `changes`, a key that the header has more than once (to be set), a header without a record line, or a header read from a URL to be changed without `dir`; an empty record name or an annotator name that is not a string, in `check_record`
`MethodError`	an argument of a type that no method accepts

```
try
    read_record("nosuchrecord")     # (the library also prints a message)
catch e
    println(e isa WfdbError, " ", e.msg)            # true can't read record nosuchrecord
end
```

9.4 The WFDB library functions in Julia

Every function of the WFDB library is a function of the module `WFDBX`, under its C name, with the same arguments, in the same order, and the same return values, so that programs and documentation written for the C library carry over directly. The functions, their arguments and their results are documented once, for all the languages, in Chapter 13; this section gives the rules by which the C declarations become Julia functions (see also Section 13.1.4). The docstring of each function (`?getvec`) summarizes it, and says what WFDB-x changes in it.

9.4.1 Conventions

The library state. The state of the C library (the open record, its signals, the annotators, the WFDB path, the time base, ...) is held by a `Wfdb` object, and each C function that uses it takes one as an optional first argument: `isigopen(w, "100s", si, 2)`, `getvec(w, v)`. Without it, the function uses the default object, `lib()`. `Wfdb()` creates an independent one; separate objects share no state, unlike the C library. The functions that use no state take none: `wfdbldflags`, `wfdbcflags`, `wfdbdefwfdb`, `wfdbdefwfdbcal`, `isann`, `ammap`, `mamap` (and their `wfdb_` forms), and `wfdbx_write_errors` and `wfdbx_note_write_error`, whose count belongs to the process.

Names. The functions, the constants (`WFDB_READ`, `WFDB_INVALID_SAMPLE`, `WFDB_HIGHRES`, ...) and the structures keep their C names. `using WFDBX` makes most of them available; the others are called with the module prefix: `WFDBX.wfdb_error`, `WFDBX.wfdb_me_fatal`, the annotation codes (`WFDBX.NORMAL`, `WFDBX.PVC`, ..., Chapter 15), and the function forms of the macros of `<wfdb/ecgmap.h>` (`WFDBX.wfdb_isann`, `WFDBX.wfdb_isqrs`, ...; `isann`, `isqrs`, `map1`, ... are exported).

Arguments. String arguments are `Strings` or bytes (`ByteStr`); a byte string ends at its first NUL byte, as a C string does. C's `NULL` is `nothing`: `sampfreq(nothing)` returns the frequency of the open record, and `getinfo(nothing)` continues with the info strings of the last record. Where C fills an array, the caller passes a vector: of `Siginfo` structures (`[Siginfo() for _ in 1:n]`), or of samples, which must be a `Vector{Int32}`; `nothing` stands for C's `NULL` (`isigopen("100s", nothing, 0)`). Where C fills a structure, the caller passes a mutable structure (`getann(0, a)` with `a = Annotation()`). Where C fills an `int` or a pointer, the caller passes a `Ref`: `getseginfo(r)` takes a `Ref`, which receives a vector of `Seginfo` (or `nothing`).

Strings are returned as bytes, `Vector{UInt8}`, or `nothing` where the C function may return `NULL` (`anndesc`, `getinfo`, `wfdbfile`, ...), except the results of `timstr`, `mstimstr` and `datstr`, which are `Strings`; unlike C's pointers to static buffers, they stay valid after later calls.

Numbers. The integer types are those of the C reference platform: `WFDB_Sample` and C's `int` are `Int32`, `WFDB_Time`, `WFDB_Date` and C's `long` are `Int64`, `WFDB_Frequency` and `WFDB_Gain` are `Float64`, C's `char` is `Int8`, and `WFDB_Signal`, `WFDB_Group` and `WFDB_Annotator` are `UInt32`. Julia's fixed-width arithmetic wraps around as C's does. Arrays are indexed from 1, but the values that C counts from 0 (signal numbers, annotator numbers, sample numbers) keep their C values.

Return values and errors. The functions return what the C functions return: an `Int32` status (negative on errors), a number of signals, a `Float64`, a string or `nothing`. Errors are reported by these values and by a message on the standard error, as in C, never by exceptions. `wfdberror()` returns the last message; `wfdbquiet()` turns the messages off and `wfdbverbose()` on again. `WFDBX.wfdb_error(fmt, args...)` formats its message as C's `printf`.

State. As in C, a `Wfdb` object keeps its state between calls: the open record, its input signals, the output signals and the annotators remain open until `wfdbquit()` closes them. `wfdbquit()` leaves the input and output directories and the network setting as they are. The high-level functions of Section 9.3 (except the metadata functions) call `wfdbquit()` on the `Wfdb` object they use, which closes anything opened with the library functions on that object.

Additions. `getvecs(out, n)` reads up to `n` sample vectors at once into `out`, a `Vector{Int32}` with room for `n` times the number of values of one vector, with exactly the result of `n` calls of `getvec`, but decoding whole blocks of the signal files; it returns the number of vectors read (≥ 0), or the negative status of `getvec` if the first call failed. `getvecs_status(out, n)` also gives the status of the last `getvec` call (Section 13.11):

```
getvecs([w], out::Vector{Int32}, n) -> Int64
getvecs_status([w], out::Vector{Int32}, n) -> (Int64, Int32)
```

`WfdbFile` is C's `FILE *` (or `WFDB_FILE`): `WFDBX.wfdb_open` and `WFDBX.wfdb_fopen` return a `WfdbFile` or `nothing`, read and written with the functions of the C standard library, which are not exported either (`WFDBX.getc`, `WFDBX.fread!`, `WFDBX.fgets!`, `WFDBX.fwrite`, `WFDBX.fseek`, `WFDBX.fclose`, ...).

9.4.2 The structures

The structures of the C library have the names of the C structures without the prefix `WFDB_`, and the C field names; they are mutable, with keyword constructors whose defaults are C's zero values (Table 9.8).

Table 9.8 The C structures in Julia

Structure	Fields
`Siginfo`	`fname`, `desc`, `units` (`Union{Nothing,Vector{UInt8}}`), `gain`, `initval`, `group`, `fmt`, `spf`, `bsize`, `adcres`, `adczero`, `baseline`, `nsamp`, `cksum` (Table 9.6)
`Seginfo`	`recname` (the name of the segment, bytes), `nsamp` (its number of samples, `Int64`), `samp0` (the number of its first sample in the record, `Int64`)
`Anninfo`	`name` (the annotator name, bytes), `stat` (`Int32`: `WFDB_READ` or `WFDB_WRITE` for the MIT format, `WFDB_AHA_READ` or `WFDB_AHA_WRITE` for the AHA format); `Anninfo("atr", WFDB_READ)`
`Annotation`	`time` (`Int64`, in sample intervals), `anntyp` (`Int8`), `subtyp` (`Int8`), `chan` (`UInt8`), `num` (`Int8`), `aux` (`nothing`, or bytes whose first byte is the length of the text that follows, as in C)
`Calinfo`	`low`, `high` (the levels of the calibration pulse, in physical units, `Float64`), `scale` (physical units per cm), `sigtype` and `units` (bytes or `nothing`), `caltype` (`Int32`)

9.4.3 The settings functions

The functions of Table 9.9 are additions of WFDB-x to the library: they set where files are read and written, network access and legacy mode, for the high-level functions, the library functions and the tools alike (Chapters 3 and 4). Like the other library functions, each takes an optional `Wfdb` object first, except `wfdbx_write_errors` and `wfdbx_note_write_error` (the count belongs to the process). Each `set` function returns the previous setting, so that it can be restored. The C function `setwfdb(p)` sets the input directory too (`p` must name at most one directory), and `getwfdb()` returns it (`"."` if none is set); in legacy mode they set and return the WFDB path, as in C.

Table 9.9 The settings functions (`[w]`: the optional `Wfdb` object)

Function	Effect and result
`wfdbx_set_input_dir([w], dir::OptByteStr)`	sets the input directory: a local directory, a ~ path or a URL; `nothing` or `""` restores the default (`WFDBX_INPUT_DIR`, or else the current directory). Returns the previous input directory as bytes, or `nothing` if none was set
`wfdbx_get_input_dir([w])`	returns the input directory as bytes, or `nothing` if none is set
`wfdbx_set_output_dir([w], dir::OptByteStr)`	sets the output directory: a local directory or a ~ path (a name that begins with a URL scheme, such as `https://` or `ftp://`, is refused, with a message, and the setting is unchanged); `nothing` or `""` restores the default (`WFDBX_OUTPUT_DIR`, or else the current directory). Returns the previous output directory, or `nothing`
`wfdbx_get_output_dir([w])`	returns the output directory as bytes, or `nothing` if none is set
`wfdbx_set_network([w], on::Union{Nothing,Bool,Integer})`	turns network access on (`true` or > 0) or off (`false` or 0); `nothing` or a negative number restores the default (on if `WFDBX_NETWORK` is a nonzero number). Returns the previous setting, 1 or 0 (`Int32`)

Function	Effect and result
`wfdbx_get_network([w])`	returns 1 if network access is on, 0 otherwise
`wfdbx_set_legacy([w], on::Union{Bool,Integer})`	turns legacy mode on (`true` or nonzero) or off. Returns the previous setting, 1 or 0
`wfdbx_get_legacy([w])`	returns 1 if legacy mode is on, 0 otherwise
`wfdbx_legacy_args([w], args::Vector{String})`	if `args[1]` (the arguments *without* the program name, as Julia's `ARGS`) is `"--legacy"`, removes it, turns legacy mode on and returns 1; otherwise returns 0
`wfdbx_time_needs_freq([w], string::ByteStr)`	returns 1 if `strtim(string)` needs a sampling frequency, 0 otherwise
`wfdbx_write_errors()`	returns the number of write errors since the program started (output files that could not be written, flushed or closed); under `WFDBX.run_tool`, since the tool started
`wfdbx_note_write_error()`	counts a write error, for a program that writes a file itself
`wfdbx_input_path([w], name::OptByteStr)`	returns, as bytes, the path at which a program opens a file that it reads itself (not through the library): where `wfdb_open` finds a file of that name (in normal mode, a relative name under the input directory, a ~ name expanded, an absolute name as given; in legacy mode, the name as given). `""` and `"-"` are returned as given; `nothing` for `nothing`, an `http://` or `https://` URL, or, in normal mode, a name that begins with another URL scheme (also once the input directory is prefixed to it)
`wfdbx_output_path([w], name::OptByteStr)`	returns, as bytes, the path at which a program creates a file that it writes itself: where `wfdb_open` creates a file of that name (in normal mode, a relative name under the output directory, a ~ name expanded, an absolute name as given; in legacy mode, any name under the output directory, used as a plain prefix). When the output directory has been prefixed, the missing directories of the path are created. `""` and `"-"` are returned as given; `nothing` for `nothing`, an `http://` or `https://` URL, or, in normal mode, a name that begins with another URL scheme

9.4.4 Examples

Reading the samples and the annotations of a record, as a C program does (the record `100s` is read from the current directory):

```
si = [Siginfo() for _ in 1:2]                       # receives one Siginfo per signal
nsig = isigopen("100s", si, 2)                      # open 2 signals
println(nsig, " ", String(si[1].desc), " ", si[1].fmt, " ", si[1].gain)  # 2 MLII 212 200.0
println(sampfreq(nothing))                          # 360.0: the frequency of the open record
```

```
v = zeros(Int32, 2)                              # receives one sample per signal
isigsettime(strtim("0:30"))                      # skip to 30 seconds
getvec(v)
println(v[1], " ", v[2], " ", aduphys(0, v[1]))  # 947 968 -0.385

frames = zeros(Int32, 360 * 2)                   # the next second, in one call
println(getvecs_status(frames, 360))             # (360, 2)

annopen("100s", [Anninfo("atr", WFDB_READ)], 1)
a = Annotation()
n = 0
while getann(0, a) == 0                          # 0: an annotation was read; -1: the end
    global n += 1
    n <= 2 && println(mstimstr(a.time), " ", String(annstr(a.anntyp)))
end
#     0:00.050 +
#     0:00.214 N
println(n, " annotations")                       # 75 annotations

wfdbquiet()                                      # no messages on the standard error
stat = isigopen("nosuchrecord", nothing, 0)
print(stat, " ", String(wfdberror()))
# -1 init: can't open header for record nosuchrecord
wfdbverbose()
wfdbquit()                                       # close everything
```

Writing a record and an annotation file, sample by sample and annotation by annotation:

```
out = Siginfo(fname=b"ramp.dat", desc=b"ramp", units=b"mV", gain=200.0, fmt=16, adcres=12)
println(osigfopen([out], 1))                     # 1: one output signal
setsampfreq(250.0)
for k in 0:499
    putvec(Int32[k])
end
println(newheader("ramp"))                       # 0: ramp.hea is written
annopen("ramp", [Anninfo("qrs", WFDB_WRITE)], 1)
for t in (100, 350)
    putann(0, Annotation(time=t, anntyp=WFDBX.NORMAL))
end
wfdbquit()                                       # completes ramp.dat and ramp.qrs
println(length(read_annotations("ramp", "qrs")))  # 2
```

9.5 The command-line tools in Julia

The 33 command-line tools (`rdsamp`, `rdann`, `wrsamp`, `xform`, ...) are functions of the package, one file per tool in `julia/src/apps`, run through the driver script `julia/bin/wfdbx.jl`: its first argument names the tool, and the other arguments go to the tool. Each follows its C program line by line. The tools, their options, their output and their exit status are those of the C tools, and are documented in Chapter 14. With the shell script `wfdbx` of Section 9.1:

```
export WFDBX_INPUT_DIR=/path/to/wfdb-x/tests/data
wfdbx rdsamp -r 100s -t 10 -p     # the first 10 seconds of 100s, in physical units
wfdbx rdann -r 100s -a atr        # its annotations
wfdbx rdsamp --legacy -r 100s     # legacy mode: --legacy as the first argument
wfdbx --version                   # WFDB-x 1.2.3
```

Without the script, the command is `julia --startup-file=no --project=/path/to/wfdb-x/julia /path/to/wfdb-x/julia/bin/wfdbx.jl TOOL ARGUMENTS`. Every run starts a new Julia process, which loads the precompiled package; most of the start-up time of a tool is Julia's own. With no arguments, or an unknown tool name, the driver prints a message on the standard error and exits with status 1. When an output file or the standard output cannot be written (a full disk, for example), a tool reports it ("PROGRAM: write error on standard output") and exits with status 1; with `--legacy`, it keeps the original exit status. A tool whose standard output is a closed pipe ends with status 141, as a C program ended by `SIGPIPE`. The driver finds the compiled package in the depot of the user's own home directory even when the environment variable `HOME` is changed (`JULIA_DEPOT_PATH`, if set, is used as given).

Inside a Julia session. `WFDBX.run_tool(tool, args; stdin=UInt8[], dir=nothing, env=nothing)` runs a tool in the current process, with a fresh library state, in the directory `dir`, with the environment variables of `env` (`name => value` pairs) set (`WFDB`, `WFDBCAL`, `WFDBANNSORT`, `WFDBGVMODE`, `WFDBX_INPUT_DIR`, `WFDBX_OUTPUT_DIR` and `WFDBX_NETWORK` are restored afterwards, other variables are not), and the bytes `stdin` as its standard input, and returns its exit status and what it wrote to the standard output and the standard error, as bytes. The caller's library state and standard streams are restored when the tool returns. An unknown tool name is an `ArgumentError`. `WFDBX.tools()` returns the names of the tools.

```
status, out, err = WFDBX.run_tool("rdsamp", ["-r", "100s", "-t", "s3"]; dir=".")
println(status)                                  # 0
print(String(out))       # 3 lines: the sample number, then 995 and 1011 (tab-separated)
println(length(WFDBX.tools()))                   # 33
```

Environment variables. The tools, like the rest of the library, use the environment variables of Table 9.10, also for the files that they read or create themselves: the `-i` file of `wrsamp`, `edf2mit`, `wav2mit`, `a2m` and `ad2m`, the input files of `ahaecg2mit` and the `-S` script of `xform` are read from the input directory (`wfdbdesc` measures a signal file in the directory of its header), and the WAV file of `mit2wav`, the EDF file of `mit2edf`, the master header of `snip -m` and the `.des`, `.ecg`, `.key` and `.txt` files of `parsescp` are created in the output directory, as `wfdbx_input_path` and `wfdbx_output_path` locate them (Section 9.4.3). Such a file cannot be read from a URL. In legacy mode, the files that a tool reads itself are opened as named.

Table 9.10 The environment variables used by the Julia library

Variable	Meaning
`WFDBX_INPUT_DIR`	the input directory (a directory, a ~ path or a URL), when `wfdbx_set_input_dir` does not set one
`WFDBX_OUTPUT_DIR`	the output directory, when `wfdbx_set_output_dir` does not set one
`WFDBX_NETWORK`	network access is on if it is a nonzero number, when `wfdbx_set_network` does not set it
`WFDBANNSORT`	`0` to keep annotation files in the order written (no sorting when they are closed)
`WFDBGVMODE`	the default getvec mode (`setgvmode` with a negative mode)
`WFDBCAL`	the calibration file read by `calopen(nothing)` (default `wfdbcal`)
`WFDB`	the WFDB path, used in legacy mode only (outside legacy mode, it is ignored, with a message)
`WFDB_FLAC_CHECK_MD5`	if set, the MD5 signature of FLAC signal files is verified when they are read
`HOME` (`USERPROFILE` on Windows)	the home directory, for ~ paths

Tools that write signal parameters require them as options: `wrsamp` (with `-o`) requires `-F` (frequency), `-G` (gain), `-R` (ADC resolution) and, for calibrated signals, `-U` (units); `wav2mit` requires `-G`, and `-U` unless the gain is 0; `ad2m` and `ahaecg2mit` require `-F`, `-G`, `-R`, and `-U` unless the gain is 0; `a2m`, `rdann`, `ann2rr`, `sortann` and `m2a` require `-F` when the record's header gives no frequency (Chapter 14).

9.6 Testing

The unit tests of the Julia implementation are in `julia/test`, and use the standard library `Test`. From the root of the repository:

```
julia --startup-file=no --project=julia julia/test/runtests.jl   # all the unit tests
julia --project=julia -e 'using Pkg; Pkg.test()'                  # the same, through Pkg
```

The tests cover the C standard functions (`printf`, `strtol`, `strtod`, `sscanf`, ..., against glibc), the files and the library functions (`runtests.jl`, `signals.jl`, `test_annotations.jl`), the FLAC codec (`test_flac.jl`), finding files, the settings and network access (`test_files.jl`), the high-level functions, the metadata functions (`test_metadata.jl`, with the cases shared by the seven languages, in `tests/metadata`), `check_record` (`test_check.jl`) and the tools run in-process. They run in temporary copies of the test records of `tests/data`. The tests of network access use a local HTTP server on 127.0.0.1 only (written with the standard library `Sockets`). The tests of `check_record` write the corpus of `python/tests/check_corpus.py` with `python3`, NumPy and the Python package of `python/src`, and compare the reports with its `expected.txt`; they are skipped if that script cannot run.

The cross-language conformance suite runs every tool and compares its results, byte for byte, with those of the original C programs (1155 cases), and, with `--legacy`, in legacy mode (886 cases). It runs the tools through `julia/bin/wfdbx.jl`:

```
PYTHONPATH=$PWD/python/src python3 tests/conformance/harness.py check --impl julia -j8
PYTHONPATH=$PWD/python/src python3 tests/conformance/harness.py check --impl julia -j8 --legacy
```

CHAPTER 10

R

The R implementation of WFDB-x is the package `wfdbx`, in the directory `r` of the repository. It is written in base R alone: it needs no other package, has no compiled code, and never calls the C library. It reads and writes the same files as the original C library, and its tools print the same output.

It offers WFDB-x in three ways:

- **The high-level API** (Section 10.3): functions that read and write whole records, headers, annotation files and header metadata as R matrices, lists and data frames, in one call, with named arguments, and that stop with R errors on failure. They read signal files in large blocks with vectorized code, and are the fastest way to read a record. They are an addition of WFDB-x: the C package has no equivalent. They are the recommended interface for new code and analysis.
- **The WFDB library functions** under their C names (`isigopen`, `getvec`, `getann`, `putann`, `...`), with the behavior of the C library (Section 10.4). They are documented once, for all the languages, in Chapter 13.
- **The command-line tools** (`rdsamp`, `rdann`, `wrsamp`, `xform` and 29 others), with the options, output and exit status of the C tools, run with `Rscript` through the driver script `wfdbx.R`, or from R with `wfdbx_tool` (Section 10.5). They are documented in Chapter 14.

The high-level functions return exactly the samples and annotations that the library functions (and the C library) give.

10.1 Installation and requirements

10.1.1 Requirements

WFDB-x for R needs R 4.1 or later, and nothing else: the package uses base R only (its one import, `tools`, is part of R). It has no compiled code and needs no C compiler, no WFDB library and no libFLAC (the FLAC formats 508, 516 and 524 are read and written by WFDB-x's own codec, in R). It runs on Linux and macOS. On Windows, the command-line tools write their output as text, so NUL bytes in it are lost (Section 10.5).

Two programs are used if they are installed: `curl`, to download remote files (base R's `url()` is used otherwise), and `stat`, to recognize an output file that is also an input file under another path. The tests also use `python3` (Section 10.6).

10.1.2 Installing the package

From the root of a copy of the repository, the package is in the directory `r`:

```
R CMD INSTALL r                  # install the package wfdbx
```

or, from R, with the path of the directory `r`:

```
install.packages("r", repos = NULL, type = "source")
library(wfdbx)
```

The package is attached with `library(wfdbx)`. Its version is given by `packageVersion("wfdbx")`, and by the driver script of the tools:

```
Rscript -e 'cat(format(packageVersion("wfdbx")), "\n")'    # 1.2.3
Rscript r/inst/bin/wfdbx.R --version                        # WFDB-x 1.2.3
```

The help pages of the package (`?read_record`, `?read_metadata`, `?isigopen`, ...) document each function; `?wfdbx` gives an overview.

The package holds code under two licenses, which its file `LICENSE` states (its `DESCRIPTION` says `License: file LICENSE`): the GPL, version 3 or later, for the tools (`R/apps_*.R`, `R/cli.R` and the driver), and the LGPL, version 2.1 or later, for everything else. It installs, with these licenses (`COPYING`, the GPL, and `COPYING.LESSER`, the LGPL), the file `NOTICE` (`system.file("NOTICE", package = "wfdbx")`), which summarizes the licenses of WFDB-x and gives the BSD-3-Clause notice of libFLAC, whose stream decoder the FLAC decoder of WFDB-x (`flac.R`) translates.

10.1.3 The command-line tools

The tools are run by the driver script `wfdbx.R` with `Rscript`. In a copy of the repository, the script is `r/inst/bin/wfdbx.R`, and it runs the tools without the package being installed (it then loads the sources of `r/R`); in an installed package, its path is `system.file("bin", "wfdbx.R", package = "wfdbx")` (Section 10.5).

10.1.4 The sample record

The repository includes the sample record `100s`, the sample record of the WFDB Software Package: the first minute of record 100 of the MIT-BIH Arrhythmia Database, two ECG signals (`MLII` and `V5`) sampled at 360 Hz, with the reference beat annotations (annotator `atr`). Its files (`100s.hea`, `100s.dat`, `100s.atr`) are in the directory `tests/data`, with the other test records of WFDB-x; the R package does not install them. Work that uses the MIT-BIH Arrhythmia Database cites it as described on PhysioNet.

10.2 Quick start

This example reads the header, ten seconds of the signals and the annotations of the sample record `100s`, from the directory `tests/data` of a copy of the repository (`input_dir`; the path is that of the copy), and computes the mean heart rate. It then writes the first ten seconds as a new record, `first10`, in the current directory, with two metadata entries in its header.

```
library(wfdbx)

data <- "wfdb-x/tests/data"            # the directory of the sample record 100s

# The header only: no samples are read.
hdr <- read_header("100s", input_dir = data)
cat(hdr$nsig, hdr$fs, hdr$nsamples, "\n")          # 2 360 21600
print(sapply(hdr$siginfo, function(s) s$desc))     # "MLII" "V5"

# Seconds 10 to 20 of both signals, in physical units: one row per sample, one
# column per signal.
rec <- read_record("100s", sampfrom = "0:10", sampto = "0:20", physical = TRUE,
                   input_dir = data)
```

```
print(dim(rec$signals))                # 3600 2
print(rec$signals[1, ])                # -0.390 -0.275

# The beat annotations, and the mean heart rate.
ann <- read_annotations("100s", "atr", input_dir = data)
cat(nrow(ann), head(ann$symbol, 3), head(ann$sample, 3), "\n")   # 75 + N N 18 77 370
beats <- ann$sample[isqrs(ann$code) == 1]
rr <- diff(beats) / attr(ann, "fs")     # RR intervals, in seconds
cat(length(beats), "beats,", round(60 / mean(rr), 1), "beats per minute\n")   # 74 beats, 73.9 ...

# Write the first 10 seconds, in ADC units, as the record "first10" (first10.hea and
# first10.dat, in the current directory), with the signal specifications of 100s.
x <- read_record("100s", sampto = "0:10", input_dir = data)
write_record("first10", x$signals, fs = x$fs, fmt = 16, gains = x$gains,
             baselines = x$baselines,
             adczeros = sapply(x$siginfo, function(s) s$adczero),
             adcres = sapply(x$siginfo, function(s) s$adcres), units = "mV",
             names = x$names, metadata = list(Source = "MIT-BIH record 100"))

# Header metadata: "# Key: value" lines of the header.
set_metadata("first10", "Age", "69")
print(read_metadata("first10"))        # c(Source = "MIT-BIH record 100", Age = "69")
print(check_record("first10"))         # no problems found
```

The header `100s.hea` gives no units: the example writes `"mV"` explicitly, since WFDB-x never assumes units, a gain, an ADC resolution or a sampling frequency that a record does not give (Chapter 4).

10.3 The high-level API

The high-level API is made of the functions of this section and of the lists and data frames that they return (Section 10.3.6). They are defined in the files `highlevel.R`, `highlevel_annot.R`, `metadata.R` and `check.R` of `r/R`, and documented by the help pages `?read_record` (which also covers `read_header` and `write_record`), `?read_annotations` (also `write_annotations`), `?check_record` and `?read_metadata` (the five metadata functions).

10.3.1 Conventions

Record names and directories. A record is named as in the other parts of WFDB-x (Chapters 2 and 3), without the `.hea` extension (the readers, `write_annotations` and `write_record` also accept a trailing `.hea`, which is removed). A record name is a full path, absolute (`"/data/mitdb/100"`) or in the home directory (`"~/Documents/100s"`), used as given; a relative path, such as `"100s"` or `"data/100s"`, read from the input directory if one is set, otherwise from the current directory, and written to the output directory if one is set, otherwise to the current directory (missing directories are created); or a URL (`"https://physionet.org/files/mitdb/1.0.0/100"`), read only if network access is on, which cannot be written. Only `http://` and `https://` URLs are read: a name that begins with another URL scheme (`ftp://`, `file://`, `...`) is refused, with or without network access (Section 3.4). Nothing is searched for (legacy mode searches the WFDB path). A writer given a name with a path creates the files in that directory, and the header names the record and its signal file without the path (`rec`, `rec.dat`), so that the record opens under any path that leads to it; in legacy mode, as in the original, the header gives the name with its path. The input and output directories are the current directory unless they are set, with `wfdbx_set_input_dir` and `wfdbx_set_output_dir` (Section 10.4.3) or with the environment variables `WFDBX_INPUT_DIR` and `WFDBX_OUTPUT_DIR`. The functions that read take the arguments `input_dir` and `network`, and those

that write take `dir`: they apply to that one call, and the previous settings are restored when it returns, also after an error. `NULL`, the default of these arguments, keeps the current setting (for `dir`, `""` too). `input_dir` applies to relative names only, not to full paths or URLs. With `dir`, the name must be a plain name (`"rec"`): the files are created in `dir`, and the record opens from that directory under its plain name. The signal files that a header names are read from the header's directory. A name with another extension, such as `"rec.edf"`, is read as an EDF file.

Times. The arguments `sampfrom` and `sampto` are sample numbers (whole numbers) or WFDB time strings, as `strtim` reads them: `"0:10"` (10 seconds), `"1:02:03.5"`, `"s3600"` (sample 3600), `"e"` (the end of the record), or a time of day in brackets, `"[14:00:00]"`. A time in seconds needs the sampling frequency; when the record does not give it, such a time is an error.

Per-signal and per-annotation arguments. The arguments that describe each signal (`fmt`, `gains`, `units`, ...) or each annotation (`subtype`, `chan`, ...) take one value for all, or a vector (or, for `units` and `names`, a list) with exactly one value per signal or per annotation; any other length is an error.

Numbers and text. Numbers are doubles: samples, times and counts are `double` vectors (exact for every 32-bit sample and for times up to 2^{53}). Signal numbers start at 0, as in WFDB. Text is handled as bytes: what the files hold is returned as character strings of their bytes, never re-encoded, and character strings given are written as their bytes.

Signal parameters. No sampling frequency, gain, ADC resolution or units is ever assumed. Where one is needed and the record does not give it, it must be given as an argument; otherwise the function stops with an error. A record whose header gives no sampling frequency has `fs = 0`.

Legacy mode. With `legacy = TRUE`, a function behaves as in the original WFDB package, with its assumed signal parameters and its search for files (Chapter 4); the library's legacy mode is on for the duration of the call only. With `legacy = FALSE` (the default), the current setting of `wfdbx_set_legacy` applies, which is off unless it was turned on.

State. The functions that read and write records and annotation files close anything opened with the WFDB library functions (they call `wfdbquit()`) before they start, and close their own files before they return, whatever the outcome. `read_record` sets the getvec mode for its own reading only. `read_record`, `read_header`, `read_annotations`, `write_record`, `write_annotations` and `check_record` leave the getvec mode (`getgvmode`) and the time resolution of output annotators (`getafreq`) as they were, also after an error: opening an EDF record sets both, as in C (`WFDB_HIGHRES` is added to the getvec mode, and the time resolution becomes the record's sampling frequency when none is set), and `wfdbquit()` resets neither; the `fs` of `write_annotations` applies to its file only. The metadata functions open no signals and leave the library's state as it is.

The examples of this section read the test records of WFDB-x (among them `100s`) from the current directory, a copy of `tests/data`, and write records of their own there. They were run with `Rscript`, and print what their comments say.

10.3.2 Reading records

`read_record` — Read the signals of a record

```
read_record(record, channels = NULL, sampfrom = 0, sampto = NULL,
            physical = FALSE, highres = FALSE, pad = FALSE,
            input_dir = NULL, network = NULL,
            legacy = FALSE, checksum_warning = TRUE)
```

Reads the signals of a record, or part of it, into a numeric matrix with one row per sample and one column per signal, together with the description of the record. The samples are those that `getvec` returns (Chapter 13), read in large blocks. Invalid samples are `WFDB_INVALID_SAMPLE` (-32768) in ADC units, and `NA` in physical units. When the read reaches the end of the record, the samples are compared with the checksums of the header (Section 10.3.7).

`record`
: Record name (a character string), without the `.hea` extension (a trailing `".hea"` is also accepted): a full path, absolute (`"/data/mitdb/100"`) or in the home directory (`"~/Documents/100s"`), used as given; a relative path (`"100s"`, `"data/100s"`), read from the input directory if one is set, otherwise from the current directory; or a URL (`"https://physionet.org/files/mitdb/1.0.0/100"`), read only if network access is on.

`channels`
: The signals to read, in the order in which they are returned: signal numbers (from 0) or signal descriptions (as in `names`, such as `"MLII"`). Default `NULL`: all the signals, in the order of the header.

`sampfrom`
: The first sample to read: a sample number (a whole number ≥ 0) or a time string. Default `0`: the beginning of the record.

`sampto`
: The sample after the last one to read: a sample number (a whole number) or a time string. A value at or before `sampfrom` reads no samples. Default `NULL`: the end of the record.

`physical`
: `FALSE` (the default) to return ADC units; `TRUE` to return physical units, computed as $(adc - baseline)/gain$. Every signal read must then have a gain.

`highres`
: `FALSE` (the default) to read a multi-frequency record at its frame frequency, with the samples of the faster signals averaged to one per frame; `TRUE` to read every sample, at the highest sampling frequency of the record (the samples of the slower signals are repeated). `fs` of the result is then that frequency.

`pad` `TRUE` to replace each invalid sample by the previous valid sample of the same signal. Default `FALSE`.

`input_dir`
: The input directory for this call only (optional): a local directory, a `~` path or a URL, under which relative record names are read. It does not apply to full paths or URLs. Default `NULL`: the current setting (`wfdbx_set_input_dir`).

`network`
: `TRUE` to allow reading URLs for this call only, `FALSE` to forbid it. Default `NULL`: the current setting (`wfdbx_set_network`).

`legacy`
: `TRUE` to behave as the original WFDB package (legacy mode, for this call only). Default `FALSE`.

`checksum_warning`
: `TRUE` (the default) to issue an R warning when the samples do not match the checksums of the header; `FALSE` for no warning. The samples are returned either way, and `checksum_ok` of the result is `FALSE`.

Returns A list (Table 10.1).

Errors Stops with an error if `sampfrom` is negative ("sampfrom must not be negative"), or `sampfrom` or `sampto` is not a whole number (a fraction, `NA`, `NaN` or an infinite value: "sampfrom must be a whole number"), is beyond the range of a 64-bit integer ("sampfrom is beyond the range of a 64-bit integer"), or is neither one number nor one time string ("sampfrom must be a sample number or a time string"); if the header cannot be read or the record has no signals ("can't read record REC"), or its signals cannot be opened ("can't open signals of record REC"); if a time string needs the sampling frequency and the header gives none ("sampling frequency of record REC is not specified"); if a channel is not in the record ("record REC has no signal C"); if `physical = TRUE` and a signal read has no gain ("signal K of record REC is uncalibrated"); if `sampfrom` cannot be reached ("can't seek to sample N of record REC"); or if a signal file is shorter than its header says or cannot be read ("can't read signal data of record REC (getvec status S)", S being the status of `getvec`, such as -3), even if the samples do not match the checksums either: no incomplete frame is ever returned (`sampto` reads the frames before the end of a truncated file). Warns on a checksum error (see `checksum_warning`).

```
library(wfdbx)

# Seconds 10 to 20 of lead MLII, in physical units.
rec <- read_record("100s", channels = "MLII", sampfrom = "0:10", sampto = "0:20",
                   physical = TRUE)
print(dim(rec$signals))                  # 3600 1
cat(rec$names, rec$fs, rec$sampfrom, "\n")   # MLII 360 3600

# The first 3 samples of both signals, in ADC units, V5 first.
rec <- read_record("100s", channels = c(1, 0), sampto = 3)
print(rec$signals)                       # three rows of 1011 995
print(rec$gains)                         # 200 200
print(rec$baselines)                     # 1024 1024
str(rec$units)                           # a list of two NULL: no units in the header

# The whole record: the checksums are verified.
rec <- read_record("100s")
cat(dim(rec$signals), rec$checksum_ok, "\n")   # 21600 2 TRUE
```

A multi-frequency record (here the test record `mfreq`, whose signals have 1, 2 and 4 samples per frame at a frame frequency of 100 Hz) is read at its frame frequency, or at its highest sampling frequency:

```
library(wfdbx)

rec <- read_record("mfreq")
cat(rec$fs, dim(rec$signals), "\n")      # 100 600 3
rec <- read_record("mfreq", highres = TRUE)
cat(rec$fs, dim(rec$signals), "\n")      # 400 2400 3
print(sapply(rec$siginfo, function(s) s$spf))   # 1 2 4
```

---

read_header

Read the header of a record

```
read_header(record, input_dir = NULL, network = NULL,
            legacy = FALSE)
```

Reads the header of a record, without opening its signal files. The result describes the record and each of its signals, and holds the info strings and the metadata entries of the header. For a

multi-segment record, the master header is read, and `nsamples` is the length of the whole record. The metadata are read from the `.hea` file only; an EDF file, which has no `.hea` file, and a header read from the standard input (`"-"`) have none.

`record`
: Record name (a character string), without the `.hea` extension (a trailing `".hea"` is also accepted): a full path, absolute (`"/data/mitdb/100"`) or in the home directory (`"~/Documents/100s"`), used as given; a relative path (`"100s"`, `"data/100s"`), read from the input directory if one is set, otherwise from the current directory; or a URL (`"https://physionet.org/files/mitdb/1.0.0/100"`), read only if network access is on.

`input_dir`
: The input directory for this call only (optional): a local directory, a ~ path or a URL, under which relative record names are read. It does not apply to full paths or URLs. Default `NULL`: the current setting (`wfdbx_set_input_dir`).

`network`
: `TRUE` to allow reading URLs for this call only, `FALSE` to forbid it. Default `NULL`: the current setting.

`legacy`
: `TRUE` to read the header as the original WFDB package does. Default `FALSE`.

Returns A list of class `"wfdbx_header"` (Table 10.2).

Errors Stops with an error if the header cannot be found or read ("can't read header for record REC"). Warns, as `read_metadata` does, for each key of which the header has more than one entry.

```
library(wfdbx)

hdr <- read_header("100s")
cat(hdr$name, hdr$fs, hdr$nsig, hdr$nsamples, "\n")   # 100s 360 2 21600
for (s in hdr$siginfo)
  cat(s$desc, s$fname, s$fmt, s$gain, s$adcres, is.null(s$units), "\n")
# MLII 100s.dat 212 200 11 TRUE
# V5 100s.dat 212 200 11 TRUE
print(hdr$comments[1])          # " 69 M 1085 1629 x1": the first info string
print(is.null(hdr$base_time))   # TRUE: the header gives no start time
print(hdr$metadata)
# c("Produced by xform from record 100, beginning at 0" = "0"): the last info string
# contains a colon, so it is an entry
```

`read_annotations` — Read an annotation file

```
read_annotations(record, annotator, sampfrom = 0, sampto = NULL, fs = NULL,
                 input_dir = NULL, network = NULL,
                 legacy = FALSE)
```

Reads the annotations of an annotation file, in file order, into a data frame with one row per annotation. The sampling frequency, needed to convert times in seconds, is that of the record's header, or, for a record without a header or whose header gives none, the argument `fs`.

`record`
: Record name (a character string), without the `.hea` extension (a trailing `".hea"` is also accepted): a full path, absolute (`"/data/mitdb/100"`) or in the home directory (`"~/Documents/100s"`),

used as given; a relative path (`"100s"`, `"data/100s"`), read from the input directory if one is set, otherwise from the current directory; or a URL (`"https://physionet.org/files/mitdb/1.0.0/100"`), read only if network access is on. Its header, if there is one, gives the sampling frequency.

annotator
Annotator name: the suffix of the annotation file. `"atr"` reads the file `RECORD.atr`.

sampfrom
The first time to read: a sample number (a whole number ≥ 0) or a time string. Default `0`: the beginning of the file.

sampto
The time at which to stop: annotations at or after it are not read. A sample number (a whole number) or a time string. Default `NULL`: the end of the file.

fs The sampling frequency, in Hz (a number > 0), to use when the record has no header or its header gives none; it is ignored otherwise. Default `NULL`: none.

input_dir
The input directory for this call only (optional): a local directory, a ~ path or a URL, under which relative record names are read. It does not apply to full paths or URLs. Default `NULL`: the current setting (`wfdbx_set_input_dir`).

network
`TRUE` to allow reading URLs for this call only, `FALSE` to forbid it. Default `NULL`: the current setting.

legacy
`TRUE` to behave as the original WFDB package (which assumes 250 Hz for a record without a frequency). Default `FALSE`.

Returns A data frame (Table 10.3), with the attributes `record`, `annotator` and `fs`. `attr(ann, "fs")` is the frequency in which the times are counted: the one written in the annotation file, if there is one, or else the sampling frequency; it is 0 if none of them is known.

Errors Stops with an error if the annotation file cannot be opened ("can't read annotator ANN for record REC"); if `sampfrom` cannot be reached ("can't seek in annotator ANN"); if the file cannot be read, or ends before its end-of-file marker, before `sampto` is reached ("can't read annotation file ANN of record REC (getann status S)", S being the status of `getann`, such as -3); if `sampfrom` or `sampto` is not valid, checked as by `read_record` before any file is opened ("sampfrom must be a whole number", "sampfrom must be a sample number or a time string", "sampfrom is beyond the range of a 64-bit integer", "sampfrom must not be negative", and the same for `sampto`, which may be negative); if `fs` is given and is not > 0 ("fs must be > 0"); or if a time string needs the sampling frequency and none is known. Whatever happens, the annotation file is closed (`wfdbquit`) before the function returns.

```
library(wfdbx)

ann <- read_annotations("100s", "atr", sampfrom = "0:10", sampto = "0:30")
cat(nrow(ann), attr(ann, "fs"), "\n")    # 24 360
print(head(ann, 3))
#   sample symbol code subtype chan num  aux
# 1   3862      N    1       0    0   0 <NA>
# 2   4170      N    1       0    0   0 <NA>
# 3   4466      N    1       0    0   0 <NA>

# The whole file: the first annotation is a rhythm change, with an aux string.
ann <- read_annotations("100s", "atr")
cat(ann$symbol[1], ann$code[1] == RHYTHM, ann$aux[1], "\n")   # + TRUE (N
```

```
print(sort(unique(ann$symbol)))          # "+" "A" "N"
```

For a record without a header, `fs` gives the frequency in which times in seconds are converted (the library reports on the standard error that it cannot open the header):

```
library(wfdbx)

write_annotations("beats", "qrs", c(100, 460, 820), symbol = "N")   # no header
ann <- read_annotations("beats", "qrs", sampto = "0:02", fs = 360)
cat(ann$sample, attr(ann, "fs"), "\n")   # 100 460 360
```

10.3.3 Writing records

write_record — Write a header and a signal file

```
write_record(record, signals, fs, fmt = 16, gains, baselines = NULL,
             adczeros = 0, adcres = NULL, units = NULL, names = NULL,
             base_time = NULL, comments = NULL, physical = FALSE,
             samps_per_frame = NULL, dir = NULL,
             legacy = FALSE, metadata = NULL, invalid_warning = TRUE)
```

Writes signals as a new record: the header `RECORD.hea` and one signal file, `RECORD.dat`, which holds all the signals (so they share one format). Existing files of these names are replaced: both files are written to temporary files (`.NAME.XXXXXXXX.tmp`, created with owner-only permissions) in the directory of their destinations, which replace the existing files (with their permissions; 0666 less the umask for new files) only when both are complete, so that an error leaves the existing files as they were. A symbolic link stays one (the file it leads to is replaced); a destination that is not a regular file (a device, a FIFO) is written directly. The sampling frequency, the gain and ADC resolution of each signal, and the units of each calibrated signal must be given: there are no defaults. The per-signal arguments (`fmt`, `gains`, `baselines`, `adczeros`, `adcres`, `units`, `names`) take one value for all the signals, or a vector (or list) with one value per signal. All the arguments, the samples included, are checked before any file is created or changed; whatever happens, the output signals are closed (`wfdbquit`) and no temporary file is left before the function returns. Text (the record name, `names`, `units`, `comments`) is written as its bytes, one byte per character (Latin-1): in an R session in UTF-8, a string that holds a character beyond U+00FF, which would be written as several bytes, is an error.

record
: Record name (a non-empty character string), without the `.hea` extension (a trailing `".hea"` is removed, as by the readers): a plain name (`"rec"`) or a relative path (`"data/rec"`), created in the output directory if one is set, otherwise in the current directory, missing directories included; or a full path, absolute or in the home directory (`"~/Documents/rec"`), used as given. With a path, the files are created in that directory, and the header names the record and its signal file without the path (`rec`, `rec.dat`), so that the record opens under any path that leads to it (in legacy mode, as in the original, the header gives the name with its path). A URL cannot be written.

signals
: The samples: a numeric matrix with one row per sample and one column per signal; a vector is one signal (an array of more than two dimensions is an error). In ADC units (the default), the values are integers within the range of a 32-bit integer, and `NA` is written as an invalid sample; with `physical = TRUE`, they are in physical units.

fs The sampling frequency, in Hz: a finite number > 0. Required.

fmt The storage format, such as `16`, `212`, `24` or `516` (FLAC); one value, or one per signal, all equal (Chapter 15). Every sample must fit the format: within its range (Table 15.1; -2048 to 2047 for format 212, for example), and, in format 8, within -128 to 127 of the sample before it. The smallest value of a format is the value of an invalid (missing) sample: a sample equal to it, or to `WFDB_INVALID_SAMPLE`, is read back as invalid (see `invalid_warning`). A FLAC file holds at most 8 signals. Default `16`.

gains
The gain of each signal, in ADC units per physical unit: a finite number; `0` marks an uncalibrated signal, which is allowed only for samples given in ADC units. Negative gains are allowed. Required (no default).

baselines
The ADC value that corresponds to 0 physical units: a whole number within the range of a 32-bit integer. Default `NULL` (or `NA`): the ADC zero of the signal.

adczeros
The ADC value at the middle of the ADC range: a whole number within the range of a 32-bit integer. Default `0`.

adcres
The ADC resolution, in bits (a whole number ≥ 1). Required (default `NULL`: not given).

units
The physical units, such as `"mV"`, as a character vector or a list. Required for each signal whose gain is not 0. Default `NULL`: not given.

names
The signal descriptions, such as `"MLII"`, as a character vector or a list. Default `NULL`: no descriptions.

base_time
The start time, with an optional date: `"hh:mm:ss"` or `"hh:mm:ss dd/mm/yyyy"`, with optional decimals of the seconds. Default `NULL`: no start time.

comments
A character vector of info strings, written to the header as comment lines (`"#"` followed by the string). Default `NULL`: none.

physical
`TRUE` if `signals` are in physical units. They are converted to ADC units as $\mathrm{round}(x \cdot gain) + baseline$ (halves rounded away from zero); `NA` becomes an invalid sample, and infinite values and values beyond the range of 32-bit ADC units are errors. Default `FALSE`: ADC units.

samps_per_frame
For a multi-frequency record, the number of samples per frame of each signal (integers ≥ 1). `signals` then has one row per frame and `sum(samps_per_frame)` columns: the samples of signal 0, then those of signal 1, and so on. `fs` is then the frame frequency. Default `NULL`: one sample per frame for every signal.

dir The directory (which must exist) in which to create the files, for this call only; `record` must then be a plain name, without a directory, and the record opens from that directory under its plain name. Default `NULL` (or `""`): the output directory.

legacy
`TRUE` to behave as the original WFDB package: the gain defaults to 200, and the ADC resolution to one based on the format. Default `FALSE`.

metadata
Metadata entries (Chapter 5) to write to the header after the comments, as `# Key: value`

(through `putinfo`): a named list or a named character vector, such as `list(Age = "63")` or the result of `read_metadata`. Keys and values follow the rules of `set_metadata`; a `NULL` or `NA` value is an error, no two keys may differ only in letter case, and none may be the key of an entry in `comments`. Default `NULL`: none.

`invalid_warning`

`TRUE` to issue an R warning (`warning()`) when samples will be read back as invalid (missing) samples ("N sample(s) of signal K will be read back as invalid (missing) samples (format F stores an invalid sample as V)"): those equal to the invalid value of the format or to `WFDB_INVALID_SAMPLE`, other than the `NA` samples of physical units, which mark missing samples deliberately. `FALSE` for no warning, when such samples are deliberate. Default `TRUE`.

Returns `NULL`, invisibly.

Errors Stops with an error if `record` is empty ("record name must not be empty"); if `signals` has more than two dimensions ("signals must be a 1-D or 2-D array"); if `signals` has no signals ("at least one signal is required"); if `fs`, a gain, an ADC resolution or the units of a calibrated signal is missing or invalid ("fs must be specified and > 0", "fs must be finite and > 0", "gains must be specified", "gains must be finite", "ADC resolution(s) must be specified", "units must be specified"); if an ADC resolution, ADC zero or baseline is not a whole number within the range of a 32-bit integer ("adcres must be whole numbers within the range of a 32-bit integer", and the same for `adczeros` and `baselines`, in legacy mode too); if the record name, a signal name, the units or a comment has a character beyond U+00FF ("record name 'X' has a character that is not Latin-1 (one byte per character)", "signal name 'X' …", "units 'X' …", "comment 'X' …"); if `base_time` cannot be parsed ("invalid base_time 'TEXT'"); if a vector of per-signal values does not have one value per signal ("NAME must have one value per signal"); if a signal in physical units has a gain of 0 ("signal K is uncalibrated"); if a sample is not valid ("samples in ADC units must be integers", "samples in ADC units must be within the range of a 32-bit integer", or, with `physical = TRUE`, "signal K has a sample that is infinite or beyond the range of 32-bit ADC units"); if the signals do not all have the same format ("all signals of a record written by write_record must have the same format"); if a sample does not fit the format ("signal K has a sample (V) outside the range of format F (LO to HI)"; in format 8, "signal K changes by D between two samples, more than format 8 can store (-128 to 127)" or "signal K has an invalid sample, which format 8 cannot store"); if a FLAC format is given for more than 8 signals ("format 516 (FLAC) stores at most 8 signals in a file; the record has 12"); if a `samps_per_frame` value is not an integer ≥ 1 ("samps_per_frame must hold integers >= 1"), or the number of columns of `signals` is not their sum ("signals has N columns, but samps_per_frame adds up to M"); if `metadata` is not valid (an error of class `wfdbx_argument_error`, Section 10.3.7); if `record` contains a directory and `dir` is given, or `dir` does not exist ("directory DIR does not exist"); and if the signal file or the header cannot be created ("can't create signal file for record REC", "can't create header for record REC"), or cannot be written completely, for example on a full disk ("write error on record REC"), the existing files being then left as they were. The arguments are checked in the order of the Python implementation (Section 6.3.3).

```r
library(wfdbx)

# Two signals of random numbers, in millivolts, at 360 Hz: the record "rand"
# (rand.hea and rand.dat).
set.seed(1)
x <- matrix(rnorm(7200), ncol = 2)
write_record("rand", x, fs = 360, fmt = 16, physical = TRUE, gains = 1000, adcres = 16,
             units = "mV", names = c("A", "B"), base_time = "14:00:03 21/06/2024",
             comments = "random numbers", metadata = list(Generator = "rnorm"))
rec <- read_record("rand", physical = TRUE)
```

```
cat(dim(rec$signals), rec$base_time, "\n")      # 3600 2 [14:00:03 21/06/2024]
print(max(abs(rec$signals - x)) <= 0.0005)      # TRUE: rounded to 1/1000 mV
print(read_header("rand")$comments)             # "random numbers" " Generator: rnorm"

# An uncalibrated signal (gain 0, no units), in ADC units, in the directory "out".
dir.create("out", showWarnings = FALSE)
write_record("counts", 0:999, fs = 100, gains = 0, adcres = 12, dir = "out")
print(read_record("out/counts")$signals[1:3, 1])   # 0 1 2
```

A multi-frequency record: signal `fast` has 2 samples per frame, signal `slow` one; each row of `signals` is one frame, with the two samples of `fast` and then the sample of `slow`:

```
library(wfdbx)

frames <- 100                                   # 1 second at 100 frames per second
fast <- matrix(0:(2 * frames - 1), ncol = 2, byrow = TRUE)   # 200 samples per second
slow <- matrix(500, frames, 1)                  # 100 samples per second
write_record("mf", cbind(fast, slow), fs = 100, gains = 200, adcres = 12, units = "mV",
             names = c("fast", "slow"), samps_per_frame = c(2, 1))
rec <- read_record("mf", highres = TRUE)
cat(rec$fs, "\n")                               # 200
print(rec$signals[1:3, ])                       # rows 0 500, 1 500, 2 500
```

---

write_annotations

Write an annotation file

```
write_annotations(record, annotator, sample, symbol = NULL, code = NULL,
                  subtype = 0, chan = 0, num = 0, aux = NULL, fs = NULL,
                  dir = NULL, legacy = FALSE, sort = NULL)
```

Writes an annotation file, `RECORD.ANNOTATOR`, in the MIT format; an existing file of that name is replaced: the new file is written (and sorted, if needed) as a temporary file, created with owner-only permissions in the same directory, which replaces it only when it is complete, so that an error leaves it as it was. The record need not have a header. The annotation types are given as `symbol` or as `code` (Chapter 15). Each per-annotation argument is one value for all the annotations, or a vector with exactly one value per annotation. If the annotations are not in time order, the file is sorted when it is written, unless `sort = FALSE`. All the arguments are checked before the file is created or changed; whatever happens, the annotator is closed (`wfdbquit`) and no temporary file is left before the function returns.

record
: Record name (a character string), without the `.hea` extension (a trailing `".hea"` is also accepted): a plain name or a relative path (`"data/100s"`) is created in the output directory if one is set, otherwise in the current directory; a full path, absolute or in the home directory (`"~/Documents/100s"`), is used as given. The file is `RECORD.ANNOTATOR` at that place (`data/100s.qrs`). A URL cannot be written.

annotator
: Annotator name: the suffix of the file, made of letters, digits, _, - and ~.

sample
: The annotation times, in samples (a vector): integers from $-2^{63}+1$ to $2^{63}-2$ (the smallest and largest 64-bit integers are reserved by the file format), one per annotation. It may also be a data frame returned by `read_annotations`, whose columns then give the other fields (`code`, `subtype`, `chan`, `num` and `aux`) unless they are given.

symbol
The mnemonics of the annotation types, such as `"N"` or `"V"`: one per annotation, or one string for all. An unknown mnemonic becomes code 0 (`NOTQRS`). Default `NULL`: the types are given by `code`.

code
The annotation codes (`NORMAL`, `PVC`, ...), from 0 to 58 (59 to 63 are the control codes of the file format), instead of `symbol`: one per annotation, or one value for all. If both are given (as `read_annotations` returns them), each symbol must be the mnemonic of its code (an empty symbol agrees with any code). Default `NULL`.

subtype
The subtype of each annotation, stored as a signed byte (other values wrap around to $-128 \ldots 127$, as in C). Default `0`.

chan
The channel number of each annotation, stored as an unsigned byte (other values wrap around to $0 \ldots 255$). Default `0`.

num The annotator number of each annotation, stored as a signed byte. Default `0`.

aux The auxiliary text of each annotation, with `NA` where there is none: a character vector, or one text for all. Texts longer than 255 bytes are cut. A text is written as its bytes (Latin-1): in an R session in UTF-8, a character beyond U+00FF is an error. Default `NULL`: none.

fs If given (a finite number > 0), the frequency in which the times are counted, written to the file. Default `NULL`: none is written.

dir The directory (which must exist) in which to create the file, for this call only; `record` must then be a plain name, without a directory: the file is `DIR/RECORD.ANNOTATOR`. Default `NULL` (or `""`): the output directory.

legacy
`TRUE` to behave as the original WFDB package (an out-of-order file is then sorted by running the `sortann` tool). Default `FALSE`.

sort
`TRUE` to always sort the file, `FALSE` to keep the given order (without the library's advice to run `sortann`), for this call only. Default `NULL`: the environment variable `WFDBANNSORT` decides (sorting is on unless it is `0`).

Returns The status of the last `putann` (0), invisibly.

Errors Stops with an error if neither `symbol` nor `code` is given for one annotation or more ("symbol or code must be given"; with no annotations, an empty file is written); if both are given and disagree ("symbols and codes disagree at annotation K", K counting from 0); if a vector of per-annotation values does not have one value per annotation ("NAME must have one value per annotation"); if `sample`, `symbol`, `code`, `subtype`, `chan` or `num` is a matrix ("samples must be a 1-D array", "NAME must be a 1-D array"); if a time is not an integer within the range of a 64-bit integer ("annotation times must be integers within the range of a 64-bit integer"), or is -2^{63} ("annotation time T cannot be written (annotation times must be integers from -9223372036854775807 to 9223372036854775806)"); if a code is not from 0 to 58 ("annotation code C is reserved (59 to 63 are the control codes of the annotation file format)", "annotation code C is out of range (0 to 58)"); if an aux text has a character beyond U+00FF ("aux text 'X' has a character that is not Latin-1 (one byte per character)"); if a code, subtype, channel or annotator number is `NA`, `NaN` or infinite ("NAME must not be NA, NaN or infinite"); if `fs` is not finite and > 0; if `sort` is not `NULL`, `TRUE` or `FALSE`; if `record` contains a directory and `dir` is given, or `dir` does not exist; and if the annotation file cannot be created ("can't create annotator ANN

for record REC") or cannot be written completely, or the library reports an error while writing it ("write error on annotator ANN for record REC"), the existing file being then left as it was.

```
library(wfdbx)

# Three beats and a rhythm change, by symbol, with an aux string for the rhythm.
write_annotations("100s", "test", c(18, 77, 370, 662),
                  symbol = c("+", "N", "V", "N"), aux = c("(N", NA, NA, NA))
ann <- read_annotations("100s", "test")
print(ann$symbol)                       # "+" "N" "V" "N"
print(ann$aux)                          # "(N" NA NA NA

# A copy of the reference annotations: the data frame gives all the fields.
a <- read_annotations("100s", "atr")
write_annotations("100s", "copy", a)
b <- read_annotations("100s", "copy")
print(identical(b$sample, a$sample) && identical(b$symbol, a$symbol) &&
      identical(b$aux, a$aux))          # TRUE

# Annotations out of time order are sorted when the file is written (the library
# reports it on the standard error).
write_annotations("100s", "unsorted", c(500, 100), code = NORMAL)
print(read_annotations("100s", "unsorted")$sample)   # 100 500
```

10.3.4 Checking records

check_record — Check a record and its annotation files

```
check_record(record, annotators = character(0), input_dir = NULL,
             network = NULL)
```

Checks the header of a record, its signal specifications, every sample of its signal files, each segment of a multi-segment record, and the annotation files named, and reports each problem found. A problem in the record never stops with an error: it is reported, with a severity (`"error"`: the record is damaged or does not follow the WFDB format, and programs will read wrong data or fail; `"warning"`: the record can be read, but something is missing or unusual, and may give wrong results; `"info"`: a fact worth knowing, such as invalid samples). The problems, their codes and their messages are the same in the seven languages (Table 6.7). The checks run in normal mode, even if legacy mode is on, and the library prints no messages during the call; both settings are restored afterwards. They are similar to those of Benjamin Moody's `wfdbcheck`.

record
: Record name (a character string), without the `.hea` extension (a trailing `".hea"` is also accepted): a full path, absolute (`"/data/mitdb/100"`) or in the home directory (`"~/Documents/100s"`), used as given; a relative path (`"100s"`, `"data/100s"`), read from the input directory if one is set, otherwise from the current directory; or a URL (`"https://physionet.org/files/mitdb/1.0.0/100"`), read only if network access is on.

annotators
: A character vector of the annotator names whose files are checked, such as `"atr"`. Default `character(0)`: none.

input_dir
: The input directory for this call only (optional): a local directory, a ~ path or a URL, under

which relative record names are read. It does not apply to full paths or URLs. Default `NULL`: the current setting (`wfdbx_set_input_dir`).

`network`
`TRUE` to allow reading URLs for this call only, `FALSE` to forbid it. Default `NULL`: the current setting.

Returns A list of class `"wfdbx_check_report"` (Table 10.4): `report$ok` is `TRUE` if no problem is an error, `report$issues` is a data frame with one row per problem, `format(report)` gives the text form, one line per problem, and `print(report)` prints it (`no problems found` if there is none).

Errors Stops with an error if `record` is not a single string ("record must be a string") or is empty ("record must not be empty"), or if `annotators` is not a character vector without `NA` ("annotator names must be strings"). A record that cannot be read is reported (code `header_unreadable`), not raised.

```r
library(wfdbx)

report <- check_record("100s", "atr")
print(report$ok)          # TRUE: no errors
report
# warning: signal 0: the units are not specified
# warning: signal 1: the units are not specified

# A damaged record: one sample of a 12-bit signal is overwritten with 2048.
write_record("damaged", 0:99, fs = 100, gains = 100, adcres = 12, units = "mV",
             names = "ramp")
con <- file("damaged.dat", "r+b")
invisible(seek(con, 20, rw = "write"))  # sample 10 (2 bytes per sample in format 16)
writeBin(as.raw(c(0x00, 0x08)), con)    # 0x0800 = 2048, little-endian
close(con)
report <- check_record("damaged")
print(report$ok)          # FALSE
print(report$issues[, c("severity", "code", "where")])
#   severity         code    where
# 1    error sample_range signal 0
# 2    error     checksum signal 0
print(format(report)[1])
# "error: signal 0: 1 sample is outside the ADC range (-2048 to 2047): sample 10 (value 2048)"
```

10.3.5 Header metadata

These functions read and change the metadata of a header: its info strings of the form `# Key: value`, such as `# Age: 63` (Chapter 5). In short: keys are compared without regard to letter case (A-Z and a-z only); a key must not be empty and must not contain `:`, `#`, a line break or a NUL byte, and a value must not contain a line break or a NUL byte; spaces, tabs, VT, FF and CR at either end of keys and values are removed. Keys and values are handled as bytes: a character string is taken as its bytes, never re-encoded, and a raw vector is also accepted, as given. The info strings are the lines that begin with `#` after the record line, which is found as the header parser of the library finds it, in every mode: the first line that holds something other than spaces, tabs, CRs and LFs and does not begin (after them) with `#`; comments and empty or blank lines before it are not info strings. (The C function `getinfo` takes the first line that does not begin with `#`, so an empty first line, for the record line; in WFDB-x, `getinfo`, the comments of `read_header` and `read_record` and the notes of `wfdbdesc` follow the header parser, except in legacy mode.) A change rewrites the header in one step, keeping every other byte of it: the new header is written to a temporary file in the same directory, created with owner-only permissions, which is then

renamed over the header and given its permission bits. All the arguments are checked first, so an error leaves the header unchanged, and nothing is written if nothing changes (unless `dir` is given). Only the `.hea` file is read (not a `.info` file); for a multi-segment record, it is the master header, unless a segment is named. The output directory is not used: a change is written to the header that was read, or to `dir`.

The entries are a named character vector: the names are the keys and the elements the values, in file order, with the spelling of the file, such as `c(Age = "63", Sex = "F")`. Since reading returns each key once, `md[["Age"]]` gives a value (with the file's spelling of the key; `get_metadata` ignores letter case). The same form, or a named list, is what `update_metadata` and the argument `metadata` of `write_record` take, so the entries of one header can be written to another.

Metadata often holds information about patients: WFDB-x stores it as plain text, exactly as given, and does not de-identify it.

> **Patient data** A header can hold identifying data about a patient (a name, a date of birth, a medical record number). The metadata functions read and write such data as given. Records that are shared must be de-identified first.

`read_metadata` — Read the metadata of a record

```
read_metadata(record, input_dir = NULL, network = NULL)
```

Reads the metadata entries of the header of a record, in file order. An info string is an entry if it contains a colon: the key is the text before the first colon, the value the rest of the line, both without spaces at either end. Info strings without a colon are free comments, which are not returned.

`record`
: Record name (a character string), without the `.hea` extension (a trailing `".hea"` is also accepted): a full path, absolute (`"/data/mitdb/100"`) or in the home directory (`"~/Documents/100s"`), used as given; a relative path (`"100s"`, `"data/100s"`), read from the input directory if one is set, otherwise from the current directory; or a URL (`"https://physionet.org/files/mitdb/1.0.0/100"`), read only if network access is on.

`input_dir`
: The input directory for this call only (optional): a local directory, a ~ path or a URL, under which relative record names are read. It does not apply to full paths or URLs. Default `NULL`: the current setting (`wfdbx_set_input_dir`).

`network`
: `TRUE` to allow reading URLs for this call only, `FALSE` to forbid it. Default `NULL`: the current setting.

Returns A named character vector of the entries, in file order (an empty named character vector if there are none). A key that appears more than once (ignoring letter case) is returned once, with its first value, and an R warning is issued (Section 10.3.7).

Errors An error of class `wfdbx_file_error` if the header cannot be found or read ("can't read header for record REC"); of class `wfdbx_argument_error` if `record` is empty or `"-"` (the standard input).

```
library(wfdbx)
```

```
write_record("pat", rep(0, 360), fs = 360, gains = 200, adcres = 12, units = "mV",
             names = "ECG", comments = "recorded at rest",
             metadata = list(Age = "63", Sex = "F"))
print(read_metadata("pat"))             # c(Age = "63", Sex = "F")
print(read_header("pat")$comments)
# "recorded at rest" " Age: 63" " Sex: F": the free comment is not an entry
```

---

get_metadata

Return the value of a metadata entry

```
get_metadata(record, key, input_dir = NULL, network = NULL)
```

Returns the value of one metadata entry of the header of a record.

record
: Record name (as for `read_metadata`).

key
: The key, such as `"Age"`: a character string or a raw vector. Letter case is ignored, and spaces at either end are removed.

input_dir
: The input directory for this call only. Default `NULL`: the current setting.

network
: `TRUE` to allow reading URLs for this call only, `FALSE` to forbid it. Default `NULL`: the current setting.

Returns The value, a character string (`""` for an entry with an empty value), or `NULL` if the header has no entry for `key`. If it has more than one, the first is returned, and an R warning is issued.

Errors An error of class `wfdbx_file_error` if the header cannot be found or read; of class `wfdbx_argument_error` if `record` is empty or `"-"`, or if `key` is not a valid key or not a string.

```
library(wfdbx)

write_record("pat", rep(0, 360), fs = 360, gains = 200, adcres = 12, units = "mV",
             metadata = c(Age = "63", Sex = "F"))
print(get_metadata("pat", "age"))       # "63"
print(get_metadata("pat", "Height"))    # NULL
```

---

set_metadata

Set the value of a metadata entry

```
set_metadata(record, key, value, input_dir = NULL, network = NULL, dir = NULL)
```

Sets the value of a metadata entry in the header of a record. If the header has an entry for `key` (ignoring letter case), its line is replaced by `# Key: value`, at the same position, with the spelling of `key` given here; otherwise the entry is added after the last info string of the header (or, if it has none, after its last line that is not blank). The entry is written as `# Key: value`, or `# Key:` if the value is empty. Every other line is kept, byte for byte, including its line ending.

record
: Record name (as for `read_metadata`). The header is changed where it was read, unless `dir` is given; a header read from a URL can only be written elsewhere, with `dir`.

key The key, such as `"Age"` or `"Date of birth"`: a character string or a raw vector. Spaces at either end are removed. It must not be empty, and must not contain `:`, `#`, a line break or a NUL byte.

value
The value: a character string, which may be a sentence, or a raw vector. Spaces at either end are removed. It must not contain a line break or a NUL byte; it may be empty.

input_dir
The input directory in which the header is read, for this call only. Default `NULL`: the current setting.

network
`TRUE` to allow reading the header from a URL, for this call only, `FALSE` to forbid it. Default `NULL`: the current setting.

dir A directory, which must exist, in which to write the changed header, under the same file name, instead of changing the header itself; the signal files are not copied. The header is then written even if nothing changes. Default `NULL` (or `""`): the header itself is changed.

Returns `NULL`, invisibly.

Errors An error of class `wfdbx_file_error` if the header cannot be read, if the changed header cannot be written ("can't write header for record REC"), or if `dir` does not exist ("directory DIR does not exist"); of class `wfdbx_argument_error` if `key` or `value` is not valid or not a string, if the header has more than one entry for `key` ("header of record REC has more than one entry for key KEY (lines L1, L2); remove them first"), if it has no record line, if it was read from a URL and `dir` is not given, or if `record` is empty or `"-"`.

```
library(wfdbx)

write_record("pat", rep(0, 360), fs = 360, gains = 200, adcres = 12, units = "mV",
             metadata = c(Age = "63", Sex = "F"))
set_metadata("pat", "AGE", "64")                                  # replaces "# Age: 63"
set_metadata("pat", "Diagnosis", "sinus rhythm, no ectopy")       # a new entry
print(read_metadata("pat"))
# c(AGE = "64", Sex = "F", Diagnosis = "sinus rhythm, no ectopy")

# Write the changed header to another directory; pat.hea is left unchanged.
dir.create("changed", showWarnings = FALSE)
set_metadata("pat", "Sex", "M", dir = "changed")
cat(get_metadata("pat", "Sex"), get_metadata("changed/pat", "Sex"), "\n")   # F M
```

remove_metadata

Remove a metadata entry

```
remove_metadata(record, key, input_dir = NULL, network = NULL, dir = NULL)
```

Removes every entry for `key` (ignoring letter case) from the header of a record, so it also repairs a header that has more than one. Other lines are kept, byte for byte. Nothing is written if the header has no entry for `key`, unless `dir` is given.

record
Record name (as for `set_metadata`).

key The key (as for `set_metadata`).

input_dir
: The input directory in which the header is read, for this call only. Default `NULL`: the current setting.

network
: `TRUE` to allow reading the header from a URL, for this call only, `FALSE` to forbid it. Default `NULL`: the current setting.

dir A directory, which must exist, in which to write the changed header instead of changing the header itself. Default `NULL` (or `""`): the header itself is changed.

Returns The number of lines removed (a number; 0 if the header has no entry for `key`).

Errors As `set_metadata`, except that more than one entry for `key` is not an error.

```
library(wfdbx)

write_record("pat", rep(0, 360), fs = 360, gains = 200, adcres = 12, units = "mV",
             metadata = c(Age = "63", Sex = "F"))
print(remove_metadata("pat", "sex"))    # 1
print(remove_metadata("pat", "Sex"))    # 0: there is no such entry now
print(read_metadata("pat"))             # c(Age = "63")
```

---

update_metadata

Make several changes to the metadata

```
update_metadata(record, changes, input_dir = NULL, network = NULL, dir = NULL)
```

Makes several changes to the metadata of a record, in one write of the header. Each change is applied in order, as `set_metadata` does, or, for a value of `NULL` or `NA`, as `remove_metadata` does. All the changes are checked first: if one is not valid, none is made.

record
: Record name (as for `set_metadata`).

changes
: A named list or a named character vector: the names are the keys, and each element a value (a character string or a raw vector), or `NULL` or `NA` to remove the entry, as in `list(Age = "64", Dx = NULL)` or `c(Age = "64", Dx = NA)`. No two keys may differ only in letter case. An empty list or vector makes no change.

input_dir
: The input directory in which the header is read, for this call only. Default `NULL`: the current setting.

network
: `TRUE` to allow reading the header from a URL, for this call only, `FALSE` to forbid it. Default `NULL`: the current setting.

dir A directory, which must exist, in which to write the changed header instead of changing the header itself. Default `NULL` (or `""`): the header itself is changed.

Returns `NULL`, invisibly.

Errors As `set_metadata`; also an error of class `wfdbx_argument_error` if two keys of `changes` differ only in letter case, or if `changes` is not a named list or a named character vector (`NULL` included).

```
library(wfdbx)
```

```
write_record("pat", rep(0, 360), fs = 360, gains = 200, adcres = 12, units = "mV",
             metadata = c(Age = "63", Sex = "F"))
update_metadata("pat", list(Age = "64", Sex = NULL, Dx = "164889003"))
print(read_metadata("pat"))             # c(Age = "64", Dx = "164889003")
tryCatch(update_metadata("pat", c(Dx = "426783006", dx = NA)),
         wfdbx_argument_error = function(e) print(conditionMessage(e)))
# "metadata key 'dx' is given more than once (keys are compared without regard to letter case)"
print(get_metadata("pat", "Dx"))        # "164889003": nothing was changed
```

10.3.6 The results

The readers return plain R objects: `read_record` a list (Table 10.1), `read_header` a list of class `"wfdbx_header"` (Table 10.2), `read_annotations` a data frame (Table 10.3), `read_metadata` a named character vector (Section 10.3.5), and `check_record` a list of class `"wfdbx_check_report"` (Table 10.4). The signals are described by `siginfo` lists (Table 10.5), which are those of the WFDB library functions (Section 10.4). The elements of these lists are listed in the order in which they appear.

Table 10.1 The elements of the list returned by `read_record`

Element	Contents
`name`	the record name, as given to `read_record`
`fs`	the sampling frequency in Hz, or 0 if the header gives none; with `highres = TRUE`, the highest sampling frequency of a multi-frequency record
`signals`	a numeric matrix with one row per sample and one column per signal read: ADC units, or physical units with `physical = TRUE`; invalid samples are `WFDB_INVALID_SAMPLE`, or `NA` in physical units
`siginfo`	a list of `siginfo` lists, one per signal read, in the order of the columns; `desc` has no white space at either end, and `adcres` is 0 if the header does not give it
`names`	the description of each signal (a character vector), such as `"MLII"`, without the white space that a header may put at either end of it
`units`	the physical units of each signal, as a list, with `NULL` where the header gives none
`gains`	the gain of each signal, in ADC units per physical unit; 0 means that the signal is uncalibrated
`baselines`	the ADC value of each signal that corresponds to 0 physical units
`base_time`	the start time and date as text, to the second, such as `"[14:00:03 21/06/2024]"` (`"[14:00:03]"` without a date), or `NULL` if the header gives none
`base_time_ms`	the start time in milliseconds after midnight, such as `50403500` for 14:00:03.500, or `NULL` if `base_time` is `NULL`
`base_date`	the start date as an integer vector `c(year, month, day)`, or `NULL` if the header gives none
`sampfrom`	the number of the first sample returned
`comments`	a character vector of the info strings of the header: its lines that begin with `#` after the record line (found as the header parser finds it, Section 10.3.5), without the `#`
`checksum_ok`	`FALSE` if the samples do not match the checksums of the header; checksums are checked only when a read reaches the end of the record, so a shorter read gives `TRUE`

Table 10.2 The elements of the list of class `"wfdbx_header"` returned by `read_header`

Element	Contents
`name`	the record name, as given to `read_header`

Element	Contents
`fs`	the sampling frequency in Hz, or 0 if the header gives none
`counter_freq`	the counter frequency, as `getcfreq()` returns it (the sampling frequency if the header gives none)
`base_counter`	the base counter value, as `getbasecount()` returns it
`nsamples`	the number of samples per signal, or 0 if the header does not give it
`nsig`	the number of signals
`siginfo`	a list of `siginfo` lists, one per signal; `desc` has no white space at either end, and `adcres` is 0 if the header does not give it
`base_time`, `base_time_ms`, `base_date`	the start time and date, as in the result of `read_record`
`comments`	a character vector of the info strings of the header: its lines that begin with `#` after the record line (found as the header parser finds it, Section 10.3.5), without the `#`
`metadata`	the metadata entries of the header, as `read_metadata` returns them: a named character vector, in file order (empty for an EDF file and for the standard input)

Table 10.3 The columns of the data frame returned by `read_annotations`, one row per annotation, in file order

Column or attribute	Contents
`sample`	the annotation times, in samples
`symbol`	the mnemonics of the codes, such as `"N"`, `"V"` or `"+"` (as `annstr` gives them)
`code`	the annotation codes, such as `NORMAL` or `PVC`
`subtype`	the subtype of each annotation
`chan`	the channel number of each annotation
`num`	the annotator number of each annotation
`aux`	the auxiliary texts of the annotations, with `NA` where there is none
`attr(, "record")`	the record name, as given
`attr(, "annotator")`	the annotator name, such as `"atr"`
`attr(, "fs")`	the frequency in which `sample` is counted, normally the sampling frequency of the record; 0 if neither the record, nor the file, nor the argument `fs` of `read_annotations` gives one

Table 10.4 The list of class `"wfdbx_check_report"` returned by `check_record`, and its methods

Element or method	Contents
`record`	the record name, as given
`ok`	`TRUE` if no issue has the severity `"error"`
`issues`	a data frame with one row per problem, in the order of the checks, and the columns `severity` (`"error"`, `"warning"` or `"info"`), `code` (a stable identifier of the kind of problem, such as `"checksum"` or `"gain_missing"`, Table 6.7), `where` (`""` for the record, `"signal N"`, `"segment SEG"`, `"segment SEG, signal N"` or `"annotator ANN"`) and `message`
`format(report)`	the text form of the report, as a character vector: one line per issue, `"SEVERITY: WHERE: MESSAGE"` (`"SEVERITY: MESSAGE"` when `where` is empty), or the one line `no problems found`
`print(report)`	prints the lines of `format(report)`, and returns the report invisibly

Table 10.5 The elements of a `siginfo` list, the specification of a signal

Element	Contents
`fname`	the name of the signal file, as the header gives it
`desc`	the signal description, such as `"MLII"`, or `NULL`
`units`	the physical units, such as `"mV"`, or `NULL` if not specified
`gain`	the gain, in ADC units per physical unit (0: uncalibrated)
`initval`	the value of the first sample
`group`	the signal group number (the signals of one group share one file)
`fmt`	the storage format, such as `16`, `212` or `516`
`spf`	the number of samples per frame (more than 1 for the faster signals of a multi-frequency record)
`bsize`	the block size, in bytes (0 for ordinary files)
`adcres`	the ADC resolution, in bits (0: not specified)
`adczero`	the ADC output for an input of 0 V
`baseline`	the ADC value that corresponds to 0 physical units
`nsamp`	the number of samples (0 if unknown)
`cksum`	the 16-bit checksum of the samples

10.3.7 Warnings and errors

Errors. The high-level functions report errors as R errors (conditions of class `"error"`, raised by `stop`), whose messages are given in the entries of this section; `tryCatch` handles them. Each function checks its arguments before it creates or changes a file, so an invalid argument leaves the files as they were; the writers replace existing files only when the new ones are complete, so an error while writing leaves them as they were too. A signal file that ends before the length its header gives is an error of `read_record` ("getvec status -3"), never hidden by a checksum error, and a problem (`length_short`) in the report of `check_record`. The library may also print a message on the standard error when it cannot open a file (as in `init: can't open header for record REC`); `wfdbquiet()` turns these messages off (Section 10.4).

The errors of the metadata functions, and those of the argument `metadata` of `write_record`, also have a class that tells their kind (Table 10.6), so that a program can handle each kind on its own, with `tryCatch(..., wfdbx_file_error = function(e) ...)`.

Table 10.6 The classes of the errors of the metadata functions

Class	Raised when
`wfdbx_file_error`	a header cannot be found, read or written, or the directory given as `dir` does not exist (the errors that Python reports as `IOError`)
`wfdbx_argument_error`	an argument is invalid (a record name that is empty or `"-"`, a key or value that is not valid or not a string, changes or metadata that are not a named list or named character vector, two keys that differ only in letter case), or the header does not allow the change (more than one entry for the key, no record line, a header read from a URL without `dir`); Python's `ValueError` and `TypeError`

Both classes are also of class `"error"` (and `"condition"`), so `tryCatch(..., error = )` catches them too.

The checksum warning. When `read_record` reads to the end of a record whose samples do not match the checksums of its header, it returns all the samples and issues an R warning: "checksum

error in record REC: the signal data do not match the checksums in its header". `checksum_warning = FALSE` turns the warning off; `checksum_ok` of the result is `FALSE` either way.

The duplicate-key warning. When a header holds two metadata entries whose keys differ only in letter case (written by other software), `read_metadata`, `get_metadata` and `read_header` use the first and issue an R warning: "header of record REC has more than one entry for key KEY (lines L1, L2); the first is used". `set_metadata` and `update_metadata` then refuse to change that key (an error of class `wfdbx_argument_error`), and `remove_metadata` removes all of its entries.

Both warnings are issued with `warning()`, without the call. As any other R warning, they can be handled (`withCallingHandlers`), silenced (`suppressWarnings`) or turned into errors (`options(warn = 2)`):

```r
library(wfdbx)

# A record whose last sample is changed after it was written: the checksum fails.
write_record("ck", 0:99, fs = 100, gains = 100, adcres = 12, units = "mV")
con <- file("ck.dat", "r+b")
invisible(seek(con, 198, rw = "write"))
writeBin(as.raw(c(0, 0)), con)
close(con)
w <- NULL
rec <- withCallingHandlers(read_record("ck"), warning = function(c) {
  w <<- conditionMessage(c)
  invokeRestart("muffleWarning")
})
cat(dim(rec$signals), rec$checksum_ok, "\n")    # 100 1 FALSE: all the samples
print(w)
# "checksum error in record ck: the signal data do not match the checksums in its header"
rec <- read_record("ck", checksum_warning = FALSE)      # no warning

# Two entries for one key: the first is used, with a warning.
write_record("dup", rep(0, 10), fs = 100, gains = 0, adcres = 12,
             comments = c(" Age: 63", " AGE: 64"))
print(suppressWarnings(get_metadata("dup", "age")))     # "63"
tryCatch(set_metadata("dup", "Age", "65"),
         wfdbx_argument_error = function(e) print(conditionMessage(e)))
# "header of record dup has more than one entry for key Age (lines 3, 4); remove them first"
print(remove_metadata("dup", "age"))                    # 2
```

```r
library(wfdbx)

tryCatch(read_record("nosuchrecord"), error = function(e) print(conditionMessage(e)))
# "can't read record nosuchrecord" (after the library's message on the standard error)

write_record("raw", 0:9, fs = 100, gains = 0, adcres = 12)      # uncalibrated
tryCatch(read_record("raw", physical = TRUE), error = function(e) print(conditionMessage(e)))
# "signal 0 of record raw is uncalibrated"

# The metadata functions: the class tells the kind of error.
e <- tryCatch(read_metadata("nosuchrecord"), error = function(e) e)
print(class(e))           # "wfdbx_file_error" "error" "condition"
e <- tryCatch(set_metadata("raw", "a:b", "1"), error = function(e) e)
print(class(e))           # "wfdbx_argument_error" "error" "condition"
print(conditionMessage(e))   # "metadata key 'a:b' must not contain ':'"
```

10.4 The WFDB library functions in R

Every function of the WFDB library is available under its C name, with the same arguments, in the same order, and the same meaning, so that programs and documentation written for the C library carry over directly. The functions, their arguments and their results are documented once, for all the languages, in Chapter 13; this section gives the rules by which the C declarations become R functions. The help pages `?isigopen`, `?timstr`, `?annopen`, `?annstr`, `?calopen`, `?getwfdb`, `?ecgcodes`, `?isann` and `?siginfo` list the functions of each part of the library, with the result of each.

10.4.1 Conventions

Names. The functions, the constants (`WFDB_READ`, `WFDB_INVALID_SAMPLE`, `WFDB_HIGHRES`, ...) and the annotation codes (`NORMAL`, `PVC`, ..., Chapter 15; `ecgcodes()` returns them as a list) keep their C names. The macros of `<wfdb/ecgmap.h>` are functions, under both their macro names and the names of the functions behind them: `isqrs` and `wfdb_isqrs`, `map1` and `wfdb_map1`, and so on; they take vectors. The C function `sample` is `wfdb_sample`, so that it does not mask R's `sample`.

Numbers are doubles. Signal, group and annotator numbers start at 0, as in C (signal `s` is element `s + 1` of an R vector). C's integer arithmetic (wrap around, casts to `short` or `char`) is applied where C applies it.

NULL is `NULL`. A `NULL` pointer argument or result is `NULL`: `sampfreq(NULL)` returns the frequency of the open record, `getinfo(NULL)` continues with the info strings of the last record, and `getinfo` returns `NULL` after the last one. A list element that is `NULL` is set with `x["f"] <- list(NULL)` (`x$f <- NULL` removes it).

Structures are lists with the C field names, made by the constructors `siginfo()`, `anninfo()`, `annotation()`, `calinfo()` and `seginfo()` (Section 10.4.2). An array of structures is a list of them: `annopen(record, list(anninfo("atr", WFDB_READ)))`, `osigfopen(list(siginfo(...), ...))`.

Arrays and output arguments. R functions do not change their arguments: where a C function fills an array or a structure that the caller passes, the R function returns it instead, in a list with the C return value:

```
r <- isigopen(record, nsig)      # r$nsig: the C result; r$si: the siginfo lists
r <- osigopen(record, nsig)      # the same
g <- getvec()                    # g$stat: the C result; g$v: one sample per signal
g <- getframe()                  # g$stat; g$v: one sample per sample of a frame
g <- getann(an)                  # g$stat; g$annot: the annotation (NULL on errors)
g <- getcal(desc, units)         # g$stat; g$cal: the calinfo list (NULL if none)
r <- wfdbinit(record, aiarray, nsig)   # r$stat; r$si
s <- getseginfo()                # a list of seginfo lists (empty: not multi-segment)
```

With `nsig` ≤ 0, `isigopen` reads the header only: `r$nsig` is then the number of signals of the record, and `r$si` the specifications of up to `-nsig` of them. The arguments that C takes only to receive results are left out (`getvec()` has none), and the lengths of arrays have defaults: `osigfopen(siarray, nsig = length(siarray))`, `setheader(record, siarray, nsig = length(siarray))`, `setmsheader(record, segment_name, nsegments = length(segment_name))`, `annopen(record, aiarray, nann = NULL)`. `putvec(vector)`, `putann(an, annot)`, `ungetann(an, annot)` and `putcal(cal)` take the values to write.

Strings are character strings, handled as bytes (ISO-8859-1, never re-encoded), so every byte sequence round-trips unchanged; functions that take a string also take a raw vector. The `aux`

element of an annotation is `NULL`, or a raw vector whose first byte is the length of the text that follows, as in C; `aux_string(aux)` returns the text (up to its first NUL byte), or `NULL`.

Return values and errors. The functions return what the C functions return: a status (negative on errors), a number of signals, a frequency, a string or `NULL`. Errors are reported by these values and by a message on the standard error, as in C, never by R errors. `wfdberror()` returns the last message; `wfdbquiet()` turns the messages off and `wfdbverbose()` on again; `wfdb_error(fmt, ...)` reports a message of a program's own in the same way.

State. As in C, the library keeps its state in the R session: the open record, its input signals, the output signals and the annotators remain open between calls, until `wfdbquit()` closes them. `wfdbquit()` leaves the input and output directories and the network setting as they are. The high-level functions of Section 10.3 call `wfdbquit()`, which closes anything opened with the library functions.

Additions. `getframes(n, minstat = 0)` reads up to `n` sample vectors at once (`n` may be `Inf`), as `n` calls of `getvec` would, but decoding whole blocks of the signal files. It returns `list(v, stat)`: a matrix with one row per vector read and one column per signal, and the `getvec` status of the first vector that could not be read, or the number of signals if all `n` were read; with `minstat`, it stops at the first vector for which `getvec` returns less than `minstat`. `putframes(m)` writes the rows of a matrix, as `putvec` on each row. `sample_valid()` returns the validity flag of the last `wfdb_sample`.

Speed. Reading sample by sample with `getvec` in an R loop is much slower than reading in blocks: `read_record` and `getframes` decode whole blocks with vectorized code, and are the way to read large records (in chunks, with `sampfrom` and `sampto`, for a record too large for memory).

10.4.2 The structures

The structures of the C library are lists whose elements have the names of the fields of the C structures (Table 10.7). Their constructors take the elements as arguments, in this order, all with defaults:

```
siginfo(fname = NULL, desc = NULL, units = NULL, gain = 0,
        initval = 0, group = 0, fmt = 0, spf = 1, bsize = 0,
        adcres = 0, adczero = 0, baseline = 0, nsamp = 0,
        cksum = 0)
anninfo(name = NULL, stat = WFDB_READ)
annotation(time = 0, anntyp = 0, subtyp = 0, chan = 0, num = 0,
           aux = NULL)
calinfo(low = 0, high = 0, scale = 0, sigtype = NULL,
        units = NULL, caltype = 0)
seginfo(recname = "", nsamp = 0, samp0 = 0)
```

For a list of `n` structures, `replicate(n, siginfo(), simplify = FALSE)`.

Table 10.7 The lists of the C structures

Constructor (C structure)	Elements
`siginfo` (`WFDB_Siginfo`)	`fname`, `desc`, `units`, `gain`, `initval`, `group`, `fmt`, `spf`, `bsize`, `adcres`, `adczero`, `baseline`, `nsamp`, `cksum` (Table 10.5)
`seginfo` (`WFDB_Seginfo`)	`recname` (the name of the segment), `nsamp` (its number of samples), `samp0` (the number of its first sample in the record)

Constructor (C structure)	Elements
anninfo (WFDB_Anninfo)	name (the annotator name, the suffix of the file), stat (WFDB_READ or WFDB_WRITE for the MIT format, WFDB_AHA_READ or WFDB_AHA_WRITE for the AHA format)
annotation (WFDB_Annotation)	time (in sample intervals), anntyp (the annotation code), subtyp, chan, num, aux (NULL, or a raw vector whose first byte is the length of the text that follows)
calinfo (WFDB_Calinfo)	low, high (the levels of the calibration pulse, in physical units), scale (physical units per cm), sigtype (matched against the beginning of a signal description), units, caltype (WFDB_AC_COUPLED or WFDB_DC_COUPLED, or-ed with WFDB_CAL_SQUARE, WFDB_CAL_SINE or WFDB_CAL_SAWTOOTH)

10.4.3 The settings functions

The functions of Table 10.8 are additions of WFDB-x to the library: they set where files are read and written, network access and legacy mode, for the high-level functions, the library functions and the tools alike (Chapters 3 and 4). Each set function returns the previous setting, invisibly, so that it can be restored. The C function setwfdb(p) sets the input directory too, and getwfdb() returns it; in legacy mode they set and return the WFDB path, as in C.

Table 10.8 The settings functions

Function	Effect and result
wfdbx_set_input_dir(dir)	sets the input directory: a local directory, a ~ path or a URL; NULL or "" restores the default (WFDBX_INPUT_DIR, or else the current directory). Returns the previous input directory, or NULL if none was set
wfdbx_get_input_dir()	returns the input directory, or NULL if none is set
wfdbx_set_output_dir(dir)	sets the output directory: a local directory or a ~ path (a name that begins with a URL scheme, such as https:// or ftp://, is refused, with a message, and the setting is unchanged); NULL or "" restores the default (WFDBX_OUTPUT_DIR, or else the current directory). Returns the previous output directory, or NULL
wfdbx_get_output_dir()	returns the output directory, or NULL if none is set
wfdbx_set_network(on)	turns network access on (TRUE or 1) or off (FALSE or 0); NULL or a negative number restores the default (on if WFDBX_NETWORK is a nonzero number). Returns the previous setting, 1 or 0
wfdbx_get_network()	returns 1 if network access is on, 0 otherwise
wfdbx_set_legacy(on)	turns legacy mode on (TRUE) or off, for the whole library. Returns the previous setting, TRUE or FALSE
wfdbx_get_legacy()	returns TRUE if legacy mode is on, FALSE otherwise
wfdbx_legacy_args(argv)	if argv[2] (argv: the argument vector, program name first) is "--legacy", turns legacy mode on; returns list(argv, legacy): the arguments without "--legacy", and whether it was found
wfdbx_time_needs_freq(string)	returns 1 if strtim(string) needs a sampling frequency, 0 otherwise
wfdbx_write_errors()	returns the number of write errors since the R session started (output files that could not be written, flushed or closed)
wfdbx_note_write_error()	counts a write error, for a program that writes a file itself

Function	Effect and result
`wfdbx_input_path(name)`	returns the path where a program finds a file that it reads itself, as `wfdb_open` finds a file of that name: in normal mode, a relative name under the input directory, if one is set, a ~ name expanded, an absolute name as given; in legacy mode, the name as given. `""` and `"-"` are returned as given; `NULL` for `NULL`, an `http://` or `https://` URL, or, in normal mode, a name that begins with another URL scheme (also once the input directory is prefixed to it)
`wfdbx_output_path(name)`	returns the path where a program creates a file that it writes itself, as `wfdb_open` creates a file of that name: in normal mode, a relative name under the output directory, if one is set, a ~ name expanded, an absolute name as given; in legacy mode, any name under the output directory, if one is set, as a plain prefix. When the output directory is prefixed, the missing directories of the path are created. `""` and `"-"` are returned as given; `NULL` for `NULL`, an `http://` or `https://` URL, or, in normal mode, a name that begins with another URL scheme

10.4.4 Examples

Reading the samples and the annotations of a record, as a C program does (the record `100s` is read from the current directory, a copy of `tests/data`):

```
library(wfdbx)

r <- isigopen("100s", 2)                # open 2 signals
nsig <- r$nsig
cat(nsig, sapply(r$si, function(s) paste(s$desc, s$fmt, s$gain)), "\n")
# 2 MLII 212 200 V5 212 200
print(sampfreq(NULL))                   # 360: the frequency of the open record

stat <- isigsettime(strtim("0:30"))     # skip to 30 seconds
g <- getvec()                           # list(stat, v)
cat(g$stat, g$v, sapply(0:(nsig - 1), function(k) aduphys(k, g$v[k + 1])), "\n")
# 2 947 968 -0.385 -0.28

f <- getframes(360)                     # the next second, as a matrix
cat(dim(f$v), f$stat, "\n")             # 360 2 2

stat <- annopen("100s", list(anninfo("atr", WFDB_READ)))
n <- 0
while ((g <- getann(0))$stat == 0) {    # 0: an annotation was read; -1: the end
  if (n < 2) cat(mstimstr(g$annot$time), annstr(g$annot$anntyp),
                 aux_string(g$annot$aux), "\n")
  n <- n + 1
}
cat(n, "annotations\n")                 # 75 annotations

wfdbquiet()                             # no messages on the standard error
cat(isigopen("nosuchrecord", 0)$nsig, wfdberror())
# -1 init: can't open header for record nosuchrecord
wfdbverbose()
stat <- wfdbquit()                      # close everything
```

Writing a record and an annotation file, sample by sample and annotation by annotation:

```
library(wfdbx)

out <- list(siginfo(fname = "ramp.dat", desc = "ramp", units = "mV", gain = 200,
                    fmt = 16, adcres = 12))
print(osigfopen(out))                    # 1: one output signal
stat <- setsampfreq(250)
for (k in 0:499) putvec(k)
print(newheader("ramp"))                 # 0: ramp.hea is written
stat <- annopen("ramp", list(anninfo("qrs", WFDB_WRITE)))
for (t in c(100, 350)) putann(0, annotation(time = t, anntyp = NORMAL))
stat <- wfdbquit()                       # completes ramp.dat and ramp.qrs

print(readLines("ramp.hea", n = 1))              # "ramp 1 250 500"
print(read_annotations("ramp", "qrs")$sample)    # 100 350
```

10.5 The command-line tools in R

The 33 command-line tools (`rdsamp`, `rdann`, `wrsamp`, `xform`, ...) are run by the driver script `wfdbx.R` with `Rscript`, followed by the name of the tool and its arguments. The tools, their options, their output and their exit status are those of the C tools, and are documented in Chapter 14. In a copy of the repository the script is `r/inst/bin/wfdbx.R`, which runs without the package being installed; in an installed package its path is `system.file("bin", "wfdbx.R", package = "wfdbx")`. A shell alias makes it convenient:

```
alias wfdbx='Rscript /path/to/wfdb-x/r/inst/bin/wfdbx.R'
export WFDBX_INPUT_DIR=/path/to/wfdb-x/tests/data   # the directory of 100s
wfdbx rdsamp -r 100s -t 10 -p     # the first 10 seconds of 100s, in physical units
wfdbx rdann -r 100s -a atr        # its annotations
wfdbx rdsamp --legacy -r 100s     # legacy mode: --legacy as the first argument
```

The driver script also takes these options, instead of a tool:

Table 10.9 The options of the driver script `wfdbx.R`

Command	Effect
`Rscript wfdbx.R TOOL [ARGUMENTS ...]`	runs the tool `TOOL`, and exits with its status
`Rscript wfdbx.R --list`	lists the names of the tools, one per line
`Rscript wfdbx.R --version`	prints the version, `WFDB-x 1.2.3`
`Rscript wfdbx.R --help`, `Rscript wfdbx.R -h`	prints the usage

Without arguments, the script prints the usage and exits with status 1; an unknown tool is an error ("wfdbx: unknown tool 'TOOL' (try 'wfdbx --list')", status 1), and an R error in a tool is reported as `wfdbx: error: MESSAGE` (status 1; with the environment variable `WFDBX_DEBUG` set, the call stack follows). When an output file or the standard output cannot be written (a full disk, for example), a tool reports it ("PROGRAM: write error on standard output") and exits with status 1 (with `--legacy`, it keeps the original exit status). The files that a tool reads or creates itself follow the input and output directories, as the records do: the `-i` file of `wrsamp`, `edf2mit`, `wav2mit`, `a2m` and `ad2m`, the input files of `ahaecg2mit` and the `-S` script of `xform` are read from the input directory, and the WAV file of `mit2wav`, the EDF file of `mit2edf`, the master header of `snip -m` and the `.des`, `.ecg`, `.key` and `.txt` files of `parsecp` are created in the output directory (`wfdbx_input_path` and `wfdbx_output_path`, Table 10.8); such a file cannot be read from a URL. The script runs the tools in the C locale, with R's warnings turned off, so that their output is that of the C tools.

The start-up cache. When run from a copy of the repository, the script keeps a compiled copy of the sources in `~/.cache/R/wfdbx/` (or under `$XDG_CACHE_HOME`), so that a tool starts quickly; `WFDBX_NO_CACHE=1` turns this off.

Windows. The tools write their output in binary through `/dev/stdout` and `/dev/stderr`. On Windows these do not exist, so the output is written as text and NUL bytes are lost.

The tools from R. `wfdbx_tool` runs a tool in the R session, and `wfdbx_tools` lists them.

---

wfdbx_tool

Run a command-line tool

```
wfdbx_tool(tool, args = character(0))
```

Runs one of the command-line tools with the given arguments, as the command line would, in the R session. The output of the tool goes directly to the standard output and error of the R process, in binary, and a graphical console may not show it. While the standard output of R is diverted by `sink` or `capture.output`, the standard output of the tool goes there instead, as Latin-1 text (one character per byte; NUL bytes, which an R string cannot hold, are lost; until WFDB-x 1.2.3 it went past the diversion). The standard error of the tool still goes to that of the process. To capture the output byte for byte, the tool is run in a separate process with the driver script, as in `system2("Rscript", c(driver, "rdsamp", ...), stdout = TRUE)`.

tool
: The name of the tool, a character string (one of `wfdbx_tools()`).

args
: The command-line arguments of the tool, as a character vector (one element per argument, without the program name). Default `character(0)`: none.

Returns The exit status of the tool (0 on success), a number.

Errors Stops with an error if `tool` is not the name of a tool ("unknown tool 'TOOL'").

```
library(wfdbx)

status <- wfdbx_tool("rdsamp", c("-r", "100s", "-t", "s2"))     # the first 2 samples
# rdsamp prints two lines: the sample numbers 0 and 1, and the samples 995 1011
print(status)                           # 0
```

---

wfdbx_tools

List the command-line tools

```
wfdbx_tools()
```

Returns the names of the command-line tools. The function has no parameters.

Returns A character vector of the names of the tools, in alphabetical order (C collation).

Errors Nothing.

```
library(wfdbx)

print(length(wfdbx_tools()))            # 33
print(head(wfdbx_tools(), 4))           # "a2m" "ad2m" "ahaecg2mit" "ann2rr"
```

Environment variables. The tools, like the rest of the package, use the environment variables of Table 10.10. They are read when they are first needed in an R session.

Table 10.10 The environment variables used by the R package

Variable	Meaning
WFDBX_INPUT_DIR	the input directory (a directory, a ~ path or a URL), when wfdbx_set_input_dir does not set one
WFDBX_OUTPUT_DIR	the output directory, when wfdbx_set_output_dir does not set one
WFDBX_NETWORK	network access is on if it is a nonzero number, when wfdbx_set_network does not set it
WFDBANNSORT	0 to keep annotation files in the order written (no sorting when they are closed)
WFDBGVMODE	the default getvec mode (setgvmode with a negative mode)
WFDBCAL	the calibration file read by calopen(NULL) (default wfdbcal)
WFDB	the WFDB path, used in legacy mode only (outside legacy mode, it is ignored, with a message)
WFDB_FLAC_CHECK_MD5	if set, the MD5 signature of FLAC signal files is verified when they are read
HOME (USERPROFILE on Windows)	the home directory, for ~ paths
WFDBX_DEBUG	if set, the driver script prints the call stack of an R error in a tool
WFDBX_NO_CACHE	1 to turn the start-up cache of the driver script off
XDG_CACHE_HOME	the directory of the start-up cache (default ~/.cache)

Remote files are downloaded with the program curl if it is installed, and with R's url() otherwise; a file that has been read is kept for the rest of the R session, one that could not be read is requested again the next time it is opened, and a download gives up when it cannot connect, or receives nothing, for 30 seconds.

10.6 Testing

The unit tests of the R package are in r/tests: the runner run_tests.R and the files unit/test_*.R, written with base R only. From the root of the repository:

```
Rscript r/tests/run_tests.R              # all the unit tests
Rscript r/tests/run_tests.R metadata     # the files whose name matches "metadata"
```

The runner loads the sources of r/R (or the installed package, under R CMD check), runs each test file, and prints one line per file and one per failed check; it exits with status 1 if a check fails. The tests check the library functions with expected values of the C library, the high-level functions, the bulk paths against the sample-by-sample paths, printf and strtod against the GNU C library, the FLAC codec, check_record (against the reports of the corpus that python/tests/check_corpus.py writes, run with python3; skipped without it), and the header metadata functions against the cases shared by the seven languages (in tests/metadata: results, warnings and the headers, byte for byte). Network access is tested against a local HTTP server (127.0.0.1, run with python3; skipped without it). The tests work in temporary directories, with copies of the test records of tests/data; the tests that need tests/data are skipped when it is not available (in the installed package).

The package check builds the package and runs its examples and tests:

```
R CMD build r && R CMD check --no-manual wfdbx_*.tar.gz
```

The cross-language conformance suite compares the output of the tools with that of the original C programs, case by case: 1155 cases, and 886 in legacy mode, which runs each tool with `--legacy`. It runs the driver script `r/inst/bin/wfdbx.R`. From the root of the repository:

```
PYTHONPATH=$PWD/python/src python3 tests/conformance/harness.py check --impl r -j8
PYTHONPATH=$PWD/python/src python3 tests/conformance/harness.py check --impl r -j8 --legacy
```

CHAPTER 11

MATLAB and Octave

The MATLAB/Octave implementation of WFDB-x is the package `wfdbx`: the folder `matlab/+wfdbx` of the repository, with its internal functions in `+wfdbx/+internal` and the command-line tools in `+wfdbx/+apps`. It is plain MATLAB code, which runs unchanged in GNU Octave and in MATLAB, needs no toolbox, and never calls the C library. Every function is called with the name of the package, `wfdbx.NAME`.

It offers WFDB-x in three ways:

- **The high-level API** (Section 11.3): functions that read and write whole records, headers, annotation files and header metadata as matrices of doubles, structures and cell arrays, in one call, with name-value options, and that throw MATLAB errors (with an identifier, such as `wfdbx:read_record`) on failure. They read signal files in large blocks, and are the fastest way to read a record. They are an addition of WFDB-x: the C package has no equivalent. They are the recommended interface for new code and analysis.
- **The WFDB library functions** under their C names, called with the name of the package as a prefix: `wfdbx.isigopen`, `wfdbx.getvec`, `wfdbx.getann`, `wfdbx.putann`, and so on. They have the behavior of the C library (Section 11.4). They are documented once, for all the languages, in Chapter 13.
- **The command-line tools** (`rdsamp`, `rdann`, `wrsamp`, `xform` and 29 others), with the options, output and exit status of the C tools, run from the shell through an Octave script or a MATLAB shell script, or inside a session as functions (Section 11.5). They are documented in Chapter 14.

The high-level functions return exactly the samples and annotations that the library functions (and the C library) give.

11.1 Installation and requirements

11.1.1 Requirements

WFDB-x for MATLAB and Octave needs GNU Octave 6 or later, or MATLAB R2019b or later, and no toolbox or other package. It uses only the language features and functions that both products provide (no `string` objects, no `arguments` blocks, no Octave-only syntax), so the same files run in both. It has no compiled code, and needs no C compiler, no WFDB library and no libFLAC: the FLAC formats 508, 516 and 524 are read and written by WFDB-x's own codec.

The code is tested with GNU Octave 8.4 (Section 11.6). It has not yet been run in MATLAB: it is written to the rules that both products share, which are checked by reading the code, and its MATLAB command-line driver, `matlab/bin/wfdbx_matlab.sh`, has been checked in Octave only.

Remote files (Chapter 3) are downloaded with the program `curl` when it is installed, and otherwise with `webread` (Octave) or `websave` (MATLAB).

11.1.2 Installing

The package is used from a copy of the repository: its folder `matlab`, which holds `+wfdbx`, is added to the path:

```
addpath('/path/to/wfdb-x/matlab')
savepath      % optional: keep the folder on the path in later sessions
```

The folder `matlab` also holds copies of the license files of the repository: `LICENSE` (the GNU Lesser General Public License, version 2.1, of the library), `LICENSE-GPL` (the GNU General Public License, version 3, of the command-line tools: `+wfdbx/+apps`, the drivers of `bin` and the dispatcher `+wfdbx/+internal/cli_main.m` and `app_run.m`; the internal functions that the library also uses, such as `tool_command.m`, which `wfdbx.oannclose` calls to run `sortann` in legacy mode, stay under the LGPL) and `NOTICE`, which summarizes the licenses and gives the BSD-3-Clause notice of libFLAC, whose stream decoder the FLAC code of `+wfdbx/+internal` translates. The folder is the MATLAB and Octave distribution of WFDB-x: a copy of it that is passed on keeps these three files.

`help wfdbx.Contents` lists the functions, by group (in MATLAB, `help wfdbx` also does), and `help wfdbx.NAME` describes each one. `+wfdbx/Contents.m` gives the version of the package, `Version 1.2.3`, and the command-line driver prints it:

```
octave-cli --no-gui --norc -q matlab/bin/wfdbx_cli.m --version    # WFDB-x 1.2.3
```

11.1.3 The sample record

The repository includes the sample record `100s` in `tests/data`: the first minute of record 100 of the MIT-BIH Arrhythmia Database, two ECG signals (`MLII` and `V5`) sampled at 360 Hz, with the reference beat annotations (annotator `atr`), in the files `100s.hea`, `100s.dat` and `100s.atr`. The package has no function that returns this folder: a program names it, or sets it as the input directory. Work that uses the MIT-BIH Arrhythmia Database cites it as described on PhysioNet.

11.2 Quick start

This example reads the header, ten seconds of the signals and the annotations of the sample record `100s`, from the folder `tests/data` of a copy of the repository (`'input_dir'`), and computes the mean heart rate. It then writes the first ten seconds as a new record, `first10`, in the current folder, with two metadata entries in its header.

```
addpath('/path/to/wfdb-x/matlab');
data = '/path/to/wfdb-x/tests/data';     % the folder of the sample record 100s

% The header only: no samples are read.
hdr = wfdbx.read_header('100s', 'input_dir', data);
fprintf('%d %g %d\n', hdr.nsig, hdr.fs, hdr.nsamples);   % 2 360 21600
fprintf('%s\n', hdr.siginfo.desc);                       % MLII, then V5

% Seconds 10 to 20 of both signals, in physical units: one row per sample, one column
% per signal.
rec = wfdbx.read_record('100s', 'sampfrom', '0:10', 'sampto', '0:20', ...
    'physical', true, 'input_dir', data);
disp(size(rec.signals));                % 3600 2
fprintf('%g %g\n', rec.signals(1, :));  % -0.39 -0.275

% The beat annotations, and the mean heart rate.
ann = wfdbx.read_annotations('100s', 'atr', 'input_dir', data);
fprintf('%d %s %s %s\n', numel(ann.samples), ann.symbols{1:3});   % 75 + N N
```

```
beats = ann.samples(wfdbx.isqrs(ann.codes) == 1);
rr = diff(beats) / ann.fs;              % RR intervals, in seconds
fprintf('%d beats, %.1f beats per minute\n', numel(beats), 60 / mean(rr));
% 74 beats, 73.9 beats per minute

% Write the first 10 seconds, in ADC units, as the record "first10" (first10.hea and
% first10.dat, in the current folder), with the signal specifications of 100s.
x = wfdbx.read_record('100s', 'sampto', '0:10', 'input_dir', data);
wfdbx.write_record('first10', x.signals, x.fs, 'fmt', 16, 'gains', x.gains, ...
    'baselines', x.baselines, 'adczeros', [x.siginfo.adczero], ...
    'adcres', [x.siginfo.adcres], 'units', 'mV', 'names', x.names, ...
    'metadata', {'Source', 'MIT-BIH record 100'});

% Header metadata: "# Key: value" lines of the header.
wfdbx.set_metadata('first10', 'Age', '69');
m = wfdbx.read_metadata('first10')';    % (transposed, for fprintf)
fprintf('%s: %s\n', m{:});      % Source: MIT-BIH record 100, then Age: 69
wfdbx.check_record('first10');  % prints: no problems found
```

The header `100s.hea` gives no units: the example writes `'mV'` explicitly, since WFDB-x never assumes units, a gain, an ADC resolution or a sampling frequency that a record does not give (Chapter 4).

11.3 The high-level API

The high-level API is made of the functions of this section, in the package `wfdbx` (`wfdbx.read_record`, ...), and of the structures of their results (Section 11.3.6). Each is one file of `matlab/+wfdbx`; `help wfdbx.NAME` gives its documentation.

11.3.1 Conventions

Arguments and options. The first arguments of a function are positional (the record, the annotator, the samples, ...); the others are options, given as name-value pairs after them, in any order: `wfdbx.read_record('100s', 'sampto', 1000, 'physical', true)`. Option names are case-insensitive. An option that is not given keeps its default, as listed in the call syntax of each entry; `[]` means "not given" for the options whose default is `[]`. An unknown option, or a name without a value, is an error (Section 11.3.7). Text arguments are `char` row vectors or MATLAB strings (converted to `char`).

Record names and directories. A record is named as in the other parts of WFDB-x (Chapters 2 and 3), without the `.hea` extension of its header (the readers and the writers, `wfdbx.write_record` and `wfdbx.write_annotations`, also accept a trailing `.hea`, which is removed). A full path, absolute (`'/data/mitdb/100'`) or in the home directory (`'~/Documents/100s'`), is used as given; a relative path, such as `'100s'` or `'data/100s'`, is read from the input directory if one is set, otherwise from the current folder, and written to the output directory if one is set, otherwise to the current folder (missing folders are created); a URL (`'https://physionet.org/files/mitdb/1.0.0/100'`) is read only if network access is on, and cannot be written. A writer given a name with a path creates the files in that folder, and the header names the record and its signal file without the path (`'rec'`, `'rec.dat'`), so that the record opens under any path that leads to it; in legacy mode, as in the original package, the header gives the name with its path. The input and output directories are set with `wfdbx.wfdbx_set_input_dir` and `wfdbx.wfdbx_set_output_dir` (Section 11.4.3) or with the environment variables `WFDBX_INPUT_DIR` and `WFDBX_OUTPUT_DIR`. The functions that read take the options `'input_dir'` and `'network'`, and those that write take `'dir'`: they apply to that one call, and the previous settings are restored when it returns, also after an error. `'input_dir'` is

optional, and applies to relative names only, not to full paths or URLs; with `'dir'`, the record name must be a plain name (`'rec'`), without a folder. The signal files that a header names are read from the header's folder.

Times. The options `'sampfrom'` and `'sampto'` are sample numbers (whole numbers) or WFDB time strings, as `strtim` reads them: `'0:10'` (10 seconds), `'1:02:03.5'`, `'s3600'` (sample 3600), `'e'` (the end of the record), or a time of day in brackets, `'[14:00:00]'`. A time in seconds needs the sampling frequency; when the record does not give it, such a time is an error.

Numbers and arrays. Samples, times, counts and codes are doubles. Signal numbers start at 0, as in C. The signals of a record are a matrix with one row per sample and one column per signal; the annotations of a file are column vectors and cell columns, one row per annotation.

Per-signal and per-annotation options. The options that describe each signal (`'fmt'`, `'gains'`, `'units'`, `...`) or each annotation (`'subtypes'`, `'chans'`, `...`) take one value for all, or a vector (for numbers) or cell array (for text) with exactly one value per signal or per annotation.

Signal parameters. No sampling frequency, gain, ADC resolution or units is ever assumed. Where one is needed and the record does not give it, it must be given as an argument; otherwise the function throws an error. A record whose header gives no sampling frequency has `fs` 0.

Text. Text is held as bytes: each character of a `char` vector is one byte of a file (codes 0 to 255), so every byte sequence round-trips unchanged, and UTF-8 text is stored as its bytes. In Octave, a `char` array holds bytes already; in MATLAB, a character above 255 cannot be written to a header. The metadata functions also accept a `uint8` vector of bytes as a key or value.

Legacy mode. With `'legacy', true`, a function behaves as in the original WFDB package, with its assumed signal parameters and its search for files (Chapter 4); the library's legacy mode is on for the duration of the call only. With `false` (the default), the current setting of `wfdbx.wfdbx_set_legacy` applies, which is off unless it was turned on.

State. The functions that read and write records and annotation files call `wfdbx.wfdbquit()`, which closes anything opened with the WFDB library functions, and close their own files before they return, whatever the outcome. `wfdbx.read_record` sets the getvec mode for its own reading only. The functions `wfdbx.read_record`, `wfdbx.read_header`, `wfdbx.read_annotations`, `wfdbx.write_record`, `wfdbx.write_annotations` and `wfdbx.check_record` leave the getvec mode (`wfdbx.getgvmode`) and the time resolution of output annotators (`wfdbx.getafreq`) as they were when they return, also after an error: opening an EDF record sets both (it adds `WFDB_HIGHRES` to the getvec mode, and sets the time resolution to the record's sampling frequency when none is set), as in C, and `wfdbx.wfdbquit()` resets neither. The metadata functions open no signals, and leave the library's open files as they are.

The examples of this section read the test records of WFDB-x (among them `100s`) from the current folder, and write records of their own there: they run in a copy of the folder `tests/data` of the repository, made the current folder (with `matlab` on the path).

11.3.2 Reading records

`wfdbx.read_record` — Read the signals of a record

```
rec = wfdbx.read_record(record, 'channels', [], 'sampfrom', 0, 'sampto', [], ...
    'physical', false, 'highres', false, 'pad', false, 'input_dir', [], ...
```

```
    'network', [], 'legacy', false, 'checksum_warning', true)
```

Reads the signals of a record, or part of it, into a matrix of doubles with one row per sample and one column per signal, together with the description of the record. The samples are those that `getvec` returns (Chapter 13), read in large blocks. Invalid samples are `wfdbx.WFDB_INVALID_SAMPLE()` (-32768) in ADC units, and NaN in physical units. When the read reaches the end of the record, the samples are compared with the checksums of the header (Section 11.3.7).

`record`
: Record name (a `char` vector or a MATLAB string), without the `.hea` extension (a trailing `.hea` is also accepted): a full path, absolute (`'/data/mitdb/100'`) or in the home directory (`'~/Documents/100s'`), used as given; a relative path (`'100s'`, `'data/100s'`), read from the input directory if one is set, otherwise from the current folder; or a URL (`'https://physionet.org/files/mitdb/1`) read only if network access is on (Section 3.2).

`channels`
: The signals to read, in the order in which they are returned: a vector of signal numbers (from 0), or a cell array of signal numbers and signal descriptions (as in `rec.names`, such as `'MLII'`), which may be mixed. Default `[]`: all the signals, in the order of the header.

`sampfrom`
: The first sample to read: a sample number (a whole number ≥ 0) or a time string. Default `0`: the beginning of the record.

`sampto`
: The sample after the last one to read: a sample number (a whole number) or a time string. A value at or before `sampfrom` reads no samples. Default `[]`: the end of the record.

`physical`
: `false` (the default) to return ADC units; `true` to return physical units, computed as $(adc - baseline)/gain$. Every signal read must then have a gain.

`highres`
: `false` (the default) to read a multi-frequency record at its frame frequency, with the samples of the faster signals averaged to one per frame; `true` to read every sample, at the highest sampling frequency of the record (the samples of the slower signals are repeated). `rec.fs` is then that frequency.

`pad` `true` to replace each invalid sample by the previous valid sample of the same signal. Default `false`.

`input_dir`
: The input directory for this call only (optional): a local folder, a ~ path or the URL of a directory, under which relative record names are read (": the default). It does not apply to full paths or URLs. Default `[]`: the current setting (`wfdbx.wfdbx_set_input_dir`).

`network`
: `true` to allow reading URLs for this call only, `false` to forbid it. Default `[]`: the current setting (`wfdbx.wfdbx_set_network`).

`legacy`
: `true` to behave as the original WFDB package (legacy mode, for this call only). Default `false`.

`checksum_warning`
: `true` (the default) to issue the warning `wfdbx:checksum` when the samples do not match the checksums of the header; `false` for no warning. The samples are returned either way, and `rec.checksum_ok` is `false`.

The memory allocated does not depend on the length that the header gives, which may be far more than the signal files hold: the samples are read in chunks of about 2^{22}, joined at the end.

Returns A structure (Table 11.1).

Errors Identifier `wfdbx:read_record`: the record cannot be read ("can't read record REC"), or its signals cannot be opened; a signal file is shorter than its header says, or cannot be read (a decoder or I/O error): "can't read signal data of record REC (getvec status -3)", even if the checksums do not match either (the checksum error is found in the frame in which the file ended), so that no incomplete frame is ever returned; a requested signal is not in the record; `sampfrom` is negative, or `sampfrom` or `sampto` is not a whole number (a fraction, NaN or an infinite value: "read_record: sampfrom must be a whole number"), or is neither a number nor a time string ("... must be a sample number or a time string"); a time string needs the sampling frequency and the header gives none; `'physical'` is `true` and a signal read is uncalibrated; or an option is not valid. Warning `wfdbx:checksum` on a checksum error (see `checksum_warning`).

```
% Seconds 10 to 20 of lead MLII, in physical units.
rec = wfdbx.read_record('100s', 'channels', {'MLII'}, 'sampfrom', '0:10', ...
    'sampto', '0:20', 'physical', true);
disp(size(rec.signals));                          % 3600 1
fprintf('%s %g %d\n', rec.names{1}, rec.fs, rec.sampfrom);   % MLII 360 3600

% The first 3 samples of both signals, in ADC units, V5 first.
rec = wfdbx.read_record('100s', 'channels', [1 0], 'sampto', 3);
disp(rec.signals);          % three rows of 1011 995
disp(rec.names);            % {'V5', 'MLII'}
disp([rec.gains; rec.baselines]);      % 200 200; 1024 1024

% The whole record: the checksums are verified.
rec = wfdbx.read_record('100s');
fprintf('%d %d %d\n', size(rec.signals), rec.checksum_ok);    % 21600 2 1
```

A multi-frequency record (here the test record `mfreq`, whose signals have 1, 2 and 4 samples per frame at a frame frequency of 100 Hz) is read at its frame frequency, or at its highest sampling frequency:

```
rec = wfdbx.read_record('mfreq');
fprintf('%g %d %d\n', rec.fs, size(rec.signals));     % 100 600 3
rec = wfdbx.read_record('mfreq', 'highres', true);
fprintf('%g %d %d\n', rec.fs, size(rec.signals));     % 400 2400 3
disp([rec.siginfo.spf]);                              % 1 2 4
```

`wfdbx.read_header` — Read the header of a record

```
hdr = wfdbx.read_header(record, 'input_dir', [], 'network', [], 'legacy', false)
```

Reads the header of a record, without opening its signal files. The result describes the record and each of its signals, and holds the info strings and the metadata entries of the header. For a multi-segment record, the master header is read, and `nsamples` is the length of the whole record. The metadata are read from the `.hea` file only; an EDF record (which has no `.hea` file) and a header read from the standard input (`'-'`) have none.

`record`

Record name, without the `.hea` extension (a trailing `.hea` is also accepted): a full path,

absolute (`'/data/mitdb/100'`) or in the home directory (`'~/Documents/100s'`), used as given; a relative path (`'100s'`, `'data/100s'`), read from the input directory if one is set, otherwise from the current folder; or a URL (`'https://physionet.org/files/mitdb/1.0.0/100'`), read only if network access is on (Section 3.2).

input_dir
The input directory for this call only (optional), under which relative record names are read; it does not apply to full paths or URLs (as for `wfdbx.read_record`). Default `[]`: the current setting.

network
`true` to allow reading URLs for this call only, `false` to forbid it. Default `[]`: the current setting.

legacy
`true` to read the header as the original WFDB package does. Default `false`.

Returns A structure (Table 11.2).

Errors Identifier `wfdbx:read_header`: the header cannot be found or read ("can't read header for record REC"), or an option is not valid. Warning `wfdbx:duplicate_key` for a metadata key that the header has more than once.

```
hdr = wfdbx.read_header('100s');
fprintf('%s %g %d %d\n', hdr.name, hdr.fs, hdr.nsig, hdr.nsamples);  % 100s 360 2 21600
for k = 1:hdr.nsig
    s = hdr.siginfo(k);
    fprintf('%s %s %d %g %d\n', s.desc, s.fname, s.fmt, s.gain, s.adcres);
end
% MLII 100s.dat 212 200 11
% V5 100s.dat 212 200 11
disp(hdr.comments{1});          % " 69 M 1085 1629 x1": the first info string
disp(isempty(hdr.base_time));   % 1: the header gives no start time
```

---

wfdbx.read_annotations

Read an annotation file

```
ann = wfdbx.read_annotations(record, annotator, 'sampfrom', 0, 'sampto', [], ...
    'fs', [], 'input_dir', [], 'network', [], 'legacy', false)
ann = wfdbx.read_annotations(record, annotator, sampfrom, sampto, ...)
```

Reads the annotations of an annotation file, in file order, into column vectors and cell columns. The sampling frequency, needed to convert times in seconds, is that of the record's header, or, for a record without a header or whose header gives none, the option `'fs'`. `'sampfrom'` and `'sampto'` may also be given as the third and fourth arguments, without their names.

record
Record name, without the `.hea` extension (a trailing `.hea` is also accepted): a full path, absolute (`'/data/mitdb/100'`) or in the home directory (`'~/Documents/100s'`), used as given; a relative path (`'100s'`, `'data/100s'`), read from the input directory if one is set, otherwise from the current folder; or a URL (`'https://physionet.org/files/mitdb/1.0.0/100'`), read only if network access is on (Section 3.2). Its header, if there is one, gives the sampling frequency.

annotator
Annotator name: the suffix of the annotation file. `'atr'` reads the file `RECORD.atr`.

sampfrom
: The first time to read: a sample number (a whole number ≥ 0) or a time string. Default `0`: the beginning of the file.

sampto
: The time at which to stop: annotations at or after it are not read. A sample number (a whole number) or a time string. Default `[]`: the end of the file.

fs
: The sampling frequency, in Hz (a number > 0), to use when the record has no header or its header gives none; it is ignored otherwise. Default `[]`: none.

input_dir
: The input directory for this call only (optional), under which relative record names are read; it does not apply to full paths or URLs (as for `wfdbx.read_record`). Default `[]`: the current setting.

network
: `true` to allow reading URLs for this call only, `false` to forbid it. Default `[]`: the current setting.

legacy
: `true` to behave as the original WFDB package (which assumes 250 Hz for a record without a frequency). Default `false`.

Returns A structure (Table 11.3). Its `fs` is the frequency in which the times are counted: the one written in the annotation file, if there is one, or else the sampling frequency; it is 0 if none of them is known.

Errors Identifier `wfdbx:read_annotations`: the annotation file cannot be opened ("can't read annotator ANN for record REC"), or it ends unexpectedly or cannot be read before `sampto` ("can't read annotation file ANN of record REC (getann status S)"); `'fs'` is given and is not > 0; `sampfrom` or `sampto` is not valid, as for `wfdbx.read_record` ("read_annotations: sampfrom must be a whole number", "... must be a sample number or a time string", "... is beyond the range of a 64-bit integer", "read_annotations: sampfrom must not be negative"), which is checked before any file is opened; a time string needs the sampling frequency and none is known; or an option is not valid. Whatever happens, the annotation file is closed (`wfdbx.wfdbquit`) before the function returns.

```
ann = wfdbx.read_annotations('100s', 'atr', 'sampfrom', '0:10', 'sampto', '0:30');
fprintf('%d %g\n', numel(ann.samples), ann.fs);       % 24 360
fprintf('%d %s\n', ann.samples(1), ann.symbols{1});   % 3862 N

% The whole file: the first annotation is a rhythm change, with an aux string.
ann = wfdbx.read_annotations('100s', 'atr');
fprintf('%s %d %s\n', ann.symbols{1}, ann.codes(1) == wfdbx.RHYTHM(), ann.aux{1});
% + 1 (N
disp(unique(ann.symbols)');     % {'+', 'A', 'N'}
```

For a record without a header, `'fs'` gives the frequency in which times in seconds are converted (the library reports on the standard error that it cannot open the header):

```
wfdbx.write_annotations('beats', 'qrs', [100 460 820], 'symbols', 'N');   % no header
ann = wfdbx.read_annotations('beats', 'qrs', 'sampto', '0:02', 'fs', 360);
fprintf('%d %d %g\n', ann.samples, ann.fs);           % 100 460 360
```

11.3.3 Writing records

wfdbx.write_record Write a header and a signal file

```
wfdbx.write_record(record, signals, fs, 'fmt', 16, 'gains', [], 'baselines', [], ...
    'adczeros', 0, 'adcres', [], 'units', [], 'names', [], 'base_time', [], ...
    'comments', [], 'physical', false, 'samps_per_frame', [], 'metadata', [], ...
    'dir', [], 'legacy', false, 'invalid_warning', true)
```

Writes signals as a new record: the header `RECORD.hea` and one signal file, `RECORD.dat`, which holds all the signals (so they share one format). Existing files of these names are replaced: both files are written to temporary files in the destination folder (`.NAME.XXXXXXXX.tmp`, created with the permissions of the files they replace), which replace the existing files only when both are complete, so that an error leaves the existing record as it was, and no temporary file is left. The sampling frequency, the gain and ADC resolution of each signal, and the units of each calibrated signal must be given: there are no defaults. The per-signal options (`'fmt'`, `'gains'`, `'baselines'`, `'adczeros'`, `'adcres'`, `'units'`, `'names'`) take one value for all the signals, or a vector or cell array with one value per signal. All the arguments are checked before any file is created or changed.

record
: Record name (not empty), without the `.hea` extension (a trailing `.hea` is removed, as by the readers): a plain name (`'rec'`) or a relative path (`'data/rec'`), created in the output directory if one is set, otherwise in the current folder, missing folders included; or a full path, absolute or in the home directory (`'~/Documents/rec'`), used as given. With a path, the files are created in that folder, and the header names the record and its signal file without the path (`'rec'`, `'rec.dat'`), so that the record opens under any path that leads to it (in legacy mode, as in the original, the header gives the name with its path). A URL cannot be written.

signals
: The samples: a matrix (a 1-D or 2-D array) with one row per sample and one column per signal. In ADC units (the default), the values are integers within the range of a 32-bit integer; with `'physical', true`, they are in physical units.

fs
: The sampling frequency, in Hz: a finite number > 0.

fmt
: The storage format: 8, 16, 61, 80, 160, 212, 310, 311, 24, 32, or 508, 516 or 524 (FLAC); one value, or one per signal, all equal (Chapter 15). Every sample must fit the format: within its range (Table 15.1; -2048 to 2047 for format 212, for example), and, in format 8, within -128 to 127 of the sample before it. The smallest value of a format is the value of an invalid (missing) sample: a sample equal to it, or to `WFDB_INVALID_SAMPLE`, is read back as invalid (see `invalid_warning`). A FLAC file holds at most 8 signals. Default `16`.

gains
: The gain of each signal, in ADC units per physical unit: a finite number; `0` marks an uncalibrated signal, which is allowed only for samples given in ADC units. Negative gains are allowed. Required (default `[]`: not given).

baselines
: The ADC value that corresponds to 0 physical units (a whole number within the range of a 32-bit integer; in a cell array, `[]` is the ADC zero). Default `[]`: the ADC zero of the signal.

adczeros
: The ADC value at the middle of the ADC range (a whole number within the range of a 32-bit integer; in a cell array, `[]` is 0). Default `0`.

adcres
The ADC resolution, in bits (a whole number ≥ 1, within the range of a 32-bit integer). Required (default `[]`: not given).

units
The physical units, such as `'mV'`: one string for all the signals, or a cell array with one per signal. Required for each signal whose gain is not 0. Default `[]`: not given.

names
The signal descriptions, such as `'MLII'`: one string, or a cell array with one per signal. Default `[]`: no descriptions.

base_time
The start time, with an optional date: `'hh:mm:ss'` or `'hh:mm:ss dd/mm/yyyy'`, with optional decimals of the seconds, as `wfdbx.setbasetime` reads it. Default `[]`: no start time.

comments
A cell array of info strings (or one string), written to the header as comment lines (`#` followed by the string). Default `[]`: none.

physical
`true` if `signals` are in physical units. They are converted to ADC units as $\mathrm{round}(x \cdot gain) + baseline$ (halves rounded away from zero); NaN becomes an invalid sample, and infinite values and values beyond the range of 32-bit ADC units are errors. Default `false`: ADC units.

samps_per_frame
For a multi-frequency record, the number of samples per frame of each signal (integers ≥ 1). `signals` then has one row per frame and `sum(samps_per_frame)` columns: the samples of signal 0, then those of signal 1, and so on. `fs` is then the frame frequency. Default `[]`: one sample per frame for every signal.

metadata
Metadata entries (Chapter 5) to write to the header after the comments, as `# Key: value` (`# Key:` for an empty value): an N-by-2 cell array with one entry per row, `{KEY, VALUE; ...}`. Keys and values follow the rules of `wfdbx.set_metadata`; no two keys may be equal without regard to letter case, nor be the key of an entry in `'comments'`, and a value must not be `[]`. Default `[]`: none.

dir The folder (which must exist) in which to create the files, for this call only; `record` must then be a plain name, without a folder, and the record opens from that folder under its plain name. Default `[]`: the output directory.

legacy
`true` to behave as the original WFDB package: the gain defaults to 200, and the ADC resolution to one based on the format. Default `false`.

invalid_warning
`true` to issue the warning `wfdbx:invalid_samples` when samples will be read back as invalid (missing) samples ("N sample(s) of signal K will be read back as invalid (missing) samples (format F stores an invalid sample as V)"): those equal to the invalid value of the format or to `WFDB_INVALID_SAMPLE`, other than the NaN samples of physical units, which mark missing samples deliberately. `false` for no warning, when such samples are deliberate. Default `true`.

Returns Nothing.

Errors Identifier `wfdbx:write_record`: `record` is empty; `record` has a character beyond 255 ("record name 'X' has a character that is not Latin-1 (one byte per character)"; the same for a signal name, the units and a comment: "signal name 'X' ...", "units 'X' ...", "comment 'X' ..."; text is one byte per character, so this can happen only in MATLAB, whose characters are 16-bit);

`signals` has more than two dimensions ("signals must be a 1-D or 2-D array"); a `'samps_per_frame'` value is not an integer ≥ 1 ("samps_per_frame must hold integers >= 1"), or the number of columns of `signals` is not their sum ("signals has N columns, but samps_per_frame adds up to M"); `signals` has no signals ("at least one signal is required"); `fs` is not finite and > 0; a gain is not finite; the gains, the ADC resolutions or the required units are not given; an ADC resolution, ADC zero or baseline is not a whole number within the range of a 32-bit integer (text, NaN, 12.5: "adcres must be whole numbers within the range of a 32-bit integer", and the same for `adczeros` and `baselines`); a per-signal option does not have one value or one value per signal; a sample in ADC units is not an integer or is outside the range of a 32-bit integer, or a sample in physical units is infinite or its ADC value is outside that range; `'physical'` is `true` and a gain is 0; `'base_time'` cannot be parsed ("invalid base_time 'T'"); the signals do not all have the same format; a sample does not fit the format ("signal K has a sample (V) outside the range of format F (LO to HI)"; in format 8, "signal K changes by D between two samples, more than format 8 can store (-128 to 127)" or "signal K has an invalid sample, which format 8 cannot store"); a FLAC format is given for more than 8 signals ("format 516 (FLAC) stores at most 8 signals in a file; the record has 12"); an entry of `'metadata'` is not valid; `record` contains a folder and `'dir'` is given, or `'dir'` does not exist; an option is not valid; or the files cannot be created, or cannot be written completely, for example on a full disk ("write error on record REC", after the library's message; the existing files are then left as they were).

```
% Two signals of random numbers, in millivolts, at 360 Hz: the record "rand"
% (rand.hea and rand.dat).
x = randn(3600, 2);
wfdbx.write_record('rand', x, 360, 'fmt', 16, 'physical', true, 'gains', 1000, ...
    'adcres', 16, 'units', 'mV', 'names', {'A', 'B'}, ...
    'base_time', '14:00:03 21/06/2024', 'comments', {'random numbers'}, ...
    'metadata', {'Generator', 'randn'});
rec = wfdbx.read_record('rand', 'physical', true);
fprintf('%d %d %s\n', size(rec.signals), rec.base_time);   % 3600 2 [14:00:03 21/06/2024]
disp(max(abs(rec.signals(:) - x(:))) <= 0.0005);           % 1: rounded to 1/1000 mV
disp(rec.comments');            % {'random numbers', ' Generator: randn'}

% An uncalibrated signal (gain 0, no units), in ADC units, in the folder "out".
mkdir('out');
wfdbx.write_record('counts', (0:999)', 100, 'gains', 0, 'adcres', 12, 'dir', 'out');
rec = wfdbx.read_record('out/counts');
disp(rec.signals(1:3)');        % 0 1 2
```

A multi-frequency record: signal `fast` has 2 samples per frame, signal `slow` one; each row of `signals` is one frame, with the two samples of `fast` and then the sample of `slow`:

```
frames = 100;                                % 1 second at 100 frames per second
fast = reshape(0:2 * frames - 1, 2, frames)';   % 200 samples per second
slow = repmat(500, frames, 1);               % 100 samples per second
wfdbx.write_record('mf', [fast, slow], 100, 'gains', 200, 'adcres', 12, ...
    'units', 'mV', 'names', {'fast', 'slow'}, 'samps_per_frame', [2 1]);
rec = wfdbx.read_record('mf', 'highres', true);
fprintf('%g\n', rec.fs);        % 200
disp(rec.signals(1:3, :));      % 0 500; 1 500; 2 500
```

`wfdbx.write_annotations` — Write an annotation file

```
wfdbx.write_annotations(record, annotator, samples, 'symbols', [], 'codes', [], ...
```

```
    'subtypes', 0, 'chans', 0, 'nums', 0, 'aux', [], 'fs', [], 'dir', [], ...
    'sort', [], 'legacy', false)
```

Writes an annotation file, `RECORD.ANNOTATOR`, in the MIT format. An existing file of that name is replaced: the new file is written to a temporary file in the same folder (created with the permissions of the file it replaces), which replaces it only when it is complete (and sorted), so that an error leaves it as it was. The record need not have a header. The annotation types are given as `'symbols'` or as `'codes'` (Chapter 15). Each per-annotation option is one value for all the annotations, or a list with exactly one value per annotation. If the annotations are not in time order, the file is sorted when it is written, unless `'sort'` is `false`. All the arguments are checked before the file is created or changed.

record
: Record name, without the `.hea` extension (a trailing `.hea` is also accepted): a plain name or a relative path (`'data/100s'`) is created in the output directory if one is set, otherwise in the current folder; a full path, absolute or in the home directory (`'~/Documents/100s'`), is used as given. The file is `RECORD.ANNOTATOR` at that place (`data/100s.qrs`). A URL cannot be written.

annotator
: Annotator name: the suffix of the file, made of letters, digits, _, - and ~.

samples
: The annotation times, in samples: a vector (a 1-D array) of integers from $-2^{63}+1$ to $2^{63}-2$, one per annotation (the smallest and the largest 64-bit integers are reserved by the file format; doubles are exact up to 2^{53}, and `int64` values are accepted).

symbols
: The mnemonics of the annotation types, such as `'N'` or `'V'`: a cell array with one per annotation, a `char` vector of one-character mnemonics (`'NNV'`), or one mnemonic for all. An unknown mnemonic becomes code 0 (`NOTQRS`). Default `[]`: the types are given by `'codes'`.

codes
: The annotation codes (`wfdbx.NORMAL()`, `wfdbx.PVC()`, ...; 0 to 58, as 59 to 63 are the control codes of the file format), instead of `'symbols'`: one per annotation, or one value for all. If both are given (as `wfdbx.read_annotations` returns them), each symbol must be that of its code (an empty symbol is not checked), and the codes are written. Default `[]`.

subtypes
: The subtype of each annotation, stored as a signed byte (other values wrap around to $-128 \ldots 127$, as in C). Default `0`.

chans
: The channel number of each annotation, stored as an unsigned byte (other values wrap around to $0 \ldots 255$). Default `0`.

nums
: The annotator number of each annotation, stored as a signed byte. Default `0`.

aux
: The auxiliary text of each annotation: a cell array with one per annotation, with `[]` where there is none, or one `char` vector for all. Default `[]`: none.

fs
: If given (a finite number > 0), the frequency in which the times are counted, written to the file. Default `[]`: none is written.

dir
: The folder (which must exist) in which to create the file, for this call only; `record` must then be a plain name, without a folder: the file is `DIR/RECORD.ANNOTATOR`. Default `[]`: the output directory.

sort
: `true` to always sort the file, `false` to keep the given order (without the library's advice to run `sortann`), for this call only. Default `[]`: the environment variable `WFDBANNSORT` decides (sorting is on unless it is `0`).

legacy
: `true` to behave as the original WFDB package. Default `false`.

Returns Nothing.

Errors Identifier `wfdbx:write_annotations`: `samples` is a matrix or an array of more dimensions ("samples must be a 1-D array"; the same for `'codes'`, `'subtypes'`, `'chans'`, `'nums'`, and for `'symbols'` given with `'codes'`); neither `'symbols'` nor `'codes'` is given for one annotation or more ("symbols or codes must be given"; with no annotations, an empty file is written); a per-annotation option given as a list does not have one value per annotation ("... must have one value per annotation"); a time is not an integer within the range of a 64-bit integer (NaN, an infinite value or a fraction); a code, subtype, channel or annotator number is NaN or infinite; `'fs'` is not finite and > 0; a time is -2^{63} or, as an `int64`, $2^{63} - 1$ ("annotation time T cannot be written (annotation times must be integers from -9223372036854775807 to 9223372036854775806)"); a code is not from 0 to 58 ("annotation code C is reserved (59 to 63 are the control codes of the annotation file format)", "annotation code C is out of range (0 to 58)"); both `'symbols'` and `'codes'` are given and disagree ("symbols and codes disagree at annotation K", K counting from 0); an aux text has a character beyond 255 ("aux text 'X' has a character that is not Latin-1 (one byte per character)", in MATLAB only); `record` contains a folder and `'dir'` is given, or `'dir'` does not exist; an option is not valid; or the file cannot be created, or cannot be written completely, or the library reports an error while writing it ("write error on annotator ANN for record REC"; the existing file is then left as it was).

```
% Three beats and a rhythm change, by symbol, with an aux string for the rhythm.
wfdbx.write_annotations('100s', 'test', [18 77 370 662], ...
    'symbols', {'+', 'N', 'V', 'N'}, 'aux', {'(N', [], [], []});
ann = wfdbx.read_annotations('100s', 'test');
fprintf('%s', ann.symbols{:}); fprintf(' %s\n', ann.aux{1});    % +NVN (N

% A copy of the reference annotations, as read_annotations returns them.
a = wfdbx.read_annotations('100s', 'atr');
wfdbx.write_annotations('100s', 'copy', a.samples, 'symbols', a.symbols, ...
    'codes', a.codes, 'subtypes', a.subtypes, 'chans', a.chans, 'nums', a.nums, ...
    'aux', a.aux);
b = wfdbx.read_annotations('100s', 'copy');
disp([isequal(b.samples, a.samples), isequal(b.symbols, a.symbols), ...
    isequal(b.aux, a.aux)]);    % 1 1 1

% Annotations out of time order are sorted when the file is written (the library
% reports it on the standard error).
wfdbx.write_annotations('100s', 'unsorted', [500 100], 'codes', wfdbx.NORMAL());
ann = wfdbx.read_annotations('100s', 'unsorted');
disp(ann.samples');             % 100 500
```

11.3.4 Checking records

wfdbx.check_record Check a record and its annotation files

```
report = wfdbx.check_record(record, 'annotators', {}, 'input_dir', [], 'network', [])
```

Checks the header of a record, its signal specifications, every sample of its signal files, each segment of a multi-segment record, and the annotation files named, and reports each problem found. A problem in the record never throws an error: it is reported, with a severity (`'error'`: the record is damaged or does not follow the WFDB format, and programs will read wrong data or fail; `'warning'`: the record can be read, but something is missing or unusual, and may give wrong results; `'info'`: a fact worth knowing, such as invalid samples). The problems, their codes and their messages are the same in the seven languages (Table 6.7). The checks run in normal mode, even if legacy mode is on, and the library prints no messages during the call. They are similar to those of Benjamin Moody's `wfdbcheck`. Called without an output argument, the function displays the lines of the report.

`record`
: Record name, without the `.hea` extension (a trailing `.hea` is also accepted): a full path, absolute (`'/data/mitdb/100'`) or in the home directory (`'~/Documents/100s'`), used as given; a relative path (`'100s'`, `'data/100s'`), read from the input directory if one is set, otherwise from the current folder; or a URL (`'https://physionet.org/files/mitdb/1.0.0/100'`), read only if network access is on (Section 3.2).

`annotators`
: The annotator names whose files are checked: a cell array, such as `{'atr'}`, or one name as a `char` vector. Default `{}`: none.

`input_dir`
: The input directory for this call only (optional), under which relative record names are read; it does not apply to full paths or URLs (as for `wfdbx.read_record`). Default `[]`: the current setting.

`network`
: `true` to allow reading URLs for this call only, `false` to forbid it. Default `[]`: the current setting.

Returns A structure (Table 11.4): `report.ok` is `true` if no problem is an error, `report.issues` is a structure array of the problems (Table 11.5), and `report.lines` is a cell column with one line per problem, or the one line `no problems found`.

Errors Identifier `wfdbx:check_record`, only for invalid arguments: `record` is not a string or is empty, an annotator name is not a string, or an option is not valid. A record that cannot be read is reported (code `header_unreadable`), not thrown.

```
report = wfdbx.check_record('100s', 'annotators', {'atr'});
disp(report.ok);                % 1: no errors
fprintf('%s\n', report.lines{:});
% warning: signal 0: the units are not specified
% warning: signal 1: the units are not specified

% A damaged record: one sample of a 12-bit signal is overwritten with 2048.
wfdbx.write_record('damaged', (0:99)', 100, 'gains', 100, 'adcres', 12, ...
    'units', 'mV', 'names', 'ramp');
fid = fopen('damaged.dat', 'r+b');
fseek(fid, 20, 'bof');          % sample 10 (2 bytes per sample in format 16)
fwrite(fid, [0 8], 'uint8');    % 0x0800 = 2048, little-endian
fclose(fid);
report = wfdbx.check_record('damaged');
disp(report.ok);                % 0
for k = 1:numel(report.issues)
```

```
    i = report.issues(k);
    fprintf('%s %s %s\n', i.severity, i.code, i.where);
end
% error sample_range signal 0
% error checksum signal 0
disp(report.lines{1});
% error: signal 0: 1 sample is outside the ADC range (-2048 to 2047): sample 10 (value 2048)
```

11.3.5 Header metadata

These functions read and change the metadata of a header: its info strings of the form `# Key: value`, such as `# Age: 63` (Chapter 5). In short: keys are compared without regard to letter case (A–Z and a–z only); a key must not be empty and must not contain `:`, `#`, a line break or a NUL character, and a value must not contain a line break or a NUL character; spaces, tabs, VT, FF and CR at either end of keys and values are removed; text is one byte per character, and keys and values may be `char` vectors, MATLAB strings or `uint8` vectors of bytes. The entries are an N-by-2 cell array of `char` row vectors, one entry per row, `{KEY, VALUE; ...}`, in file order (`cell(0, 2)` if there is none); a structure cannot hold keys with spaces, such as `Date of birth`, so none is used. The info strings are the lines that begin with `#` after the record line, which is found as the header parser of the library finds it, in every mode: the first line that holds something other than white space (spaces, tabs, CRs and LFs, up to a NUL character) and does not begin, after white space, with `#`. Comment lines, and empty or blank lines, before the record line are never read or changed. A change rewrites the header in one step, through a temporary file, keeping every other byte of it; all the arguments are checked first, so an error leaves the header unchanged. Only the `.hea` file is read (not a `.info` file); for a multi-segment record, it is the master header, unless a segment is named. Metadata often holds information about patients: WFDB-x stores it as plain text, exactly as given, and does not de-identify it.

Each function has two error identifiers (Section 11.3.7): `wfdbx:NAME` for an invalid argument, and `wfdbx:NAME:file` for a header that cannot be read or written, or a `'dir'` that does not exist (NAME is the name of the function, such as `set_metadata`).

Patient data A header can hold identifying data about a patient (a name, a date of birth, a medical record number). The metadata functions read and write such data as given. Records that are shared must be de-identified first.

Permissions of a changed header The temporary file is renamed over the header (`rename` in Octave, `movefile` in MATLAB), so the header keeps its permissions only if the temporary file has them. The temporary file is created with the permissions of the header, whatever the name of the header (parentheses, spaces and any other characters included), and no shell command is run on the name: Octave, which has no `chmod` function, creates it under a `umask` that leaves exactly the read and write permissions of the header, so that a private (0600) header stays private, and is never readable by others while it is rewritten; MATLAB gives it the read, write and execute permissions of the header with `fileattrib`. In Octave, execute permissions (which a header does not normally have) are not kept. The files that `wfdbx.write_record` and `wfdbx.write_annotations` replace keep their permissions in the same way.

`wfdbx.read_metadata` Read the metadata of a record

```
M = wfdbx.read_metadata(record, 'input_dir', [], 'network', [])
```

Reads the metadata entries of the header of a record, in file order. An info string is an entry if it contains a colon: the key is the text before the first colon, the value the rest of the line, both without spaces at either end. Info strings without a colon, and those whose key is empty or contains `#`, are free comments, which are not returned.

`record`
: Record name, without the `.hea` extension (a trailing `.hea` is also accepted): a full path, absolute (`'/data/mitdb/100'`) or in the home directory (`'~/Documents/100s'`), used as given; a relative path (`'100s'`, `'data/100s'`), read from the input directory if one is set, otherwise from the current folder; or a URL (`'https://physionet.org/files/mitdb/1.0.0/100'`), read only if network access is on (Section 3.2).

`input_dir`
: The input directory for this call only (optional), under which relative record names are read; it does not apply to full paths or URLs (as for `wfdbx.read_record`). Default `[]`: the current setting.

`network`
: `true` to allow reading URLs for this call only, `false` to forbid it. Default `[]`: the current setting.

Returns An N-by-2 cell array of `char` row vectors, `{KEY, VALUE; ...}`, in file order. A key that appears more than once (ignoring letter case) is returned once, with its first value, and the warning `wfdbx:duplicate_key` is issued (Section 11.3.7).

Errors Identifier `wfdbx:read_metadata`: `record` is empty or `'-'` (the standard input), or an option is not valid. Identifier `wfdbx:read_metadata:file`: the header cannot be found or read ("can't read header for record REC").

```
wfdbx.write_record('pat', zeros(360, 1), 360, 'gains', 200, 'adcres', 12, ...
    'units', 'mV', 'names', 'ECG', 'comments', {'recorded at rest'}, ...
    'metadata', {'Age', '63'; 'Sex', 'F'});
M = wfdbx.read_metadata('pat');
disp(size(M));                  % 2 2
M = M';
fprintf('%s: %s\n', M{:});      % Age: 63, then Sex: F
hdr = wfdbx.read_header('pat');
disp(hdr.comments');
% {'recorded at rest', ' Age: 63', ' Sex: F'}: the free comment is not an entry
```

`wfdbx.get_metadata` — Return the value of a metadata entry

```
value = wfdbx.get_metadata(record, key, 'input_dir', [], 'network', [])
```

Returns the value of one metadata entry of the header of a record.

`record`
: Record name (as for `wfdbx.read_metadata`).

`key` The key, such as `'Age'`. Letter case is ignored, and spaces at either end are removed.

`input_dir`
: The input directory for this call only (optional), under which a relative record name is read (as for `wfdbx.read_metadata`). Default `[]`: the current setting.

network
: `true` to allow reading URLs for this call only, `false` to forbid it. Default `[]`: the current setting.

Returns The value, a `char` row vector (empty for an entry with an empty value), or `[]` if the header has no entry for `key` (`ischar` tells them apart). If it has more than one, the first is returned, and the warning `wfdbx:duplicate_key` is issued.

Errors Identifier `wfdbx:get_metadata`: `record` is empty or `'-'`, `key` is missing, not a string (or `uint8` vector) or not a valid key, or an option is not valid. Identifier `wfdbx:get_metadata:file`: the header cannot be found or read.

```
wfdbx.write_record('pat', zeros(360, 1), 360, 'gains', 200, 'adcres', 12, ...
    'units', 'mV', 'metadata', {'Age', '63'; 'Sex', 'F'});
disp(wfdbx.get_metadata('pat', 'age'));         % 63
v = wfdbx.get_metadata('pat', 'Height');
disp([isempty(v), ischar(v)]);                  % 1 0: there is no such entry
```

---

wfdbx.set_metadata

Set the value of a metadata entry

```
wfdbx.set_metadata(record, key, value, 'input_dir', [], 'network', [], 'dir', [])
```

Sets the value of a metadata entry in the header of a record. If the header has an entry for `key` (ignoring letter case), its line is replaced by `# Key: value`, at the same position and with the same line ending, with the spelling of `key` given here (even if the value is the same); otherwise the entry is added after the last info string of the header (or, if it has none, after its last line that is not blank), with the line ending of the record line. The entry is written as `# Key: value`, or `# Key:` if the value is empty. Every other line is kept, byte for byte, including its line ending. The header is changed where it was read (not in the output directory), unless `'dir'` is given.

record
: Record name (as for `wfdbx.read_metadata`). The header is changed where it was read, unless `'dir'` is given; a header read from a URL can only be written elsewhere, with `'dir'`.

key The key, such as `'Age'` or `'Date of birth'`. Spaces at either end are removed. It must not be empty, and must not contain `:`, `#`, a line break or a NUL character.

value
: The value: text, which may be a sentence. Spaces at either end are removed. It must not contain a line break or a NUL character; it may be empty (`''`).

input_dir
: The input directory in which the header is read, for this call only. Default `[]`: the current setting.

network
: `true` to allow reading the header from a URL, for this call only, `false` to forbid it. Default `[]`: the current setting.

dir A folder, which must exist, in which to write the changed header, under the same file name, instead of changing the header itself (which is then unchanged); the signal files are not copied. An unchanged header is also written there. Default `[]` (or `''`): the header itself is changed.

Returns Nothing.

Errors Identifier `wfdbx:set_metadata`: `key` or `value` is missing, not a string (or `uint8` vector) or not valid; the header has more than one entry for `key` ("... remove them first"); it has no record line; it was read from a URL and `'dir'` is not given; `record` is empty or `'-'`; `'dir'` is not a string; or an option is not valid. Identifier `wfdbx:set_metadata:file`: the header cannot be read, or the changed header cannot be written ("can't write header for record REC"); or `'dir'` does not exist ("directory DIR does not exist").

```
wfdbx.write_record('pat', zeros(360, 1), 360, 'gains', 200, 'adcres', 12, ...
    'units', 'mV', 'metadata', {'Age', '63'; 'Sex', 'F'});
wfdbx.set_metadata('pat', 'AGE', '64');                % replaces "# Age: 63"
wfdbx.set_metadata('pat', 'Diagnosis', 'sinus rhythm, no ectopy');   % a new entry
M = wfdbx.read_metadata('pat')';
fprintf('%s: %s\n', M{:});
% AGE: 64
% Sex: F
% Diagnosis: sinus rhythm, no ectopy

% Write the changed header to another folder; pat.hea is left unchanged.
mkdir('changed');
wfdbx.set_metadata('pat', 'Sex', 'M', 'dir', 'changed');
fprintf('%s %s\n', wfdbx.get_metadata('pat', 'Sex'), ...
    wfdbx.get_metadata('changed/pat', 'Sex'));         % F M
```

---

wfdbx.remove_metadata

Remove a metadata entry

```
n = wfdbx.remove_metadata(record, key, 'input_dir', [], 'network', [], 'dir', [])
```

Removes every entry for `key` (ignoring letter case) from the header of a record, line ending included, so it also repairs a header that has more than one. Other lines are kept, byte for byte. Nothing is written if the header has no entry for `key`, unless `'dir'` is given.

`record`
: Record name (as for `wfdbx.set_metadata`).

`key` The key (as for `wfdbx.set_metadata`).

`input_dir`
: The input directory in which the header is read, for this call only. Default `[]`: the current setting.

`network`
: `true` to allow reading the header from a URL, for this call only, `false` to forbid it. Default `[]`: the current setting.

`dir` A folder, which must exist, in which to write the changed header instead of changing the header itself. Default `[]`: the header itself is changed.

Returns The number of lines removed (0 if the header has no entry for `key`).

Errors As `wfdbx.set_metadata`, with the identifiers `wfdbx:remove_metadata` and `wfdbx:remove_metadata:file`, except that more than one entry for `key` is not an error.

```
wfdbx.write_record('pat', zeros(360, 1), 360, 'gains', 200, 'adcres', 12, ...
    'units', 'mV', 'metadata', {'Age', '63'; 'Sex', 'F'});
disp(wfdbx.remove_metadata('pat', 'sex'));      % 1
disp(wfdbx.remove_metadata('pat', 'Sex'));      % 0: there is no such entry now
disp(wfdbx.read_metadata('pat'));               % {'Age', '63'}
```

wfdbx.update_metadata Make several changes to the metadata

```
wfdbx.update_metadata(record, changes, 'input_dir', [], 'network', [], 'dir', [])
```

Makes several changes to the metadata of a record, in one write of the header. Each change is applied in order, as `wfdbx.set_metadata` does, or, for a value of `[]`, as `wfdbx.remove_metadata` does. All the changes are checked first: if one is not valid, none is made.

record
: Record name (as for `wfdbx.set_metadata`).

changes
: An N-by-2 cell array with one change per row, `{KEY, VALUE; ...}`; a value of `[]` (an empty double) removes the entry, and `''` sets an empty value. No two keys may be equal without regard to letter case. An empty cell array (`{}` or `cell(0, 2)`) changes nothing.

input_dir
: The input directory in which the header is read, for this call only. Default `[]`: the current setting.

network
: `true` to allow reading the header from a URL, for this call only, `false` to forbid it. Default `[]`: the current setting.

dir
: A folder, which must exist, in which to write the changed header instead of changing the header itself. Default `[]`: the header itself is changed.

Returns Nothing.

Errors As `wfdbx.set_metadata`, with the identifiers `wfdbx:update_metadata` and `wfdbx:update_metadata:file`; also an argument error if `changes` is not an N-by-2 cell array, or if two of its keys differ only in letter case.

```
wfdbx.write_record('pat', zeros(360, 1), 360, 'gains', 200, 'adcres', 12, ...
    'units', 'mV', 'metadata', {'Age', '63'; 'Sex', 'F'});
wfdbx.update_metadata('pat', {'Age', '64'; 'Sex', []; 'Dx', '164889003'});
M = wfdbx.read_metadata('pat')';
fprintf('%s: %s\n', M{:});      % Age: 64, then Dx: 164889003
try
    wfdbx.update_metadata('pat', {'Dx', '426783006'; 'dx', []});
catch err
    disp(err.identifier);       % wfdbx:update_metadata
    disp(err.message);          % metadata key 'dx' is given more than once (...)
end
disp(wfdbx.get_metadata('pat', 'Dx'));          % 164889003: nothing was changed
```

11.3.6 The result structures

The readers return structures, whose fields are listed in Tables 11.1 to 11.3; `wfdbx.check_record` returns a structure that holds a structure array of issues (Tables 11.4 and 11.5). The signals of a record are described by `wfdbx.siginfo` structures (Table 11.6), which are those of the WFDB library (Section 11.4). The fields are ordinary structure fields, which may be changed.

Table 11.1 The fields of the result of `wfdbx.read_record`

Field	Contents
`name`	the record name, as given to `wfdbx.read_record`
`fs`	the sampling frequency in Hz, or 0 if the header gives none; with `'highres', true`, the highest sampling frequency of a multi-frequency record
`signals`	a matrix of doubles with one row per sample and one column per signal read: ADC units, or physical units with `'physical', true`; invalid samples are `wfdbx.WFDB_INVALID_SAMPLE()` (−32768), or NaN in physical units
`siginfo`	a 1-by-N `wfdbx.siginfo` structure array, one per signal read, in the order of the columns; `desc` has no white space at either end, and `adcres` is 0 if the header does not give it
`names`	a 1-by-N cell array of the signal descriptions, such as `'MLII'`, without the white space that a header may put at either end of them
`units`	a 1-by-N cell array of the physical units, with `[]` where the header gives none
`gains`	a row vector of the gains, in ADC units per physical unit; 0 means that the signal is uncalibrated
`baselines`	a row vector of the ADC values that correspond to 0 physical units
`base_time`	the start time and date as text, to the second, such as `'[14:00:03 21/06/2024]'` (`'[14:00:03]'` without a date), or `[]` if the header gives none
`base_time_ms`	the start time in milliseconds after midnight, such as `50403500` for 14:00:03.500, or `[]` if `base_time` is `[]`
`base_date`	the start date as `[year month day]`, or `[]` if the header gives none
`sampfrom`	the number of the first sample returned
`comments`	a cell column of the info strings of the header (without the `#`): its lines that begin with `#` after the record line, found as the header parser finds it (an empty or blank first line is not the record line; in legacy mode, as the C function `getinfo` finds it: the first line that does not begin with `#`)
`checksum_ok`	`false` if the samples do not match the checksums of the header; checksums are checked only when a read reaches the end of the record, so a shorter read gives `true`

Table 11.2 The fields of the result of `wfdbx.read_header`

Field	Contents
`name`	the record name, as given to `wfdbx.read_header`
`fs`	the sampling frequency in Hz, or 0 if the header gives none
`counter_freq`	the counter frequency, as `wfdbx.getcfreq()` returns it (the sampling frequency if the header gives none)
`base_counter`	the base counter value, as `wfdbx.getbasecount()` returns it
`nsamples`	the number of samples per signal, or 0 if the header does not give it
`nsig`	the number of signals
`siginfo`	a 1-by-N `wfdbx.siginfo` structure array, one per signal; `desc` has no white space at either end, and `adcres` is 0 if the header does not give it
`base_time`, `base_time_ms`, `base_date`	the start time and date, as for `wfdbx.read_record`
`comments`	a cell column of the info strings of the header (without the `#`), as for `wfdbx.read_record`

Field	Contents
`metadata`	the metadata entries of the header, an N-by-2 cell array `{KEY, VALUE; ...}` in file order, as `wfdbx.read_metadata` returns them: from the `.hea` file only; `cell(0, 2)`, without an error, for an EDF record and for a header read from the standard input

Table 11.3 The fields of the result of `wfdbx.read_annotations`: one row per annotation, in file order

Field	Contents
`record`	the record name, as given
`annotator`	the annotator name, such as `'atr'`
`samples`	a column of the annotation times, in samples
`codes`	a column of the annotation codes, such as `wfdbx.NORMAL()` or `wfdbx.PVC()`
`symbols`	a cell column of the mnemonics of the codes, such as `'N'`, `'V'` or `'+'`
`subtypes`	a column of the subtypes
`chans`	a column of the channel numbers
`nums`	a column of the annotator numbers
`aux`	a cell column of the auxiliary texts (`char` vectors), with `[]` where there is none
`fs`	the frequency in which `samples` are counted, normally the sampling frequency of the record; 0 if neither the record, nor the file, nor the option `'fs'` of `wfdbx.read_annotations` gives one

Table 11.4 The fields of the result of `wfdbx.check_record`

Field	Contents
`record`	the record name, as given
`ok`	`true` if no issue has the severity `'error'`
`issues`	an N-by-1 structure array of the issues (Table 11.5), in the order of the checks
`lines`	the text form of the report, a cell column: one line per issue, `'SEVERITY: WHERE: MESSAGE'` (`'SEVERITY: MESSAGE'` when `where` is empty), or the one line `'no problems found'`

Table 11.5 The fields of an issue, a problem found by `wfdbx.check_record`

Field	Contents
`severity`	`'error'`, `'warning'` or `'info'`
`code`	a stable identifier of the kind of problem, such as `'checksum'` or `'gain_missing'` (Table 6.7)
`where`	where the problem is: `''` (the record), `'signal N'`, `'segment SEG'`, `'segment SEG, signal N'` or `'annotator ANN'`
`message`	the description of the problem

Table 11.6 The fields of a `wfdbx.siginfo` structure, the specification of a signal

Field	Contents
`fname`	the name of the signal file, as the header gives it
`desc`	the signal description, such as `'MLII'`, or `[]`
`units`	the physical units, such as `'mV'`, or `[]` if not specified

Field	Contents
`gain`	the gain, in ADC units per physical unit (0: uncalibrated)
`initval`	the value of the first sample
`group`	the signal group number, from 0 (the signals of one group share one file)
`fmt`	the storage format, such as `16`, `212` or `516`
`spf`	the number of samples per frame (more than 1 for the faster signals of a multi-frequency record)
`bsize`	the block size, in bytes (0 for ordinary files)
`adcres`	the ADC resolution, in bits (0: not specified)
`adczero`	the ADC output for an input of 0 V
`baseline`	the ADC value that corresponds to 0 physical units
`nsamp`	the number of samples (0 if unknown)
`cksum`	the 16-bit checksum of the samples

11.3.7 Warnings and errors

Errors. The high-level functions report errors with `error`, with a message and an identifier (Table 11.7), so that `err.identifier` of a `try/catch` tells which function failed, and, for the metadata functions, whether an argument was invalid or a file could not be read or written. Each function checks its arguments before it creates or changes a file, so an invalid argument leaves the files as they were. After an error, as after a successful call, the settings of the call (`'input_dir'`, `'network'`, `'dir'`, `'legacy'`), the getvec mode and the time resolution of output annotators are as they were before it (Section 11.3.1). The library may also print a message on the standard error when it cannot open a file (as in `init: can't open header for record REC`); `wfdbx.wfdbquiet()` turns these messages off (Section 11.4).

Table 11.7 The error and warning identifiers of the high-level API

Identifier	Issued when
`wfdbx:NAME`	an error of the function `wfdbx.NAME`: for `read_record`, `read_header`, `read_annotations`, `write_record`, `write_annotations` and `check_record`, any error (a file that cannot be found, read, created or written completely; an invalid option or argument; a value that the operation needs and the record does not give, such as a time in seconds without a sampling frequency); for the five metadata functions, an invalid argument: an invalid key or value, a record name that is empty or `'-'`, a key that the header has more than once (`set_metadata`, `update_metadata`), a header without a record line, a header read from a URL to be changed without `'dir'`, or an invalid option
`wfdbx:NAME:file`	a metadata function cannot read or write a file: a header that cannot be found or read, a changed header that cannot be written, or a `'dir'` that does not exist
`wfdbx:checksum`	(a warning) the samples read by `wfdbx.read_record` do not match the checksums of the header
`wfdbx:duplicate_key`	(a warning) a header has more than one metadata entry for a key

The checksum warning. When `wfdbx.read_record` reads to the end of a record whose samples do not match the checksums of its header, it returns all the samples and issues the warning `wfdbx:checksum`: "checksum error in record REC: the signal data do not match the checksums in its header". `'checksum_warning', false` turns the warning off; `rec.checksum_ok` is `false` either way. This applies only when every sample could be read: a signal file that ends before the length that its header gives is an error ("can't read signal data of record REC (getvec status -3)"), even

though the checksum, verified at the frame in which the file ended, does not match either; `wfdbx.check_record` reports it as `length_short`, not as a checksum error.

The duplicate-key warning. When a header holds two metadata entries whose keys differ only in letter case (written by other software), `wfdbx.read_metadata`, `wfdbx.get_metadata` and `wfdbx.read_header` use the first and issue the warning `wfdbx:duplicate_key`: "header of record REC has more than one entry for key KEY (lines L1, L2); the first is used". `wfdbx.set_metadata` and `wfdbx.update_metadata` then refuse to change that key (an argument error), and `wfdbx.remove_metadata` removes all of its entries.

Both are issued with `warning`, so `warning('off', 'wfdbx:checksum')` turns one off, `lastwarn` returns the last one, and `warning('error', ...)` (MATLAB) makes it an error, as for any other warning:

```
% A record whose last sample is changed after it was written: the checksum fails.
wfdbx.write_record('ck', (0:99)', 100, 'gains', 100, 'adcres', 12, 'units', 'mV');
fid = fopen('ck.dat', 'r+b');
fseek(fid, -2, 'eof');
fwrite(fid, [0 0], 'uint8');
fclose(fid);
lastwarn('');
rec = wfdbx.read_record('ck');                  % prints the checksum warning
fprintf('%d %d %d\n', size(rec.signals), rec.checksum_ok);   % 100 1 0: all the samples
[msg, id] = lastwarn();
disp(id);                                       % wfdbx:checksum
rec = wfdbx.read_record('ck', 'checksum_warning', false);    % no warning

% Two entries for one key: the first is used, with a warning.
wfdbx.write_record('dup', zeros(10, 1), 100, 'gains', 0, 'adcres', 12, ...
    'comments', {' Age: 63', ' AGE: 64'});
disp(wfdbx.get_metadata('dup', 'age'));         % 63, with the warning
[msg, id] = lastwarn();
disp(id);                                       % wfdbx:duplicate_key
try
    wfdbx.set_metadata('dup', 'Age', '65');
catch err
    fprintf('%s: %s\n', err.identifier, err.message);
    % wfdbx:set_metadata: header of record dup has more than one entry for key Age
    % (lines 3, 4); remove them first
end
disp(wfdbx.remove_metadata('dup', 'age'));      % 2
```

```
try
    wfdbx.read_record('nosuchrecord');
catch err
    fprintf('%s: %s\n', err.identifier, err.message);
    % wfdbx:read_record: can't read record nosuchrecord
end
try
    wfdbx.read_metadata('nosuchrecord');
catch err
    fprintf('%s: %s\n', err.identifier, err.message);
    % wfdbx:read_metadata:file: can't read header for record nosuchrecord
end
wfdbx.write_record('raw', (0:9)', 100, 'gains', 0, 'adcres', 12);   % uncalibrated
try
    wfdbx.read_record('raw', 'physical', true);
catch err
```

```
    fprintf('%s: %s\n', err.identifier, err.message);
    % wfdbx:read_record: signal 0 of record raw is uncalibrated
end
```

11.4 The WFDB library functions in MATLAB and Octave

Every function of the WFDB library is a function of the package `wfdbx` under its C name, with the same arguments, in the same order, and the same return values, so that programs and documentation written for the C library carry over directly. The functions, their arguments and their results are documented once, for all the languages, in Chapter 13; this section gives the rules by which the C declarations become MATLAB functions (see also Section 13.1.6). `help wfdbx.NAME` gives the documentation of each function, including what WFDB-x changes in it.

11.4.1 Conventions

Names. The functions keep their C names, with the prefix of the package: `wfdbx.isigopen`, `wfdbx.getvec`, ... The constants (`WFDB_READ`, `WFDB_INVALID_SAMPLE`, `WFDB_HIGHRES`, ...) and the annotation codes (`NORMAL`, `PVC`, ..., Chapter 15) are functions without arguments: `wfdbx.WFDB_READ()`, `wfdbx.NORMAL()`; `C = wfdbx.ecgcodes()` returns all the annotation codes as the fields of one structure (`C.NORMAL`, `C.PVC`, ...), which is faster in loops. The calibration constants (`WFDB_AC_COUPLED`, ...) and `WFDB_FMT_LIST` are not defined; their values are given in Table 13.2. The macros of `<wfdb/ecgmap.h>` are functions, under both their macro names and the names of the functions behind them: `wfdbx.isqrs` and `wfdbx.wfdb_isqrs`, `wfdbx.map1` and `wfdbx.wfdb_map1`, and so on.

NULL is `[]`. A C `NULL` string, as argument or result, is `[]` (an empty double), and an empty C string is ''; `ischar` tells them apart. `wfdbx.sampfreq([])` returns the frequency of the open record, `wfdbx.getinfo([])` continues with the info strings of the last record, and `wfdbx.getinfo` returns `[]` after the last one.

Arrays and output arguments. Where a C function fills an array or a structure that the caller passes, the MATLAB function returns it as an additional output value, and omits it from the arguments:

```
[nsig, si] = wfdbx.isigopen(record, nsig)     % si: a struct array of siginfo
[nsig, si] = wfdbx.osigopen(record, nsig)
[stat, si] = wfdbx.wfdbinit(record, aiarray, nann, nsig)
[stat, v] = wfdbx.getvec(v0)                  % v0 (optional): the vector C leaves unchanged
[stat, v] = wfdbx.getframe(v0)
[stat, annot] = wfdbx.getann(an, annot0)      % annot0 (optional), as v0
[stat, cal] = wfdbx.getcal(desc, units)
[stat, si] = wfdbx.setheader(record, si, nsig)
segs = wfdbx.getseginfo()                     % a struct array of seginfo
[r, argv] = wfdbx.wfdbx_legacy_args(argv)     % argv: a cell array
```

With `nsig` ≤ 0, `wfdbx.isigopen` reads the header only, and `si` describes `-nsig` signals. Arrays of structures passed in are struct arrays: `wfdbx.annopen(record, wfdbx.anninfo('atr', wfdbx.WFDB_READ()))`, `wfdbx.osigfopen(si, nsig)`. `wfdbx.setmsheader(record, segnames)` takes a cell array of segment names.

Numbers and strings. Samples, times, counts and frequencies are doubles, exact for every 32-bit sample and for times up to 2^{53}. Strings are `char` row vectors whose characters are the bytes of the C strings (codes 0 to 255); record names may also be MATLAB strings. The `aux` field of an annotation is `[]`, or a `uint8` row vector whose first element is the length of the text that follows, as in C: `char(a.aux(2:1 + a.aux(1)))` is the text.

Return values and errors. The functions return what the C functions return: a status (negative on errors), a number of signals, a frequency, a string or `[]`. Errors are reported by these values and by a message on the standard error, as in C, never by MATLAB errors. `wfdbx.wfdberror()` returns the last message; `wfdbx.wfdbquiet()` turns the messages off and `wfdbx.wfdbverbose()` on again; `wfdbx.wfdb_error(fmt, ...)` reports a message as the library does, formatting as C's `printf`.

State. As in C, the library keeps its state in the session (in `persistent` variables): the open record, its input signals, the output signals and the annotators remain open between calls, until `wfdbx.wfdbquit()` closes them. `wfdbx.wfdbquit()` leaves the input and output directories, the network setting, legacy mode, the getvec mode and the time resolution of output annotators (which opening an EDF record sets, as in C) as they are; `clear functions` (or `clear all`) resets the whole state. The high-level functions of Section 11.3 (except the metadata functions) call `wfdbx.wfdbquit()`, which closes anything opened with the library functions.

Additions. `[v, stat, last] = wfdbx.getframes(n, minstat)` reads up to `n` sample vectors at once (`n` may be `Inf`), as `n` calls of `wfdbx.getvec` would, but decoding whole blocks of the signal files: `v` is a matrix with one row per vector and one column per signal, `stat` the status of the first vector that could not be read, or the number of signals if all `n` were read, and `last` the vector filled by the call that stopped the reading (Section 13.11). `stat = wfdbx.putframes(M)` writes the rows of `M`, as `wfdbx.putvec` for each row would. Calling `wfdbx.getvec` sample by sample is much slower in MATLAB and Octave than these functions, and than the high-level functions.

11.4.2 The structures

The structures of the C library are MATLAB structures with the names of the fields of the C structures (Table 11.8), made with C's zero values by these functions:

```
si = wfdbx.siginfo(n)            % 1-by-n struct array (n = 1 by default); spf is 1
s = wfdbx.seginfo(n)             % 1-by-n struct array (n = 1 by default)
ai = wfdbx.anninfo(name, stat)   % one structure
a = wfdbx.annotation(n)          % 1-by-n struct array (n = 1 by default)
cal = wfdbx.calinfo()            % one structure
```

Table 11.8 The C structures in MATLAB and Octave

Constructor (C structure)	Fields
`wfdbx.siginfo` (`WFDB_Siginfo`)	`fname`, `desc`, `units` (`[]` or `char`), `gain`, `initval`, `group`, `fmt`, `spf`, `bsize`, `adcres`, `adczero`, `baseline`, `nsamp`, `cksum` (Table 11.6)
`wfdbx.seginfo` (`WFDB_Seginfo`)	`recname` (the name of the segment), `nsamp` (its number of samples), `samp0` (the number of its first sample in the record)
`wfdbx.anninfo` (`WFDB_Anninfo`)	`name` (the annotator name, the suffix of the file), `stat` (`wfdbx.WFDB_READ()` or `wfdbx.WFDB_WRITE()` for the MIT format, `wfdbx.WFDB_AHA_READ()` or `wfdbx.WFDB_AHA_WRITE()` for the AHA format)
`wfdbx.annotation` (`WFDB_Annotation`)	`time` (in sample intervals), `anntyp` (the annotation code), `subtyp`, `chan`, `num`, `aux` (`[]`, or a `uint8` row vector whose first element is the length of the text that follows)
`wfdbx.calinfo` (`WFDB_Calinfo`)	`low`, `high` (the levels of the calibration pulse, in physical units), `scale` (physical units per cm), `sigtype` (matched against the beginning of a signal description), `units`, `caltype`

11.4.3 The settings functions

The functions of Table 11.9 are additions of WFDB-x to the library: they set where files are read and written, network access and legacy mode, for the high-level functions, the library functions and the tools alike (Chapters 3 and 4), and give the path under which a program opens a file that it reads or creates itself, as the tools do (`wfdbx.wfdbx_input_path`, `wfdbx.wfdbx_output_path`; Section 3.3.2). Each `set` function returns the previous setting, so that it can be restored. The C function `wfdbx.setwfdb(p)` sets the input directory too (`p` must name at most one directory), and `wfdbx.getwfdb()` returns it; in legacy mode they set and return the WFDB path, as in C.

Table 11.9 The settings functions

Function	Effect and result
`previous = wfdbx.wfdbx_set_input_dir(d)`	sets the input directory: a local folder, a ~ path or a URL; `[]` or '' restores the default (`WFDBX_INPUT_DIR`, or else the current folder). Returns the previous input directory, or `[]` if none was set
`d = wfdbx.wfdbx_get_input_dir()`	returns the input directory, or `[]` if none is set
`previous = wfdbx.wfdbx_set_output_dir(d)`	sets the output directory: a local folder or a ~ path (a URL is refused, with a message, and the setting is unchanged); `[]` or '' restores the default (`WFDBX_OUTPUT_DIR`, or else the current folder). Returns the previous output directory, or `[]`
`d = wfdbx.wfdbx_get_output_dir()`	returns the output directory, or `[]` if none is set
`previous = wfdbx.wfdbx_set_network(on)`	turns network access on (`true` or 1) or off (`false` or 0); `[]` or a negative number restores the default (on if `WFDBX_NETWORK` is a nonzero number). Returns the previous setting, 1 or 0
`on = wfdbx.wfdbx_get_network()`	returns 1 if network access is on, 0 otherwise
`previous = wfdbx.wfdbx_set_legacy(on)`	turns legacy mode on (`true` or 1) or off, for the whole library. Returns the previous setting, 1 or 0
`on = wfdbx.wfdbx_get_legacy()`	returns 1 if legacy mode is on, 0 otherwise
`[r, argv] = wfdbx.wfdbx_legacy_args(argv)`	if `argv{2}` (`argv`: a cell row of the arguments, program name first) is `'--legacy'`, removes it, turns legacy mode on and returns `r` = 1; otherwise returns `argv` unchanged and `r` = 0
`r = wfdbx.wfdbx_time_needs_freq(string)`	returns 1 if `wfdbx.strtim(string)` needs a sampling frequency, 0 otherwise
`n = wfdbx.wfdbx_write_errors()`	returns the number of write errors since the program started (output files that could not be written, flushed or closed)
`wfdbx.wfdbx_note_write_error()`	counts a write error, for a program that writes a file itself
`p = wfdbx.wfdbx_input_path(name)`	returns the path under which a program opens a file that it reads itself: in normal mode, a relative name under the input directory, a ~ name expanded, an absolute name as given; in legacy mode, the name as given. `[]`, '' and `'-'` are returned as given; `[]` for a URL, or a name that becomes one under the input directory
`p = wfdbx.wfdbx_output_path(name)`	returns the path under which a program creates a file that it writes itself: in normal mode, a relative name under the output directory (whose missing folders are then created), a ~ name expanded, an absolute name as given; in legacy mode, any name under the output directory, as a plain prefix. `[]`, '' and `'-'` are returned as given; `[]` for a URL

11.4.4 Examples

Reading the samples and the annotations of a record, as a C program does (the record `100s` is read from the current folder):

```
[nsig, si] = wfdbx.isigopen('100s', 2);         % open 2 signals
fprintf('%d %s %d %g\n', nsig, si(1).desc, si(1).fmt, si(1).gain);   % 2 MLII 212 200
fprintf('%g\n', wfdbx.sampfreq([]));            % 360: the frequency of the open record

wfdbx.isigsettime(wfdbx.strtim('0:30'));        % skip to 30 seconds
[stat, v] = wfdbx.getvec();                     % one sample per signal
fprintf('%d %d %g\n', v(1), v(2), wfdbx.aduphys(0, v(1)));      % 947 968 -0.385

[frames, stat] = wfdbx.getframes(360);          % the next second, as a matrix
fprintf('%d %d %d\n', size(frames), stat);      % 360 2 2

wfdbx.annopen('100s', wfdbx.anninfo('atr', wfdbx.WFDB_READ()));
n = 0;
[stat, a] = wfdbx.getann(0);
while stat == 0                                 % 0: an annotation was read; -1: the end
    if n < 2
        fprintf('%s %s\n', wfdbx.mstimstr(a.time), wfdbx.annstr(a.anntyp));
    end
    n = n + 1;
    [stat, a] = wfdbx.getann(0);
end
fprintf('%d annotations\n', n);                 % 75 annotations

wfdbx.wfdbquiet();                              % no messages on the standard error
stat = wfdbx.isigopen('nosuchrecord', 0);
fprintf('%d %s', stat, wfdbx.wfdberror());
% -1 init: can't open header for record nosuchrecord
wfdbx.wfdbverbose();
wfdbx.wfdbquit();                               % close everything
```

Writing a record and an annotation file, sample by sample and annotation by annotation:

```
out = wfdbx.siginfo(1);
out.fname = 'ramp.dat';
out.desc = 'ramp';
out.units = 'mV';
out.gain = 200;
out.fmt = 16;
out.adcres = 12;
disp(wfdbx.osigfopen(out, 1));      % 1: one output signal
wfdbx.setsampfreq(250);
for k = 0:499
    wfdbx.putvec(k);
end
disp(wfdbx.newheader('ramp'));      % 0: ramp.hea is written
wfdbx.annopen('ramp', wfdbx.anninfo('qrs', wfdbx.WFDB_WRITE()));
a = wfdbx.annotation();
a.anntyp = wfdbx.NORMAL();
for t = [100 350]
    a.time = t;
    wfdbx.putann(0, a);
end
wfdbx.wfdbquit();                   % completes ramp.dat and ramp.qrs
```

```
ann = wfdbx.read_annotations('ramp', 'qrs');
disp(ann.samples');                % 100 350
```

11.5 The command-line tools in MATLAB and Octave

The 33 command-line tools (`rdsamp`, `rdann`, `wrsamp`, `xform`, ...) are functions of the package `wfdbx.apps`, one file per tool in `+wfdbx/+apps`. The tools, their options, their output and their exit status are those of the C tools, and are documented in Chapter 14; `help wfdbx.apps.TOOL` shows the usage of a tool.

From the shell, with Octave. The Octave script `matlab/bin/wfdbx_cli.m` runs a tool: its first argument names the tool, and the others are the tool's arguments. It adds `matlab` to the path, and exits with the tool's status. An alias makes the command shorter:

```
alias wfdbx='octave-cli --no-gui --norc -q /path/to/wfdb-x/matlab/bin/wfdbx_cli.m'
export WFDBX_INPUT_DIR=/path/to/wfdb-x/tests/data
wfdbx rdsamp -r 100s -t 10 -p     # the first 10 seconds of 100s, in physical units
wfdbx rdann -r 100s -a atr        # its annotations
wfdbx rdsamp --legacy -r 100s     # legacy mode: --legacy as the first argument
```

The script also takes these arguments, instead of a tool:

Table 11.10 The arguments of the script `wfdbx_cli.m` (as `wfdbx`)

Command	Effect
`wfdbx TOOL [ARGUMENTS ...]`	runs the tool `TOOL`, and exits with its status
`wfdbx --list`	lists the names of the tools, one per line
`wfdbx --version`	prints the version, `WFDB-x 1.2.3`
`wfdbx --help`, `wfdbx -h`	prints the usage

Without arguments, the script prints the usage and exits with status 1; an unknown tool is an error ("wfdbx: unknown tool 'TOOL' (try 'wfdbx --list')", status 1), and a MATLAB error in a tool is reported as `wfdbx: error: MESSAGE` (status 1). When an output file cannot be written completely (a full disk, for example), or the standard output cannot be written ("TOOL: write error on standard output"), a tool exits with status 1; in legacy mode, it keeps the original status, as in C. Octave starts in a fraction of a second, so the script suits everyday use.

From the shell, with MATLAB. The shell script `matlab/bin/wfdbx_matlab.sh` runs a tool with `matlab -batch`, or the program named by the environment variable `WFDBX_MATLAB`. It saves its standard input to a temporary file (named by the environment variable `WFDBX_STDIN`) before it starts MATLAB, so a script whose standard input stays open runs it with `< /dev/null`, and it passes the arguments in hexadecimal, so that any bytes arrive unchanged. MATLAB starts slowly, so the script is mainly useful for testing.

```
export WFDBX_INPUT_DIR=/path/to/wfdb-x/tests/data
/path/to/wfdb-x/matlab/bin/wfdbx_matlab.sh rdsamp -r 100s -t 10 -p
```

In a session. Each tool is a function that takes the arguments as a cell array (without the program name), writes to the standard output and error, and returns the exit status:

```
status = wfdbx.apps.rdsamp({'-r', '100s', '-t', 's2'})    % the first 2 samples
```

A call of `exit` in a C tool ends the function, not the session.

Environment variables. The tools, like the rest of the package, use the environment variables of Table 11.11, also for the files that they read or create themselves (Section 3.3.2): the `-i` file of `wrsamp`, `edf2mit`, `wav2mit`, `a2m` and `ad2m`, the input files of `ahaecg2mit` and the `-S` script of `xform` are read from the input directory, and the WAV file of `mit2wav`, the EDF file of `mit2edf`, the master header of `snip -m` and the `.des`, `.ecg`, `.key` and `.txt` files of `parsescp` are created in the output directory. Such a file cannot be read from a URL. In legacy mode, the files that a tool reads itself are opened as named, and the output directory, if set, is a plain prefix of every output name. In Octave and in MATLAB, `getenv` returns '' for an unset and an empty variable alike, so an empty variable counts as unset.

Table 11.11 The environment variables used by the MATLAB/Octave package

Variable	Meaning
`WFDBX_INPUT_DIR`	the input directory (a folder, a ~ path or a URL), when `wfdbx.wfdbx_set_input_dir` does not set one
`WFDBX_OUTPUT_DIR`	the output directory, when `wfdbx.wfdbx_set_output_dir` does not set one
`WFDBX_NETWORK`	network access is on if it is a nonzero number, when `wfdbx.wfdbx_set_network` does not set it
`WFDBANNSORT`	`0` to keep annotation files in the order written (no sorting when they are closed)
`WFDBGVMODE`	the default getvec mode (`setgvmode` with a negative mode)
`WFDBCAL`	the calibration file read by `wfdbx.calopen([])` (default `wfdbcal`)
`WFDB`	the WFDB path, used in legacy mode only (outside legacy mode, it is ignored, with a message)
`WFDB_FLAC_CHECK_MD5`	if set, the MD5 signature of FLAC signal files is verified when they are read
`WFDBX_MATLAB`	the MATLAB command that `wfdbx_matlab.sh` runs (default `matlab`)

Tools that write signal parameters require them as options: `wrsamp` (with `-o`) requires `-F` (frequency), `-G` (gain), `-R` (ADC resolution) and, for calibrated signals, `-U` (units); `wav2mit` requires `-G`, and `-U` unless the gain is 0; `ad2m` and `ahaecg2mit` require `-F`, `-G`, `-R`, and `-U` unless the gain is 0; `a2m`, `rdann`, `ann2rr`, `sortann` and `m2a` require `-F` when the record's header gives no frequency (Chapter 14).

11.6 Testing

The unit tests of the MATLAB/Octave implementation are in `matlab/tests`: one function `test_*.m` per group, run by `run_tests`. From the root of the repository:

```
octave-cli --no-gui --norc -q --eval "cd matlab/tests; run_tests"           # Octave
matlab -batch "cd matlab/tests; run_tests"                                  # MATLAB
octave-cli --no-gui --norc -q --eval "cd matlab/tests; run_tests('metadata')"  # by name
```

`run_tests` prints `ok NAME` or `FAIL NAME: MESSAGE` for each test, then a summary, and ends with an error (so that the command exits with status 1) if a test failed; `run_tests(PATTERN)` runs the tests whose names match a regular expression, and `n = run_tests(...)` returns the number of failures. The tests cover the C standard library emulation, the WFDB library functions (with expected values printed by the C library), the high-level functions, the bulk readers and writers against the per-sample functions, the FLAC codec, finding files, `wfdbx.check_record`, and the metadata functions, with the cases shared by the seven languages (in `tests/metadata`: results, warnings, and the headers, byte for byte). They run in temporary folders, with copies of the test records

of `tests/data`. The tests of network access use a local HTTP server on 127.0.0.1 only (a Python script, started with `python3`; without it, they are skipped).

The cross-language conformance suite runs every tool through `matlab/bin/wfdbx_cli.m` and compares its results, byte for byte, with those of the original C programs (1155 cases), and, with `--legacy`, in legacy mode (886 cases). From the root of the repository:

```
PYTHONPATH=$PWD/python/src python3 tests/conformance/harness.py check --impl octave -j8
PYTHONPATH=$PWD/python/src python3 tests/conformance/harness.py check --impl octave -j8 --legacy
```

The implementation `matlab` of the harness runs the tools through `matlab/bin/wfdbx_matlab.sh` instead.

CHAPTER 12

Kotlin

The Kotlin implementation of WFDB-x is the Kotlin Multiplatform library `wfdbx`, with the Gradle coordinates `io.github.alphanumericslab:wfdbx:1.2.3`, in the directory `kotlin` of the repository. It is written in common Kotlin code, which runs on every JVM, on Android and on iOS, with the Kotlin standard library as its only dependency, and never calls the C library. It is meant for mobile applications and JVM programs that read and write WFDB records on the device, with the behavior of the other implementations of WFDB-x, without native libraries, JNI or another runtime.

It offers WFDB-x in three ways:

- **The high-level API** (Section 12.3): functions that read and write whole records, headers, annotation files and header metadata in one call, with named arguments and default values; the samples are an `IntArray` (and a `DoubleArray` in physical units), frame by frame, and errors throw a `WfdbException`. They read signal files in large blocks, and are the fastest way to read a record. They are an addition of WFDB-x: the C package has no equivalent. They are the recommended interface for new code and apps.
- **The WFDB library functions** under their C names (`isigopen`, `getvec`, `getann`, `putann`, ...), with the behavior of the C library (Section 12.4). They are documented once, for all the languages, in Chapter 13.
- **The command-line tools** (`rdsamp`, `rdann`, `wrsamp`, `xform` and 29 others), with the options, output and exit status of the C tools, run on the JVM through the launcher `kotlin/bin/wfdbx` (Section 12.5). They are documented in Chapter 14.

The high-level functions return exactly the samples and annotations that the library functions (and the C library) give.

12.1 Installation and requirements

12.1.1 Requirements

The library is built with Gradle 8.14.3 (the Gradle wrapper of the repository, `kotlin/gradlew`, which downloads itself) and Kotlin 2.1.21, and needs a JDK 11 or later to build. It has no dependency besides the Kotlin standard library (and `kotlin-test` for the tests); the FLAC formats 508, 516 and 524 are read and written by WFDB-x's own codec. The targets are:

- `jvm()`: bytecode for Java 11, which runs on every current JVM and on Android (API level 26 or later; the library uses `java.nio.file`);
- `iosArm64()`, `iosSimulatorArm64()` and `iosX64()`: declared, with a small platform layer for iOS (POSIX files and `NSURLSession`); they can only be built on macOS;
- `linuxX64()`, built only with the Gradle option `-Pwfdbx.linux=true`, with the Kotlin/Native toolchain.

The native code (iOS and Linux) is not compiled by the tests of the repository, which build the JVM code and check that the common code, which holds all of the library's logic, uses only common APIs.

12.1.2 Building and installing the library

The library is built from a copy of the repository and installed in the local Maven repository (`~/.m2`). In the directory `kotlin`:

```
./gradlew publishToMavenLocal
```

A Gradle project (an Android app, a JVM program, or the common source set of a Kotlin Multiplatform project) that lists `mavenLocal()` among its repositories then adds it to its dependencies:

```
dependencies {
    implementation("io.github.alphanumericslab:wfdbx:1.2.3")
}
```

The build downloads the Kotlin Gradle plugin from Maven Central. Where Maven Central cannot be reached, the Gradle property `mavenMirror` names another Maven repository, for plugins and dependencies alike, on the command line or in `~/.gradle/gradle.properties`:

```
./gradlew -PmavenMirror=https://maven-central.storage-download.googleapis.com/maven2/
    ↪ publishToMavenLocal
```

The Kotlin package is `wfdbx` (`import wfdbx.*`); the C standard library emulation is in `wfdbx.cstd`, the tools in `wfdbx.apps`, and the command-line driver in `wfdbx.cli` (Table 12.8).

12.1.3 The command-line tools

The tools need the runnable jar `kotlin/build/libs/wfdbx-all.jar`, built in the directory `kotlin` by:

```
./gradlew fatJar
bin/wfdbx --version               # WFDB-x 1.2.3
bin/wfdbx --list                  # the tools
```

12.1.4 Licenses

The library is distributed under the GNU Lesser General Public License, version 2.1 or later, and the command-line tools (`wfdbx.apps`, `wfdbx.cli`) under the GNU General Public License, version 3 or later; the FLAC decoder, translated from libFLAC, also carries libFLAC's BSD-3-Clause notice. Every jar of the build (the library jars, the sources jars and `wfdbx-all.jar`) holds the files `LICENSE`, `LICENSE-GPL` and `NOTICE` of the repository in `META-INF` (`NOTICE` gives the licenses and the notice of libFLAC), and the POM of the published artifacts lists the three licenses.

12.1.5 The sample record

The repository includes the sample record `100s` in `tests/data`: the first minute of record 100 of the MIT-BIH Arrhythmia Database, two ECG signals (`MLII` and `V5`) sampled at 360 Hz, with the reference beat annotations (annotator `atr`). The library jar does not include it: an app copies the files it needs (`100s.hea`, `100s.dat`, `100s.atr`) to its own storage. Work that uses the MIT-BIH Arrhythmia Database cites it as described on PhysioNet.

12.1.6 Android and iOS

On Android and iOS, the current directory is not a useful place for files: the input and output directories are set to a directory of the app, such as `context.filesDir.path` on Android (Section 12.3.6). Reading URLs on Android needs the `INTERNET` permission; files are downloaded with `java.net.HttpURLConnection` on the JVM and Android, and with `NSURLSession` on iOS. Legacy-mode sorting of annotation files runs the `sortann` program in a new JVM (`java -cp . . .`), which an Android or iOS app cannot do (the sort of normal mode, done in the library, works everywhere).

12.2 Quick start

This example reads the header, ten seconds of the signals and the annotations of the sample record `100s`, and computes the mean heart rate. It then writes the first ten seconds as a new record, `first10`, with a metadata entry in its header. It assumes that the input and output directories are a copy of `tests/data` (`wfdbxSetInputDir("tests/data")`, as in `kotlin/README.md`, reads the sample record from the repository).

```
import wfdbx.*

// The header only: no samples are read.
val hdr = readHeader("100s")
println("${hdr.nsig} ${hdr.fs} ${hdr.nsamples}")       // 2 360.0 21600

// Seconds 10 to 20 of both signals, in ADC and physical units, frame by frame.
val rec = readRecord("100s", from = TimeSpec("0:10"), to = TimeSpec("0:20"), physical = true)
println("${rec.names} ${rec.nsamples}")                 // [MLII, V5] 3600
println("${rec.physical!![0]} ${rec.physicalAt(0, 1)}")    // -0.39 -0.275

// The beat annotations, and the mean heart rate.
val ann = readAnnotations("100s", "atr")
println("${ann.size} ${ann.symbols.take(3)} ${ann.samples.take(3)}")   // 75 [+, N, N] [18, 77,
    ↪ 370]
val beats = (0 until ann.size).filter { isqrs(ann.codes[it]) != 0 }.map { ann.samples[it] }
val rr = (1 until beats.size).map { (beats[it] - beats[it - 1]) / ann.fs }
println("${beats.size} beats, ${"%.1f".format(60 / rr.average())} beats per minute")   // 74 beats
    ↪ , 73.9 ...

// Write the first 10 seconds, in ADC units, as the record "first10".
val x = readRecord("100s", to = TimeSpec("0:10"))
writeRecord("first10", x.samples, x.nsig, x.fs, fmt = 16, gains = x.gains.toDoubleArray(),
            baselines = x.baselines.toIntArray(), adczeros = x.siginfo.map { it.adczero }.
    ↪ toIntArray(),
            adcres = x.siginfo.map { it.adcres }.toIntArray(), units = listOf("mV"), names = x.
    ↪ names,
            metadata = listOf("Source" to "MIT-BIH record 100"))

// Header metadata: "# Key: value" lines of the header.
setMetadata("first10", "Age", "69")
println(readMetadata("first10"))     // [(Source, MIT-BIH record 100), (Age, 69)]
println(checkRecord("first10"))      // no problems found
```

The header `100s.hea` gives no units: the example writes `"mV"` explicitly, since WFDB-x never assumes units, a gain, an ADC resolution or a sampling frequency that a record does not give (Chapter 4).

12.3 The high-level API

The high-level API is made of the functions of this section and of the classes of their results (Section 12.3.7), all in the package `wfdbx`. They are defined in the files `Record.kt`, `Check.kt` and `Metadata.kt` of `kotlin/src/commonMain/kotlin/wfdbx`, whose KDoc comments document each.

12.3.1 Conventions

The library state: `Wfdb` objects and the default instance. The state of the library (the open record, the annotators, the input and output directories, network access, legacy mode) is held by an object of the class `Wfdb`. Each high-level function is an extension function of `Wfdb`, such as `fun Wfdb.readRecord(...)`, called on an object: `w.readRecord("100s")`. Each is also a top-level function with the same name and the same parameters, such as `readRecord("100s")`, which runs it on the default instance of the calling thread, `wfdbDefault()` (one per thread; `withDefault { w -> ... }` runs a block with it). The entries of this section give the declarations of the extension functions; the top-level functions have the same parameters, defaults and results. Separate `Wfdb` objects share no state (except the process-wide list of open output files, the count of write errors and the cache of fetched URLs), so several records can be read at once, in one thread or in several; a `Wfdb` object is used by one thread at a time.

```
val w = Wfdb()                          // a library state of its own
w.wfdbxSetInputDir("tests/data")
val a = w.readRecord("100s")            // on w
val b = readRecord("100s", inputDir = "tests/data")   // on the default instance
```

Named arguments. The options are parameters with default values, given as named arguments (`physical = true`); there are no option structures. A `null` default means "not given" (for `inputDir` and `network`: the current setting).

Record names and directories. A record is named as in the other parts of WFDB-x (Chapters 2 and 3), without the `.hea` extension (the readers and the writers, `writeRecord` and `writeAnnotations`, also accept a trailing `.hea`, which is removed). A record name is a full path, absolute (`"/data/mitdb/100"`) or in the home directory (`"~/Documents/100s"`), used as given; a relative path (`"100s"`, `"data/100s"`), read from the input directory if one is set, otherwise from the current directory, and written to the output directory if one is set, otherwise to the current directory (missing directories are created); or a URL (`"https://physionet.org/files/mitdb/1.0.0/100"`), read only if network access is on, which cannot be written. A writer given a name with a path creates the files in that directory, and the header names the record and its signal file without the path (`rec`, `rec.dat`), so that the record opens under any path that leads to it; in legacy mode, as in the original, the header gives the name with its path. The input and output directories are set with `wfdbxSetInputDir` and `wfdbxSetOutputDir` (Section 12.3.6) or with the environment variables `WFDBX_INPUT_DIR` and `WFDBX_OUTPUT_DIR`. The functions that read take the parameters `inputDir` and `network`, and those that write take `dir`: they apply to that one call, and the previous settings are restored when it returns, also after an error. `inputDir` is optional, and applies to relative names only, not to full paths or URLs; with `dir`, the record name must be a plain name (`"rec"`), and the record opens from that directory under it. The signal files that a header names are read from the header's directory. Record, annotator and directory names are converted to bytes with the platform's file name encoding.

Times. The parameters `from` and `to` are `TimeSpec` values: a sample number (a `TimeSpec.Sample`, made from an `Int` or a `Long`), or a WFDB time string, as `strtim` reads it (a `TimeSpec.Text`):

```
TimeSpec(3600)            // sample 3600
TimeSpec("0:10")          // 10 seconds; also "1:02:03.5"
```

```
TimeSpec("s3600")         // sample 3600
TimeSpec("e")             // the end of the record
TimeSpec("[14:00:00]")    // a time of day
```

A time in seconds needs the sampling frequency; when the record does not give it, such a time throws a `WfdbException`.

Samples, frame by frame. The samples of a record are one array, frame by frame, as the library returns them: `samples[i * nsig + s]` is sample `i` of signal `s` (`rec.at(i, s)`), so that a record of millions of samples is one array, without a copy per signal. A signal with `spf` samples per frame has `spf` consecutive values in each frame.

Per-signal and per-annotation arguments. The arguments that describe each signal (`gains`, `units`, ...) take one value for all the signals, or exactly one value per signal. Those that describe each annotation (`subtypes`, `chans`, ...) have exactly one value per annotation.

Text. Text in the files (signal descriptions, units, info strings, metadata, aux strings, mnemonics) is a `String` with one character per byte (ISO-8859-1), so that every byte round-trips. The fields of `Siginfo` are byte strings (`ByteArray`), as in C; `latin1()`, from `wfdbx.cstd`, converts a `ByteArray` to a `String` and back.

Signal parameters. No sampling frequency, gain, ADC resolution or units is ever assumed. Where one is needed and the record does not give it, it must be given as an argument; otherwise the function throws a `WfdbException`. A record whose header gives no sampling frequency has `fs = 0`.

Legacy mode. With `legacy = true`, a function behaves as in the original WFDB package, with its assumed signal parameters and its search for files (Chapter 4); legacy mode is on, on that `Wfdb` object, for the duration of the call only. With `legacy = false` (the default), the current setting of `wfdbxSetLegacy` applies, which is off unless it was turned on.

State. The functions that read and write records and annotation files close anything opened with the WFDB library functions on the same `Wfdb` object (they call `wfdbquit()`) before they start, and close their own files before they return, whatever the outcome. They leave the time resolution of output annotators (`getafreq()`) and the getvec mode (`getgvmode()`) as they were, also after an error and after reading an EDF record, whose opening sets both, as in C (`wfdbquit` resets neither): `readRecord` sets the getvec mode for its own reading only, and the `fs` of `writeAnnotations` applies to that file only. The metadata functions open no signals and leave the library's state as it is.

The examples of this section use the top-level functions, with the input and output directories set to a copy of `tests/data` (`wfdbxSetInputDir(d)` and `wfdbxSetOutputDir(d)`), which holds the test records of WFDB-x, among them `100s`; they write records of their own there. The listings of style `example` were compiled and run with the library on the JVM, and print what their comments say.

12.3.2 Reading records

`readRecord` — Read the signals of a record

```
fun Wfdb.readRecord(
    record: String,
    channels: List<String>? = null,
    from: TimeSpec? = null,
```

```
    to: TimeSpec? = null,
    physical: Boolean = false,
    highres: Boolean = false,
    pad: Boolean = false,
    checksumWarning: Boolean = true,
    onWarning: ((String) -> Unit)? = null,
    inputDir: String? = null,
    network: Boolean? = null,
    legacy: Boolean = false,
): Record
```

Reads the signals of a record, or part of it, into one `IntArray` of ADC units, frame by frame, together with the description of the record. The samples are those that `getvec` returns (Chapter 13), read in large blocks. Invalid samples are `WFDB_INVALID_SAMPLE` (-32768) in ADC units, and NaN in physical units. When the read reaches the end of the record, the samples are compared with the checksums of the header (Section 12.3.8). On any error once the record is open, its input signals are closed, and the getvec mode and the settings changed for the call are restored.

record
: Record name, without the `.hea` extension (a trailing `".hea"` is also accepted): a full path, absolute (`"/data/mitdb/100"`) or in the home directory (`"~/Documents/100s"`), used as given; a relative path (`"100s"`, `"data/100s"`), read from the input directory if one is set, otherwise from the current directory; or a URL (`"https://physionet.org/files/mitdb/1.0.0/100"`), read only if network access is on.

channels
: The signals to read, in the order in which they are returned: a list of signal numbers (`"0"`, `"1"`, ...) or signal descriptions (as in `Record.names`, such as `"MLII"`), which may be mixed. A name is looked up among the descriptions first, then with `findsig`. Default `null`: all the signals, in the order of the header.

from
: The first sample to read: a `TimeSpec`, a sample number (≥ 0) or a time string. Default `null`: the beginning of the record.

to
: The sample after the last one to read: a `TimeSpec`. Default `null`: the end of the record.

physical
: `true` to return the samples in physical units too, in `Record.physical` (a `DoubleArray` in the layout of `samples`), computed as $(adc - baseline)/gain$. Every signal read must then have a gain. Default `false`.

highres
: `false` (the default) to read a multi-frequency record at its frame frequency, with the samples of the faster signals averaged to one per frame; `true` to read every sample, at the highest sampling frequency of the record (the samples of the slower signals are repeated). `Record.fs` is then that frequency.

pad
: `true` to replace each invalid sample by the previous valid sample of the same signal. Default `false`.

checksumWarning
: `true` (the default) to warn when the samples do not match the checksums of the header; `false` for no warning. The samples are returned either way, and `checksumOk` of the result is `false`.

onWarning
: A function that receives the warning (without the `WFDB-x:` prefix and the newline). Default

`null`: the library prints `WFDB-x:` followed by the warning on the standard error, unless `wfdbquiet()` was called.

`inputDir`
The input directory for this call only (optional): a local directory, a ~ path or a URL, under which relative record names are read. It does not apply to full paths or URLs. Default `null`: the current setting (`wfdbxSetInputDir`).

`network`
`true` to allow reading URLs for this call only, `false` to forbid it. Default `null`: the current setting (`wfdbxSetNetwork`).

`legacy`
`true` to behave as the original WFDB package (legacy mode, for this call only). Default `false`.

At most 2^{26} values are allocated in advance, whatever length the header gives; the array grows as the frames are read.

Returns A `Record` (Table 12.1).

Errors `WfdbException` if the header or a signal file cannot be read ("can't read record REC"), if a channel is not in the record ("record REC has no signal C"), if `from` is a negative sample number ("readRecord: from must not be negative"), if a time string needs the sampling frequency and the header gives none ("sampling frequency of record REC is not specified"), if `physical` is `true` and a signal read has no gain ("signal N of record REC is uncalibrated"), if `from` is beyond the end of the record ("can't seek to sample N of record REC"), if a signal file is shorter than its header says or cannot be read ("can't read signal data of record REC (getvec status -3)"), even if a checksum error is found in the same frame: an incomplete frame is never returned, or if the samples do not fit in one array ("record REC is too long to be read at once"). Warns on a checksum error alone (see `checksumWarning` and `onWarning`).

```
import wfdbx.*

// Seconds 10 to 20 of lead MLII, in physical units.
var rec = readRecord("100s", channels = listOf("MLII"), from = TimeSpec("0:10"),
                     to = TimeSpec("0:20"), physical = true)
println("${rec.nsamples} ${rec.nsig} ${rec.names} ${rec.fs} ${rec.sampfrom}")   // 3600 1 [MLII]
    ↪ 360.0 3600

// The first 3 samples of both signals, in ADC units, V5 first.
rec = readRecord("100s", channels = listOf("1", "0"), to = TimeSpec(3))
println(rec.samples.toList())       // [1011, 995, 1011, 995, 1011, 995]
println("${rec.names} ${rec.gains} ${rec.baselines} ${rec.units}")
// [V5, MLII] [200.0, 200.0] [1024, 1024] [null, null]

// The whole record: the checksums are verified.
rec = readRecord("100s")
println("${rec.nsamples} ${rec.checksumOk}")      // 21600 true
```

A multi-frequency record (here the test record `mfreq`, whose signals have 1, 2 and 4 samples per frame at a frame frequency of 100 Hz) is read at its frame frequency, or at its highest sampling frequency:

```
import wfdbx.*

var rec = readRecord("mfreq")
println("${rec.fs} ${rec.nsamples} ${rec.nsig}")     // 100.0 600 3
```

```
rec = readRecord("mfreq", highres = true)
println("${rec.fs} ${rec.nsamples} ${rec.nsig}")     // 400.0 2400 3
println(rec.siginfo.map { it.spf })                  // [1, 2, 4]
```

readHeader

Read the header of a record

```
fun Wfdb.readHeader(
    record: String,
    inputDir: String? = null,
    network: Boolean? = null,
    legacy: Boolean = false,
    onWarning: ((String) -> Unit)? = null,
): Header
```

Reads the header of a record, without opening its signal files. The result describes the record and each of its signals, and holds the info strings and the metadata entries of the header (of the `.hea` file only, not of a `.info` file; none for an EDF file or a header read from the standard input). For a multi-segment record, the master header is read, and `nsamples` is the length of the whole record.

record
: Record name, without the `.hea` extension (a trailing `".hea"` is also accepted): a full path, absolute (`"/data/mitdb/100"`) or in the home directory (`"~/Documents/100s"`), used as given; a relative path (`"100s"`, `"data/100s"`), read from the input directory if one is set, otherwise from the current directory; or a URL (`"https://physionet.org/files/mitdb/1.0.0/100"`), read only if network access is on.

inputDir
: The input directory for this call only (optional): a local directory, a ~ path or a URL, under which relative record names are read. It does not apply to full paths or URLs. Default `null`: the current setting (`wfdbxSetInputDir`).

network
: `true` to allow reading URLs for this call only, `false` to forbid it. Default `null`: the current setting.

legacy
: `true` to read the header as the original WFDB package does. Default `false`.

onWarning
: A function that receives the warnings about `Header.metadata` (a key given more than once, Section 12.3.8). Default `null`: the library prints `WFDB-x:` followed by the warning on the standard error, unless `wfdbquiet()` was called.

Returns A `Header` (Table 12.2).

Errors `WfdbException` if the header cannot be found or read ("can't read header for record REC").

```
import wfdbx.*
import wfdbx.cstd.latin1

val hdr = readHeader("100s")
println("${hdr.name} ${hdr.fs} ${hdr.nsig} ${hdr.nsamples}")    // 100s 360.0 2 21600
for (s in hdr.siginfo) {
```

```
    println("${s.desc?.latin1()} ${s.fname?.latin1()} ${s.fmt} ${s.gain} ${s.adcres} ${s.units?.
    ↪ latin1()}")
}
// MLII 100s.dat 212 200.0 11 null
// V5 100s.dat 212 200.0 11 null
println("\"${hdr.comments[0]}\"")   // " 69 M 1085 1629 x1": the first info string
println(hdr.baseTime)               // null: the header gives no start time
```

---

readAnnotations

Read an annotation file

```
fun Wfdb.readAnnotations(
    record: String,
    annotator: String,
    from: TimeSpec? = null,
    to: TimeSpec? = null,
    fs: Double? = null,
    inputDir: String? = null,
    network: Boolean? = null,
    legacy: Boolean = false,
): Annotations
```

Reads the annotations of an annotation file, in file order. The sampling frequency, needed to convert times in seconds, is that of the record's header, or, for a record without a header or whose header gives none, the argument `fs`.

record
: Record name, without the `.hea` extension (a trailing `".hea"` is also accepted): a full path, absolute (`"/data/mitdb/100"`) or in the home directory (`"~/Documents/100s"`), used as given; a relative path (`"100s"`, `"data/100s"`), read from the input directory if one is set, otherwise from the current directory; or a URL (`"https://physionet.org/files/mitdb/1.0.0/100"`), read only if network access is on. Its header, if there is one, gives the sampling frequency.

annotator
: Annotator name: the suffix of the annotation file. `"atr"` reads the file `RECORD.atr`.

from
: The first time to read: a `TimeSpec`. Default `null`: the beginning of the file.

to
: The time at which to stop: annotations at or after it are not read. Default `null`: the end of the file.

fs
: The sampling frequency, in Hz (> 0), to use when the record has no header or its header gives none; it is ignored otherwise. Default `null`: none.

inputDir
: The input directory for this call only (optional): a local directory, a ~ path or a URL, under which relative record names are read. It does not apply to full paths or URLs. Default `null`: the current setting (`wfdbxSetInputDir`).

network
: `true` to allow reading URLs for this call only, `false` to forbid it. Default `null`: the current setting.

legacy
: `true` to behave as the original WFDB package (which assumes 250 Hz for a record without a frequency). Default `false`.

Returns An `Annotations` object (Table 12.3). Its `fs` is the frequency in which the times are counted: the one written in the annotation file, if there is one, or else the sampling frequency; it is 0 if none of them is known.

Errors `WfdbArgumentException`, before any file is opened, if `from` is a negative sample number ("readAnnotations: from must not be negative"), or if `fs` is given and is not > 0 ("readAnnotations: fs must be > 0"); `WfdbException` if the annotation file cannot be opened ("can't read annotator ANN for record REC"), if a time string needs the sampling frequency and none is known, or if the file ends before its end-of-file marker ("can't read annotation file ANN of record REC (getann status -3)"). Whatever happens, the annotation file is closed (`wfdbquit`) before the function returns.

```
import wfdbx.*

var ann = readAnnotations("100s", "atr", from = TimeSpec("0:10"), to = TimeSpec("0:30"))
println("${ann.size} ${ann.fs}")                              // 24 360.0
println("${ann.samples.take(3)} ${ann.symbols.take(3)}")      // [3862, 4170, 4466] [N, N, N]

// The whole file: the first annotation is a rhythm change, with an aux string.
ann = readAnnotations("100s", "atr")
println("${ann.symbols[0]} ${ann.codes[0] == RHYTHM} ${ann.aux[0]}")   // + true (N
println(ann.symbols.toSortedSet())                            // [+, A, N]
```

For a record without a header, `fs` gives the frequency in which times in seconds are converted (the library reports on the standard error that it cannot open the header):

```
import wfdbx.*

writeAnnotations("beats", "qrs", longArrayOf(100, 460, 820), symbols = listOf("N", "N", "N"))
val ann = readAnnotations("beats", "qrs", to = TimeSpec("0:02"), fs = 360.0)
println("${ann.samples.toList()} ${ann.fs}")        // [100, 460] 360.0
```

12.3.3 Writing records

`writeRecord` — Write a header and a signal file

```
fun Wfdb.writeRecord(
    record: String,
    samples: IntArray,
    nsig: Int,
    fs: Double,
    fmt: Int = 16,
    gains: DoubleArray? = null,
    baselines: IntArray? = null,
    adczeros: IntArray? = null,
    adcres: IntArray? = null,
    units: List<String?>? = null,
    names: List<String>? = null,
    baseTime: String? = null,
    comments: List<String>? = null,
    sampsPerFrame: IntArray? = null,
    dir: String? = null,
    legacy: Boolean = false,
    metadata: List<Pair<String, String?>>? = null,
```

```
    invalidWarning: Boolean = true,
    onWarning: ((String) -> Unit)? = null,
)

fun Wfdb.writeRecord(
    record: String,
    samples: DoubleArray,
    nsig: Int,
    fs: Double,
    fmt: Int = 16,
    gains: DoubleArray? = null,
    baselines: IntArray? = null,
    adczeros: IntArray? = null,
    adcres: IntArray? = null,
    units: List<String?>? = null,
    names: List<String>? = null,
    baseTime: String? = null,
    comments: List<String>? = null,
    sampsPerFrame: IntArray? = null,
    dir: String? = null,
    legacy: Boolean = false,
    metadata: List<Pair<String, String?>>? = null,
    invalidWarning: Boolean = true,
    onWarning: ((String) -> Unit)? = null,
)
```

Writes samples as a new record: the header `RECORD.hea` and one signal file, `RECORD.dat`, which holds all the signals. Existing files of these names are replaced: both files are written to temporary files in the destination directory (`.NAME.XXXXXXXX.tmp`, readable and writable by their owner only), which replace the existing files, keeping their permissions, only when both are complete, so that an error leaves the existing files as they were and no temporary file. A symbolic link is kept (the file it leads to is replaced), and a destination that is not a regular file (a device, a FIFO) is written directly. The first overload takes the samples in ADC units (an `IntArray`), the second in physical units (a `DoubleArray`), converted to ADC units as $\mathrm{round}(x \cdot gain) + baseline$, with halves rounded away from zero; NaN becomes an invalid sample, and an infinite value, or one whose ADC value is beyond the range of 32-bit integers, is an error. The sampling frequency, the gain and ADC resolution of each signal, and the units of each calibrated signal must be given: there are no defaults. The per-signal arguments (`gains`, `baselines`, `adczeros`, `adcres`, `units`, `names`, `sampsPerFrame`) take one value for all the signals, or one value per signal. The record name, the signal names, the units and the comments are text of one byte per character (ISO-8859-1). All the arguments are checked before any file is created or changed, and the output signals are closed whatever happens.

record
: Record name (not empty), without the `.hea` extension (a trailing `".hea"` is removed, as by the readers): a plain name (`"rec"`) or a relative path (`"data/rec"`), created in the output directory if one is set, otherwise in the current directory, missing directories included; or a full path, absolute or in the home directory (`"~/Documents/rec"`), used as given. With a path, the files are created in that directory, and the header names the record and its signal file without the path (`rec`, `rec.dat`), so that the record opens under any path that leads to it (in legacy mode, as in the original, the header gives the name with its path). A URL cannot be written.

samples
: The samples, frame by frame: signal 0, signal 1, … of frame 0, then those of frame 1, and

so on (a signal with `spf` samples per frame has `spf` consecutive values in each frame). An `IntArray` of ADC units, or a `DoubleArray` of physical units. Its size must be a multiple of the frame size.

`nsig`
The number of signals (at least 1).

`fs` The sampling frequency, in Hz (frames per second): a finite number > 0.

`fmt` The storage format of all the signals, such as `16`, `212`, `24` or `516` (FLAC) (Chapter 15). Every sample must fit the format: within its range (Table 15.1; -2048 to 2047 for format 212, for example), and, in format 8, within -128 to 127 of the sample before it in the same signal. The smallest value of a format is the value of an invalid (missing) sample: a sample equal to it, or to `WFDB_INVALID_SAMPLE`, is read back as invalid (see `invalidWarning`). A FLAC file holds at most 8 signals. Default `16`.

`gains`
The gain of each signal, in ADC units per physical unit: a finite number; `0` marks an uncalibrated signal, which is allowed only for samples given in ADC units. Negative gains are allowed. Required (default `null`: not given).

`baselines`
The ADC value that corresponds to 0 physical units. Default `null`: the ADC zero of the signal.

`adczeros`
The ADC value at the middle of the ADC range. Default `null`: 0.

`adcres`
The ADC resolution, in bits (≥ 1). Required (default `null`: not given).

`units`
The physical units, such as `"mV"`; required for each signal whose gain is not 0 (`null` for an uncalibrated signal). Default `null`: not given.

`names`
The signal descriptions, such as `"MLII"`. Default `null`: no descriptions.

`baseTime`
The start time, with an optional date: `"hh:mm:ss"` or `"hh:mm:ss dd/mm/yyyy"`. Default `null`: no start time.

`comments`
Info strings, written at the end of the header as comment lines (`"#"` followed by the string). Default `null`: none.

`sampsPerFrame`
For a multi-frequency record, the number of samples per frame of each signal (integers ≥ 1); `fs` is then the frame frequency. Default `null`: one sample per frame for every signal.

`dir` A directory (which must exist) in which to create the files, for this call only. `record` must then be a plain name, without a directory, and the record opens from that directory under its plain name. Default `null` (or `""`): the output directory.

`legacy`
`true` to behave as the original WFDB package: the gain defaults to 200, the ADC resolution to one based on the format, and no units are required. Default `false`.

`metadata`
Metadata entries (Chapter 5) to write to the header after the comments, as `# Key: value` (through `putinfo`), in the order given: a list of pairs, `listOf("Age" to "63")` (a map converts with `toList()`). Keys and values follow the rules of `setMetadata`; no two keys may be equal

(ignoring letter case), none may be the key of an entry in `comments`, and a `null` value is not allowed. Default `null`: none.

`invalidWarning`
: `true` to give a warning when samples will be read back as invalid (missing) samples ("N sample(s) of signal K will be read back as invalid (missing) samples (format F stores an invalid sample as V)"): those equal to the invalid value of the format or to `WFDB_INVALID_SAMPLE`, other than the NaN samples of physical units, which mark missing samples deliberately. `false` for no warning, when such samples are deliberate. Default `true`.

`onWarning`
: Receives the warnings (without the `"WFDB-x: "` prefix and the newline). Default `null`: the library prints `"WFDB-x: "` and the warning on the standard error, unless `wfdbquiet` was called, as for the checksum warning of `readRecord`.

Returns Nothing (`Unit`).

Errors `WfdbArgumentException`, before any file is created or changed, in this order: "writeRecord: sampsPerFrame must hold integers >= 1" (checked first, also in legacy mode), "writeRecord: the record name must not be empty", "writeRecord: record name 'X' has a character that is not Latin-1 (one byte per character)", "writeRecord: at least one signal is required", "writeRecord: NAME must have one value per signal"; in normal mode "writeRecord: gains must be specified", "writeRecord: ADC resolution(s) must be specified" (missing, or < 1), "writeRecord: units must be specified"; "writeRecord: the sampling frequency must be > 0" (not checked in legacy mode), "writeRecord: the sampling frequency must be finite", "writeRecord: gains must be finite", "writeRecord: signal name 'X' has a character that is not Latin-1 (one byte per character)", "writeRecord: units 'X' has a character ...", "writeRecord: signal N is uncalibrated" and "writeRecord: sample I is infinite or beyond the range of 32-bit ADC units" (physical units), "writeRecord: the number of samples is not a multiple of the frame size", a sample that does not fit the format ("writeRecord: format 516 (FLAC) stores at most 8 signals in a file; the record has 12"; "writeRecord: signal K has an invalid sample, which format 8 cannot store", "writeRecord: signal K changes by D between two samples, more than format 8 can store (-128 to 127)"; "writeRecord: signal K has a sample (V) outside the range of format F (LO to HI)"), "writeRecord: comment 'X' has a character ...", an entry of `metadata` that is not valid ("writeRecord: metadata key 'a:b' must not contain ':'", "writeRecord: value of metadata key 'Age' must not be null", "writeRecord: metadata key 'Age' is also given in comments", ...), "writeRecord: incorrect base time 'TIME'". `WfdbException`: "record name must not contain a directory when dir is given", "directory DIR does not exist"; if the signal file or the header cannot be created ("can't create signal file for record REC", "can't create header for record REC"), or cannot be written completely, for example on a full disk, or cannot replace the existing file ("write error on record REC").

```
import wfdbx.*
import kotlin.math.abs

// Two signals of random numbers, in millivolts, at 360 Hz: the record "rand"
// (rand.hea and rand.dat).
val rnd = kotlin.random.Random(1)
val x = DoubleArray(3600 * 2) { rnd.nextDouble(-3.0, 3.0) }
writeRecord("rand", x, nsig = 2, fs = 360.0, fmt = 16, gains = doubleArrayOf(1000.0),
            adcres = intArrayOf(16), units = listOf("mV"), names = listOf("A", "B"),
            baseTime = "14:00:03 21/06/2024", comments = listOf("random numbers"),
            metadata = listOf("Generator" to "kotlin.random"))
val rec = readRecord("rand", physical = true)
println("${rec.nsamples} ${rec.baseTime}")     // 3600 [14:00:03 21/06/2024]
```

```
println(x.indices.all { abs(rec.physical!![it] - x[it]) <= 0.0005 })   // true: rounded to 1/1000
    ↪ mV
println(readHeader("rand").comments)           // [random numbers,  Generator: kotlin.random]

// An uncalibrated signal (gain 0, no units), in ADC units, in the directory "out".
writeRecord("out/counts", IntArray(1000) { it }, nsig = 1, fs = 100.0, gains = doubleArrayOf(0.0),
            adcres = intArrayOf(12))
println(readRecord("out/counts").samples.take(3))    // [0, 1, 2]
```

A multi-frequency record: signal `fast` has 2 samples per frame, signal `slow` one; each frame holds the two samples of `fast` and then the sample of `slow`:

```
import wfdbx.*

val frames = 100                                    // 1 second at 100 frames per second
val x = IntArray(3 * frames) { if (it % 3 == 2) 500 else 2 * (it / 3) + it % 3 }
writeRecord("mf", x, nsig = 2, fs = 100.0, gains = doubleArrayOf(200.0), adcres = intArrayOf(12),
            units = listOf("mV"), names = listOf("fast", "slow"), sampsPerFrame = intArrayOf(2, 1)
    ↪ )
val rec = readRecord("mf", highres = true)
println("${rec.fs} ${rec.samples.take(6)}")        // 200.0 [0, 500, 1, 500, 2, 500]
```

---

writeAnnotations

Write an annotation file

```
fun Wfdb.writeAnnotations(
    record: String,
    annotator: String,
    samples: LongArray,
    symbols: List<String>? = null,
    codes: IntArray? = null,
    subtypes: IntArray? = null,
    chans: IntArray? = null,
    nums: IntArray? = null,
    aux: List<String?>? = null,
    fs: Double? = null,
    sort: Boolean? = null,
    dir: String? = null,
    legacy: Boolean = false,
)

fun Wfdb.writeAnnotations(
    record: String,
    annotator: String,
    a: Annotations,
    fs: Double? = null,
    sort: Boolean? = null,
    dir: String? = null,
    legacy: Boolean = false,
)
```

Writes an annotation file, `RECORD.ANNOTATOR`, in the MIT format; an existing file of that name is replaced: the new file is written to a temporary file in the same directory (readable and writable by its owner only), which replaces it, keeping its permissions, only when it is complete (and, if needed, sorted), so that an error leaves it as it was. The record need not have a header. The

annotation types are given as `symbols` or as `codes` (Chapter 15). If the annotations are not in time order, the file is sorted when it is written, unless `sort = false`. The second overload writes the annotations of an `Annotations` object, as `readAnnotations` returns them: its samples, codes, subtypes, channels, annotator numbers and aux strings (its record, annotator and `fs` are not used). All the arguments are checked before the file is created or changed.

`record`
: Record name, without the `.hea` extension (a trailing `".hea"` is also accepted): a plain name or a relative path (`"data/100s"`) is created in the output directory if one is set, otherwise in the current directory; a full path, absolute or in the home directory (`"~/Documents/100s"`), is used as given. The file is `RECORD.ANNOTATOR` at that place (`data/100s.qrs`). A URL cannot be written.

`annotator`
: Annotator name: the suffix of the file, made of letters, digits, _, - and ~.

`samples`
: The annotation times, in samples, one per annotation: from $-2^{63}+1$ to $2^{63}-2$ (the smallest and largest 64-bit integers are reserved by the file format).

`symbols`
: The mnemonics of the annotation types, such as `"N"` or `"V"`, one per annotation. An unknown mnemonic becomes code 0 (`NOTQRS`). Default `null`: the types are given by `codes`.

`codes`
: The annotation codes (`NORMAL`, `PVC`, ...), from 0 to 58 (59 to 63 are the control codes of the file format), one per annotation, instead of `symbols`. If both are given (as `readAnnotations` returns them), each symbol must be that of its code (an empty symbol is not checked). Default `null`.

`subtypes`
: The subtype of each annotation, stored as a signed byte (other values wrap around to $-128 \ldots 127$, as in C). Default `null`: 0.

`chans`
: The channel number of each annotation, stored as an unsigned byte (other values wrap around to $0 \ldots 255$). Default `null`: 0.

`nums`
: The annotator number of each annotation, stored as a signed byte. Default `null`: 0.

`aux` The auxiliary text of each annotation (ISO-8859-1 text), with `null` where there is none. Texts longer than 255 bytes are cut. Default `null`: none.

`fs` If given (a finite number > 0), the frequency in which the times are counted, written to the file. Default `null`: none is written.

`sort`
: `true` to always sort the file, `false` to keep the given order (without the library's advice to run `sortann`), for this call only. Default `null`: the environment variable `WFDBANNSORT` decides (sorting is on unless it is `0`).

`dir` A directory (which must exist) in which to create the file, for this call only. `record` must then be a plain name, without a directory: the file is `DIR/RECORD.ANNOTATOR`. Default `null`: the output directory.

`legacy`
: `true` to behave as the original WFDB package. Default `false`.

`a` The annotations to write (second overload): an `Annotations` object, as `readAnnotations` returns it.

Returns Nothing (`Unit`).

Errors `WfdbArgumentException`, before the file is created or changed, in this order: if an array does not have one value per annotation ("writeAnnotations: NAME must have one value per annotation"), if `fs` is not finite and > 0 ("writeAnnotations: fs must be finite and > 0"), if a time is -2^{63} or $2^{63} - 1$ ("writeAnnotations: annotation time T cannot be written (annotation times must be integers from -9223372036854775807 to 9223372036854775806)"), if neither `symbols` nor `codes` is given for one annotation or more ("symbols or codes must be given"; with no annotations, an empty file is written), if a code is not from 0 to 58 ("writeAnnotations: annotation code C is reserved (59 to 63 are the control codes of the annotation file format)", or "writeAnnotations: annotation code C is out of range (0 to 58)"), if both are given and disagree ("writeAnnotations: symbols and codes disagree at annotation K", K counting from 0), if an aux text has a character beyond U+00FF ("writeAnnotations: aux text 'X' has a character that is not Latin-1 (one byte per character)"); `WfdbException` as for the `dir` of `writeRecord`, if the file cannot be created ("can't create annotator ANN for record REC"), or cannot be written completely, or the library reports an error while writing it ("write error on annotator ANN for record REC").

```
import wfdbx.*

// Three beats and a rhythm change, by symbol, with an aux string for the rhythm.
writeAnnotations("100s", "test", longArrayOf(18, 77, 370, 662),
                 symbols = listOf("+", "N", "V", "N"), aux = listOf("(N", null, null, null))
var ann = readAnnotations("100s", "test")
println("${ann.symbols} ${ann.aux}")       // [+, N, V, N] [(N, null, null, null]

// A copy of the reference annotations, as readAnnotations returns them.
val a = readAnnotations("100s", "atr")
writeAnnotations("100s", "copy", a)
val b = readAnnotations("100s", "copy")
println("${b.samples.contentEquals(a.samples)} ${b.symbols == a.symbols} ${b.aux == a.aux}")
// true true true

// Annotations out of time order are sorted when the file is written (the library
// reports it on the standard error).
writeAnnotations("100s", "unsorted", longArrayOf(500, 100), codes = intArrayOf(NORMAL, NORMAL))
println(readAnnotations("100s", "unsorted").samples.toList())     // [100, 500]
```

12.3.4 Checking records

checkRecord Check a record and its annotation files

```
fun Wfdb.checkRecord(
    record: String,
    annotators: List<String> = emptyList(),
    inputDir: String? = null,
    network: Boolean? = null,
): CheckReport
```

Checks the header of a record, its signal specifications, every sample of its signal files, each segment of a multi-segment record, and the annotation files named, and reports each problem found. A problem in the record never throws an exception: it is reported, with a severity (`Severity.ERROR`: the record is damaged or does not follow the WFDB format, and programs will read wrong data or fail; `Severity.WARNING`: the record can be read, but something is missing

or unusual, and may give wrong results; `Severity.INFO`: a fact worth knowing, such as invalid samples). The problems, their codes and their messages are the same in the seven languages (Table 6.7). The checks run in normal mode, even if legacy mode is on, and the library prints no messages during the call. They are similar to those of Benjamin Moody's `wfdbcheck`.

`record`
: Record name, without the `.hea` extension (a trailing `".hea"` is also accepted): a full path, absolute (`"/data/mitdb/100"`) or in the home directory (`"~/Documents/100s"`), used as given; a relative path (`"100s"`, `"data/100s"`), read from the input directory if one is set, otherwise from the current directory; or a URL (`"https://physionet.org/files/mitdb/1.0.0/100"`), read only if network access is on (as for `readRecord`).

`annotators`
: The annotator names whose files are checked, such as `listOf("atr")`. Default `emptyList()`: none.

`inputDir`
: The input directory for this call only (optional): a local directory, a ~ path or a URL, under which relative record names are read. It does not apply to full paths or URLs. Default `null`: the current setting (`wfdbxSetInputDir`).

`network`
: `true` to allow reading URLs for this call only, `false` to forbid it. Default `null`: the current setting.

Returns A `CheckReport` (Table 12.4): `report.ok` is `true` if no problem is an error, `report.issues` lists the problems (`Issue` objects, Table 12.5), and `println(report)` prints one line per problem, or `no problems found`.

Errors `WfdbException` if `record` is empty ("record must not be empty"). A record that cannot be read is reported (code `header_unreadable`), not thrown.

```
import wfdbx.*

var report = checkRecord("100s", listOf("atr"))
println(report.ok)          // true: no errors
println(report)
// warning: signal 0: the units are not specified
// warning: signal 1: the units are not specified

// A damaged record: one sample of a 12-bit signal is overwritten with 2048 (JVM file
// access, in the output directory).
writeRecord("damaged", IntArray(100) { it }, nsig = 1, fs = 100.0, gains = doubleArrayOf(100.0),
            adcres = intArrayOf(12), units = listOf("mV"), names = listOf("ramp"))
java.io.RandomAccessFile(wfdbxGetOutputDir() + "/damaged.dat", "rw").use { f ->
    f.seek(20)                                  // sample 10 (2 bytes per sample in format 16)
    f.write(byteArrayOf(0x00, 0x08))            // 0x0800 = 2048, little-endian
}
report = checkRecord("damaged")
println(report.ok)          // false
for (issue in report.issues) println("${issue.severity} ${issue.code} ${issue.where}")
// error sample_range signal 0
// error checksum signal 0
println(report.lines()[0])
// error: signal 0: 1 sample is outside the ADC range (-2048 to 2047): sample 10 (value 2048)
```

12.3.5 Header metadata

These functions read and change the metadata of a header: its info strings of the form `# Key: value`, such as `# Age: 63` (Chapter 5). In short: keys are compared without regard to letter case (A–Z and a–z only); a key must not be empty and must not contain `:`, `#`, a line break or a NUL character, and a value must not contain a line break or a NUL character; spaces, tabs, VT, FF and CR at either end of keys and values are removed; text is one character per byte (ISO-8859-1), and a character beyond U+00FF is an argument error. The info strings are the comment lines after the record line, which is found as the header parser finds it, in every mode: the first line that holds something other than spaces, tabs, CRs and LFs and does not begin (after them) with `#`; comments and blank lines before it are never read or changed. A change rewrites the header in one step: the new header is written to a temporary file in the header's directory, readable and writable by its owner only until it is complete (a header may hold patient data), which then replaces the header, keeping its permission bits where the platform has them; every other byte of the header is kept, and all the arguments are checked first, so an error leaves the header unchanged. If nothing changes (and `dir` is not given), nothing is written. Only the `.hea` file is read (not a `.info` file); for a multi-segment record, it is the master header, unless a segment is named. Invalid arguments throw a `WfdbArgumentException`, and a header that cannot be read or written a `WfdbException` (Section 12.3.8). Metadata often holds information about patients: WFDB-x stores it as plain text, exactly as given, and does not de-identify it.

> **Patient data** A header can hold identifying data about a patient (a name, a date of birth, a medical record number). The metadata functions read and write such data as given. Records that are shared must be de-identified first.

`readMetadata` — Read the metadata of a record

```
fun Wfdb.readMetadata(
    record: String,
    onWarning: ((String) -> Unit)? = null,
    inputDir: String? = null,
    network: Boolean? = null,
): List<Pair<String, String>>
```

Reads the metadata entries of the header of a record, in file order. An info string is an entry if it contains a colon: the key is the text before the first colon, the value the rest of the line, both without spaces at either end. Info strings without a colon are free comments, which are not returned.

`record`
: Record name, without the `.hea` extension (a trailing `".hea"` is also accepted): a full path, absolute (`"/data/mitdb/100"`) or in the home directory (`"~/Documents/100s"`), used as given; a relative path (`"100s"`, `"data/100s"`), read from the input directory if one is set, otherwise from the current directory; or a URL (`"https://physionet.org/files/mitdb/1.0.0/100"`), read only if network access is on.

`onWarning`
: A function that receives the warnings (without the `WFDB-x:` prefix and the newline). Default `null`: the library prints `WFDB-x:` followed by the warning on the standard error, unless `wfdbquiet()` was called.

`inputDir`
: The input directory for this call only (optional): a local directory, a ~ path or a URL, under

which relative record names are read. It does not apply to full paths or URLs. Default `null`: the current setting (`wfdbxSetInputDir`).

network
`true` to allow reading URLs for this call only, `false` to forbid it. Default `null`: the current setting.

Returns A list of (key, value) pairs, in file order. A key that appears more than once (ignoring letter case) is returned once, with its first value, and a warning is issued (Section 12.3.8).

Errors `WfdbException` if the header cannot be found or read ("can't read header for record REC"); `WfdbArgumentException` if `record` is empty or `"-"` (the standard input).

```
import wfdbx.*

writeRecord("pat", IntArray(360), nsig = 1, fs = 360.0, gains = doubleArrayOf(200.0),
            adcres = intArrayOf(12), units = listOf("mV"), names = listOf("ECG"),
            comments = listOf("recorded at rest"), metadata = listOf("Age" to "63", "Sex" to "F"))
println(readMetadata("pat"))           // [(Age, 63), (Sex, F)]
println(readHeader("pat").comments)
// [recorded at rest,  Age: 63,  Sex: F]: the free comment is not an entry
```

---

getMetadata

Return the value of a metadata entry

```
fun Wfdb.getMetadata(
    record: String,
    key: String,
    onWarning: ((String) -> Unit)? = null,
    inputDir: String? = null,
    network: Boolean? = null,
): String?
```

Returns the value of one metadata entry of the header of a record. The key is checked before the header is read.

record
Record name (as for `readMetadata`).

key The key, such as `"Age"`. Letter case is ignored, and spaces at either end are removed.

onWarning
A function that receives the warnings (as for `readMetadata`). Default `null`: the standard error.

inputDir
The input directory for this call only. Default `null`: the current setting.

network
`true` to allow reading URLs for this call only, `false` to forbid it. Default `null`: the current setting.

Returns The value (`""` for an entry with an empty value), or `null` if the header has no entry for `key`. If it has more than one, the first is returned, with a warning.

Errors `WfdbException` if the header cannot be found or read; `WfdbArgumentException` if `record` is empty or `"-"`, or if `key` is not a valid key (see `setMetadata`).

```
import wfdbx.*
```

```
writeRecord("pat", IntArray(360), nsig = 1, fs = 360.0, gains = doubleArrayOf(200.0),
            adcres = intArrayOf(12), units = listOf("mV"), metadata = listOf("Age" to "63", "Sex"
    ↪ to "F"))
println(getMetadata("pat", "age"))       // 63
println(getMetadata("pat", "Height"))    // null
```

---

setMetadata

Set the value of a metadata entry

```
fun Wfdb.setMetadata(
    record: String,
    key: String,
    value: String,
    inputDir: String? = null,
    network: Boolean? = null,
    dir: String? = null,
)
```

Sets the value of a metadata entry in the header of a record. If the header has an entry for `key` (ignoring letter case), its line is replaced by `# Key: value`, at the same position and with the same line ending, with the spelling of `key` given here (even if the value is the same); otherwise the entry is added after the last info string of the header (or, if it has none, after its last line that is not blank), ending as the record line ends. The entry is written as `# Key: value`, or `# Key:` if the value is empty. Every other byte of the header is kept.

record
: Record name (as for `readMetadata`). The header is changed where it was read, unless `dir` is given; a header read from a URL can only be written elsewhere, with `dir`.

key
: The key, such as `"Age"` or `"Date of birth"`. Spaces at either end are removed. It must not be empty, and must not contain `:`, `#`, a line break or a NUL character.

value
: The value, which may be a sentence. Spaces at either end are removed. It must not contain a line break or a NUL character; it may be empty.

inputDir
: The input directory in which the header is read, for this call only. Default `null`: the current setting.

network
: `true` to allow reading the header from a URL, for this call only, `false` to forbid it. Default `null`: the current setting.

dir
: A directory, which must exist, in which to write the changed header, under the same file name, instead of changing the header itself (the header is then written even if nothing changes); the signal files are not copied. Default `null`: the header itself is changed.

Returns Nothing (`Unit`).

Errors `WfdbArgumentException` if `key` or `value` is not valid ("metadata key 'a:b' must not contain ':'", "value of metadata key 'Age' must not contain a line break", ...), if the header has more than one entry for `key` ("header of record REC has more than one entry for key KEY (lines L1, L2); remove them first"), if it has no record line, if it was read from a URL and `dir` is not given, or if `record` is empty or `"-"`; `WfdbException` if the header cannot be read ("can't read header for record REC"), if `dir` does not exist ("directory DIR does not exist"), or if the changed header cannot be written ("can't write header for record REC").

```
import wfdbx.*

writeRecord("pat", IntArray(360), nsig = 1, fs = 360.0, gains = doubleArrayOf(200.0),
            adcres = intArrayOf(12), units = listOf("mV"), metadata = listOf("Age" to "63", "Sex"
    ↪ to "F"))
setMetadata("pat", "AGE", "64")                                  // replaces "# Age: 63"
setMetadata("pat", "Diagnosis", "sinus rhythm, no ectopy")       // a new entry
println(readMetadata("pat"))
// [(AGE, 64), (Sex, F), (Diagnosis, sinus rhythm, no ectopy)]

// Write the changed header to another directory; pat.hea is left unchanged.
val changed = wfdbxGetOutputDir() + "/changed"
java.io.File(changed).mkdir()                                    // (JVM)
setMetadata("pat", "Sex", "M", dir = changed)
println("${getMetadata("pat", "Sex")} ${getMetadata("$changed/pat", "Sex")}")    // F M
```

---

removeMetadata

Remove a metadata entry

```
fun Wfdb.removeMetadata(
    record: String,
    key: String,
    inputDir: String? = null,
    network: Boolean? = null,
    dir: String? = null,
): Int
```

Removes every entry for `key` (ignoring letter case) from the header of a record, with its line ending, so it also repairs a header that has more than one. Other lines are kept, byte for byte. Nothing is written if the header has no entry for `key`, unless `dir` is given.

record
: Record name (as for `setMetadata`).

key
: The key (as for `setMetadata`).

inputDir
: The input directory in which the header is read, for this call only. Default `null`: the current setting.

network
: `true` to allow reading the header from a URL, for this call only, `false` to forbid it. Default `null`: the current setting.

dir
: A directory, which must exist, in which to write the changed header instead of changing the header itself. Default `null`: the header itself is changed.

Returns The number of lines removed (0 if the header has no entry for `key`).

Errors As `setMetadata`, except that more than one entry for `key` is not an error.

```
import wfdbx.*

writeRecord("pat", IntArray(360), nsig = 1, fs = 360.0, gains = doubleArrayOf(200.0),
            adcres = intArrayOf(12), units = listOf("mV"), metadata = listOf("Age" to "63", "Sex"
    ↪ to "F"))
println(removeMetadata("pat", "sex"))     // 1
println(removeMetadata("pat", "Sex"))     // 0: there is no such entry now
```

```
println(readMetadata("pat"))              // [(Age, 63)]
```

updateMetadata

Make several changes to the metadata

```
fun Wfdb.updateMetadata(
    record: String,
    changes: List<Pair<String, String?>>,
    inputDir: String? = null,
    network: Boolean? = null,
    dir: String? = null,
)

fun Wfdb.updateMetadata(
    record: String,
    changes: Map<String, String?>,
    inputDir: String? = null,
    network: Boolean? = null,
    dir: String? = null,
)
```

Makes several changes to the metadata of a record, in one write of the header. Each change is applied in order, as `setMetadata` does, or, for a `null` value, as `removeMetadata` does. All the changes are checked first: if one is not valid, none is made. The second overload takes the changes as a map, applied in its iteration order (the order of insertion for the maps of `mapOf` and `linkedMapOf`).

record
: Record name (as for `setMetadata`).

changes
: The changes: a list of (key, value) pairs, or a map from key to value; a `null` value removes the entry. No two keys may differ only in letter case.

inputDir
: The input directory in which the header is read, for this call only. Default `null`: the current setting.

network
: `true` to allow reading the header from a URL, for this call only, `false` to forbid it. Default `null`: the current setting.

dir
: A directory, which must exist, in which to write the changed header instead of changing the header itself. Default `null`: the header itself is changed.

Returns Nothing (`Unit`).

Errors As `setMetadata`; also `WfdbArgumentException` if two keys of `changes` differ only in letter case ("metadata key 'dx' is given more than once (keys are compared without regard to letter case)").

```
import wfdbx.*

writeRecord("pat", IntArray(360), nsig = 1, fs = 360.0, gains = doubleArrayOf(200.0),
            adcres = intArrayOf(12), units = listOf("mV"), metadata = listOf("Age" to "63", "Sex"
    ↪ to "F"))
updateMetadata("pat", mapOf("Age" to "64", "Sex" to null, "Dx" to "164889003"))
println(readMetadata("pat"))         // [(Age, 64), (Dx, 164889003)]
```

```
try {
    updateMetadata("pat", listOf("Dx" to "426783006", "dx" to null))
} catch (e: WfdbArgumentException) {
    println(e.message)        // metadata key 'dx' is given more than once (...)
}
println(getMetadata("pat", "Dx"))    // 164889003: nothing was changed
```

12.3.6 Settings

These functions set where files are read and written, network access and legacy mode, on a `Wfdb` object (the top-level functions: on the default instance), for the high-level functions and the library functions alike (Chapters 3 and 4). Each `set` function returns the previous setting, so that it can be restored. They are the high-level forms of the library functions `wfdbx_set_input_dir`, `wfdbx_set_network`, ... (Section 12.4), which take and return bytes and integers.

`wfdbxSetInputDir` — Set the input directory

```
fun Wfdb.wfdbxSetInputDir(dir: String?): String?
```

Sets the input directory, from which names that are not full paths are read.

`dir` A local directory, a ~ path or a URL; `null` (or `""`) restores the default: `WFDBX_INPUT_DIR`, or else the current directory.

Returns The previous input directory, or `null` if none was set.

Errors Nothing.

```
val previous = wfdbxSetInputDir("/data/mitdb")
val rec = readRecord("100")             // reads /data/mitdb/100.hea and 100.dat
wfdbxSetInputDir(previous)
```

`wfdbxGetInputDir` — Return the input directory

```
fun Wfdb.wfdbxGetInputDir(): String?
```

Returns the input directory. The function has no parameters.

Returns The input directory, or `null` if none is set (the default).

Errors Nothing.

```
println(Wfdb().wfdbxGetInputDir())      // null, unless WFDBX_INPUT_DIR is set
```

`wfdbxSetOutputDir` — Set the output directory

```
fun Wfdb.wfdbxSetOutputDir(dir: String?): String?
```

Sets the output directory, in which files named without a full path are created.

dir A local directory or a ~ path (a URL is refused, with a message); `null` (or `""`) restores the default: `WFDBX_OUTPUT_DIR`, or else the current directory.

Returns The previous output directory, or `null` if none was set.

Errors Nothing.

```
wfdbxSetOutputDir("out")
writeAnnotations("100", "qrs", longArrayOf(100, 460), symbols = listOf("N", "N"))   // out/100.qrs
```

---

wfdbxGetOutputDir

Return the output directory

```
fun Wfdb.wfdbxGetOutputDir(): String?
```

Returns the output directory. The function has no parameters.

Returns The output directory, or `null` if none is set (the default).

Errors Nothing.

```
val out = wfdbxGetOutputDir() ?: "."
```

---

wfdbxSetNetwork

Turn network access on or off

```
fun Wfdb.wfdbxSetNetwork(on: Boolean): Boolean
```

Turns network access (reading URLs) on or off. It is off unless the environment variable `WFDBX_NETWORK` is a nonzero number.

on `true` to allow reading URLs, `false` to forbid it.

Returns The previous setting.

Errors Nothing.

```
wfdbxSetNetwork(true)
wfdbxSetInputDir("https://physionet.org/files/mitdb/1.0.0/")
val rec = readRecord("100", to = TimeSpec("0:05"))      // from PhysioNet
```

---

wfdbxGetNetwork

Return whether network access is on

```
fun Wfdb.wfdbxGetNetwork(): Boolean
```

Returns whether network access is on. The function has no parameters.

Returns `true` if network access is on.

Errors Nothing.

```
if (!wfdbxGetNetwork()) println("network access is off")
```

wfdbxSetLegacy Turn legacy mode on or off

```
fun Wfdb.wfdbxSetLegacy(on: Boolean): Boolean
```

Turns legacy mode (the behavior of the original WFDB package, including its assumed signal parameters and its search of the WFDB path, Chapter 4) on or off, for the whole `Wfdb` object. The parameter `legacy` of the high-level functions turns it on for one call.

on `true` to turn legacy mode on, `false` to turn it off.

Returns The previous setting.

Errors Nothing.

```
val previous = wfdbxSetLegacy(true)
val rec = readRecord("100s")            // as the original package reads it
wfdbxSetLegacy(previous)
```

wfdbxGetLegacy Return whether legacy mode is on

```
fun Wfdb.wfdbxGetLegacy(): Boolean
```

Returns whether legacy mode is on. The function has no parameters.

Returns `true` if legacy mode is on.

Errors Nothing.

```
println(wfdbxGetLegacy())               // false, unless it was turned on
```

12.3.7 The result classes

The readers return objects of the classes `Record`, `Header` and `Annotations`, whose properties are listed in Tables 12.1 to 12.3; `checkRecord` returns a `CheckReport`, which holds `Issue` objects (Tables 12.4 and 12.5). The signals of a record are described by `Siginfo` objects (Table 12.6), which are those of the WFDB library (Section 12.4). `Record`, `Header`, `Annotations`, `CheckReport`, `Issue` and `BaseDate` are data classes, whose properties are read-only (`val`); `Record` and `Annotations` compare their arrays by content in `equals`. A `BaseDate` has the properties `year`, `month` (1–12) and `day` (1–31).

Table 12.1 The properties and methods of `Record`, the result of `readRecord`

Property	Contents
`name`	the record name, as given to `readRecord` (`String`)
`fs`	the sampling frequency in Hz (`Double`), or 0 if the header gives none; with `highres = true`, the highest sampling frequency of a multi-frequency record
`nsig`	the number of signals returned
`nsamples`	the number of samples per signal (the number of frames, or of sample vectors with `highres`): `samples.size / nsig`
`signalNumbers`	the number, in the record, of each signal returned (`IntArray`)
`samples`	the samples in ADC units, frame by frame (`IntArray`): `samples[i * nsig + s]` is sample `i` of signal `s`; invalid samples are `WFDB_INVALID_SAMPLE`

Property	Contents
`physical`	with `physical = true`, the samples in physical units, in the layout of `samples` (`DoubleArray`; NaN for an invalid sample); otherwise `null`
`siginfo`	a `List<Siginfo>`, one per signal returned; `desc` has no white space at either end, and `adcres` is 0 if the header does not give it
`names`	the description of each signal, such as `"MLII"` (`""` if none), without the white space that a header may put at either end of it
`units`	the physical units of each signal, or `null` where the header gives none (`List<String?>`)
`gains`	the gain of each signal, in ADC units per physical unit (`List<Double>`); 0 means that the signal is uncalibrated
`baselines`	the ADC value of each signal that corresponds to 0 physical units (`List<Int>`)
`baseTime`	the start time and date as text, to the second, such as `"[14:00:03 21/06/2024]"`, or `null` if the header gives none
`baseTimeMs`	the start time in milliseconds after midnight, such as `50403500` for 14:00:03.500 (`Long`), or `null` if `baseTime` is `null`
`baseDate`	the start date as a `BaseDate(year, month, day)`, or `null` if the header gives none
`sampfrom`	the number of the first sample returned (`Long`)
`comments`	the info strings of the header (without the `#`), and of the record's `.info` file, if any: the lines that begin with `#` after the record line, found as the header parser finds it (the first line that holds something other than spaces, tabs, CRs and LFs and does not begin, after them, with `#`; in legacy mode, as C's `getinfo`, the first line that does not begin with `#`)
`checksumOk`	`false` if the samples do not match the checksums of the header; checksums are checked only when a read reaches the end of the record, so a shorter read gives `true`
`legacy`	`true` if the record was read in legacy mode (a gain of 200 is then used for uncalibrated signals in physical units)
`at(i, s)`	sample `i` of signal `s`, in ADC units
`signal(s)`	all the samples of signal `s`, in ADC units (`IntArray`); `IllegalArgumentException` ("no signal S") if there is no signal `s`
`physicalAt(i, s)`	sample `i` of signal `s` in physical units, $(adc - baseline)/gain$ (NaN for an invalid sample); `WfdbException` ("signal N of record REC is uncalibrated") if the signal has no gain
`physicalSignal(s)`	all the samples of signal `s` in physical units (`DoubleArray`), as `physicalAt`

Table 12.2 The properties of `Header`, the result of `readHeader`

Property	Contents
`name`	the record name, as given to `readHeader`
`fs`	the sampling frequency in Hz, or 0 if the header gives none
`counterFreq`	the counter frequency, as `getcfreq()` returns it
`baseCounter`	the base counter value, as `getbasecount()` returns it
`nsamples`	the number of samples per signal (`Long`), or 0 if the header does not give it
`nsig`	the number of signals
`siginfo`	a `List<Siginfo>`, one per signal; `desc` has no white space at either end, and `adcres` is 0 if the header does not give it
`baseTime`, `baseTimeMs`, `baseDate`	the start time and date, as in `Record`

Property	Contents
`comments`	the info strings of the header (without the `#`), found after the record line as in `Record`
`metadata`	the (key, value) pairs of the metadata entries of the header (`List<Pair<String, String>>`), in file order, as `readMetadata` returns them; empty for an EDF file or a header read from the standard input

Table 12.3 The properties of `Annotations`, the result of `readAnnotations`. The annotations are in file order; each array or list has one entry per annotation.

Property	Contents
`record`	the record name, as given
`annotator`	the annotator name, such as `"atr"`
`fs`	the frequency in which `samples` are counted, normally the sampling frequency of the record; 0 if neither the record, nor the file, nor the argument `fs` of `readAnnotations` gives one
`samples`	the annotation times, in samples (`LongArray`)
`codes`	the annotation codes (`IntArray`), such as `NORMAL` or `PVC`
`symbols`	the mnemonics of the codes, such as `"N"`, `"V"` or `"+"` (`List<String>`)
`subtypes`	the subtype of each annotation (`IntArray`)
`chans`	the channel number of each annotation (`IntArray`)
`nums`	the annotator number of each annotation (`IntArray`)
`aux`	the auxiliary text of each annotation, or `null` where there is none (`List<String?>`)
`size`	the number of annotations

Table 12.4 The properties and methods of `CheckReport`, the result of `checkRecord`

Property	Contents
`record`	the record name, as given
`ok`	`true` if no issue has the severity `Severity.ERROR`
`issues`	the `List<Issue>`, in the order of the checks
`lines()`	the text form of the report, as a list of lines: one line per issue (`issue.toString()`), or the one line `no problems found`
`toString()`	the lines of `lines()`, joined by line feeds (what `println(report)` prints)

Table 12.5 The properties of `Issue`, a problem found by `checkRecord`

Property	Contents
`severity`	a `Severity`: `ERROR`, `WARNING` or `INFO`; its `text` (and `toString()`) is `"error"`, `"warning"` or `"info"`
`code`	a stable identifier of the kind of problem, such as `"checksum"` or `"gain_missing"` (Table 6.7)
`where`	where the problem is: `""` (the record), `"signal N"`, `"segment SEG"`, `"segment SEG, signal N"` or `"annotator ANN"`
`message`	the description of the problem
`toString()`	`"SEVERITY: WHERE: MESSAGE"`, or `"SEVERITY: MESSAGE"` when `where` is empty

Two `Issue` objects are equal (`==`) when their four properties are. The codes of the issues are those of Table 6.7.

Table 12.6 The properties of `Siginfo`, the specification of a signal (mutable, with C's zero values by default)

Property	Contents
`fname`	the name of the signal file, as the header gives it (`ByteArray?`)
`desc`	the signal description, such as `"MLII"` (`ByteArray?`)
`units`	the physical units, such as `"mV"`, or `null` if not specified (`ByteArray?`)
`gain`	the gain, in ADC units per physical unit (`Double`; 0: uncalibrated)
`initval`	the value of the first sample
`group`	the signal group number (the signals of one group share one file)
`fmt`	the storage format, such as `16`, `212` or `516`
`spf`	the number of samples per frame (more than 1 for the faster signals of a multi-frequency record)
`bsize`	the block size, in bytes (0 for ordinary files)
`adcres`	the ADC resolution, in bits (0: not specified)
`adczero`	the ADC output for an input of 0 V
`baseline`	the ADC value that corresponds to 0 physical units
`nsamp`	the number of samples (`Long`; 0 if unknown)
`cksum`	the 16-bit checksum of the samples

12.3.8 Warnings and errors

Exceptions. The high-level functions report errors by exceptions (Table 12.7), whose messages say what failed. Each function checks its arguments before it creates or changes a file, so an invalid argument leaves the files as they were. The library may also print a message on the standard error when it cannot open a file (as in `init: can't open header for record REC`); `wfdbquiet()` turns these messages off (Section 12.4).

Table 12.7 The exceptions of the high-level API

Exception	Thrown when
`WfdbException`	every error of a high-level function: a header, signal file or annotation file cannot be found, read, created or written completely (for example on a full disk; the writers then leave the existing files as they were); a signal or annotation file ends before its header or its end-of-file marker says ("can't read signal data of record REC (getvec status -3)", even if a checksum error is found in the same frame); an argument is invalid, or the operation needs a value that the record does not give (a time in seconds without a sampling frequency, physical units without a gain); the directory given as `dir` does not exist
`WfdbArgumentException`	a subclass of `WfdbException`, thrown for an invalid argument by `writeRecord` and `writeAnnotations` (before any file is touched: text that is not Latin-1, a reserved annotation code or time, ...), by the checks of `readAnnotations`, and by the metadata functions (and for the `metadata` argument of `writeRecord`): an invalid key or value, a key given twice, a header with two entries for the key to set, a header without a record line, a header read from a URL without `dir`, an empty record name or `"-"` (Python's `ValueError`); a header that cannot be read or written, or a `dir` that does not exist, is a plain `WfdbException` (Python's `IOError`)

The checksum warning. When `readRecord` reads to the end of a record whose samples do not match the checksums of its header, it returns all the samples and warns: "checksum error in

record REC: the signal data do not match the checksums in its header". `checksumWarning = false` turns the warning off; `checksumOk` of the result is `false` either way.

The duplicate-key warning. When a header holds two metadata entries whose keys differ only in letter case (written by other software), `readMetadata`, `getMetadata` and `readHeader` use the first and warn: "header of record REC has more than one entry for key KEY (lines L1, L2); the first is used". `setMetadata` and `updateMetadata` then refuse to change that key (`WfdbArgumentException`), and `removeMetadata` removes all of its entries.

Both warnings go to the function given as `onWarning`, which receives the message; by default, the library prints `WFDB-x:` followed by the message on the standard error, through `wfdb_error`, so that `wfdbquiet()` silences it and `wfdberror()` returns it:

```
import wfdbx.*

// A record whose last sample is changed after it was written: the checksum fails.
writeRecord("ck", IntArray(100) { it }, nsig = 1, fs = 100.0, gains = doubleArrayOf(100.0),
            adcres = intArrayOf(12), units = listOf("mV"))
java.io.RandomAccessFile(wfdbxGetOutputDir() + "/ck.dat", "rw").use { f ->
    f.seek(f.length() - 2)
    f.write(byteArrayOf(0, 0))
}
val caught = ArrayList<String>()
var rec = readRecord("ck", onWarning = { caught.add(it) })
println("${rec.nsamples} ${rec.checksumOk}")    // 100 false: all the samples
println(caught[0])
// checksum error in record ck: the signal data do not match the checksums in its header
rec = readRecord("ck", checksumWarning = false)      // no warning

// Two entries for one key: the first is used, with a warning.
writeRecord("dup", IntArray(10), nsig = 1, fs = 100.0, gains = doubleArrayOf(0.0),
            adcres = intArrayOf(12), comments = listOf(" Age: 63", " AGE: 64"))
println(getMetadata("dup", "age", onWarning = { caught.add(it) }))     // 63
println(caught[1])
// header of record dup has more than one entry for key Age (lines 3, 4); the first is used
try {
    setMetadata("dup", "Age", "65")
} catch (e: WfdbArgumentException) {
    println("WfdbArgumentException: ${e.message}")
}
println(removeMetadata("dup", "age"))       // 2
```

```
import wfdbx.*

wfdbquiet()                           // no library messages on the standard error
try {
    readRecord("nosuchrecord")
} catch (e: WfdbException) {
    println("WfdbException: ${e.message}")    // WfdbException: can't read record nosuchrecord
}
wfdbverbose()

writeRecord("raw", IntArray(10) { it }, nsig = 1, fs = 100.0, gains = doubleArrayOf(0.0),
            adcres = intArrayOf(12))          // uncalibrated
try {
    readRecord("raw", physical = true)
} catch (e: WfdbException) {
```

```
    println("WfdbException: ${e.message}")    // WfdbException: signal 0 of record raw is
    ↪ uncalibrated
}
```

12.4 The WFDB library functions in Kotlin

Every function of the WFDB library is available under its C name, with the same arguments, in the same order, and the same return values, so that programs and documentation written for the C library carry over directly. The functions, their arguments and their results are documented once, for all the languages, in Chapter 13; this section gives the rules by which the C declarations become Kotlin functions. The KDoc comment of each function documents it, including what WFDB-x changes in it.

12.4.1 Conventions

The library state. All the state that the C library keeps in global variables (the WFDB path, the open record, its signals, the annotators, the calibration list, the settings) is held by a `Wfdb` object. Each C function is an extension function of `Wfdb` with the C name and arguments, `fun Wfdb.isigopen(record, siarray, nsig)`, called as `w.isigopen(...)`, and also a top-level function with the same name and arguments, `isigopen(...)`, which uses the default instance of the calling thread (`wfdbDefault()`), for line-by-line ports of C programs. Separate `Wfdb` objects share no state (except the process-wide structures of Section 12.3.1), so several records can be open at once, in one thread or in several; a `Wfdb` object is used by one thread at a time (it may be handed from one thread to another). As in C, the open record, its input signals, the output signals and the annotators remain open between calls, until `wfdbquit()` closes them; `wfdbquit()` leaves the input and output directories and the network setting as they are. There are no destructors: an output file is completed by `wfdbquit()` (or the function that closes it), and `wfdbx.cstd.exit` flushes the open output files as C's `exit` does, but writes no annotation end marks and no header updates.

Names. The functions, the constants (`WFDB_READ`, `WFDB_INVALID_SAMPLE`, `WFDB_HIGHRES`, ...) and the annotation codes (`NORMAL`, `PVC`, ..., Chapter 15) keep their C names. The functions of `<wfdb/ecgmap.h>` are provided under their macro names (`isqrs`, `map1`, `map2`, ...).

Strings are bytes. WFDB files are not UTF-8: every string of the C-style API is a `ByteArray`, which, like a C string, ends at its first NUL byte, and C's `NULL` is `null`. Where natural, a function also has an overload that takes a `String`, one character per byte (ISO-8859-1): `"..".latin1()` and `bytes.latin1()`, from `wfdbx.cstd`, convert. Where both overloads exist, a `null` argument is written `null as ByteArray?`, as in `sampfreq(null as ByteArray?)`. Functions that return a C string (`timstr`, `mstimstr`, `datstr`, `getinfo`, ...) return a new `ByteArray` (a copy of C's static buffer); `getinfo` returns `null` after the last info string.

Numbers. C's `int` and `WFDB_Sample` are `Int`, `long`, `WFDB_Time` and `WFDB_Date` are `Long`, and `double` and `WFDB_Frequency` are `Double`. `WFDB_Signal` and `WFDB_Group` are `Int`; negative values are invalid signal numbers, as C's unsigned values are. The `char` fields of the structures are `Int` properties that keep C's range: `anntyp`, `subtyp` and `num` from -128 to 127, `chan` from 0 to 255.

Arrays and output arguments. Where a C function fills an array that the caller passes, the Kotlin function fills an array that the caller passes:

- `isigopen(record, siarray, nsig)` (also `osigopen` and `wfdbinit`) takes an `Array<Siginfo>?`, whose elements it fills (allocated by the caller: `Array(n) { Siginfo() }`); `null` is C's `NULL`. With `nsig` ≤ 0, `isigopen` reads the header only, and returns the number of signals of the record.

- `getvec(v)`, `getframe(v)` and `putvec(v)` take an `IntArray`; a shorter array receives what fits.
- A C function that fills a structure through a pointer fills the object passed: `getann(n, annot)` fills an `Annotation`, `getcal(desc, units, cal)` a `Calinfo`. `putann(n, annot)` and `putcal(cal)` read them.
- An array of structures passed in is an array of objects, as in `osigfopen(siarray, nsig)` and `annopen(record, aiarray, nann)`, where `aiarray` is, for example, `arrayOf(Anninfo("atr", WFDB_READ))`.
- `getseginfo(sarray)` takes a `MutableList<Seginfo>?`, which it clears and fills with copies of the segments; `setmsheader(record, names, n)` takes an `Array<ByteArray>` or an `Array<String>`.

Return values and errors. The functions return what the C functions return: an `Int` status (negative on errors), a number of signals, a `Double`, a byte string or `null`. Errors are reported by these values and by a message on the standard error, as in C, never by exceptions. `wfdberror()` returns the last message; `wfdbquiet()` turns the messages off and `wfdbverbose()` on again. `wfdbx_write_errors()` returns the number of write errors since the program started (output files that could not be written, flushed or closed), `wfdbx_note_write_error()` counts one more, and `wfdbx_fclose_output(file, pname, name)` closes a file that a program writes itself, reporting "PNAME: write error on NAME" and counting it. `wfdbx_input_path(name)` and `wfdbx_output_path(name)` return the path under which a program opens a file that it reads or creates itself, where the library would open a file of that name: under the input or output directory (the missing directories of an output path are created), a `~` name expanded, a full path as given; `null` for a URL, and `"-"` as given. In legacy mode, an input name is used as given, and the output directory, if set, is a prefix of any output name. `wfdbquit()` closes the open record and annotators, and keeps the directory and network settings, the time resolution of output annotators (`getafreq()`) and the getvec mode (`getgvmode()`), as in C; the high-level functions restore the last two (Section 12.3.1).

Additions. `getvecs(out, n)` reads up to `n` sample vectors at once into the `IntArray` `out`, vector after vector, as `n` calls of `getvec` would, but decoding whole blocks of the signal files; it returns the number of vectors read, or the negative status of `getvec` if the first read failed. `getvecs_status(out, n)` returns a `GetvecsResult(count, status)`, with the value returned by the last `getvec`. The C standard library functions that the WFDB library relies on are in `wfdbx.cstd`: `cformat` and `sprintf` (to a `ByteArray`), `printf` and `eprintf`, which format numbers exactly as the GNU C library does, `strtol`, `strtod`, `sscanf`, ...

Settings. The settings functions of WFDB-x (`wfdbx_set_input_dir`, `wfdbx_get_input_dir`, `wfdbx_set_output_dir`, `wfdbx_get_output_dir`, `wfdbx_set_network`, `wfdbx_get_network`, `wfdbx_set_legacy`, `wfdbx_get_legacy`, `wfdbx_legacy_args`) keep their C names and types (byte strings, `Int` for the switches); the high-level functions of Section 12.3.6 are their Kotlin forms.

12.4.2 Package layout

12.4.3 The structures

The structures of the C library are mutable classes whose properties have the names of the fields of the C structures and C's zero values by default (Table 12.9). `copy()` is C's structure assignment, and `set(o)` copies another object into an existing one (what a C function filling `*p` does); `equals` compares contents.

```
Siginfo(fname: ByteArray? = null, desc: ByteArray? = null, units: ByteArray? = null, gain: Double
    ↪ = 0.0,
        initval: Int = 0, group: Int = 0, fmt: Int = 0, spf: Int = 0, bsize: Int = 0, adcres: Int
    ↪ = 0,
        adczero: Int = 0, baseline: Int = 0, nsamp: Long = 0, cksum: Int = 0)
```

Table 12.8 The packages of the Kotlin library

Package or file	Contents
`wfdbx`: `Wfdb.kt`	the class `Wfdb`, `wfdbDefault()`, `withDefault`
`Types.kt`	the constants and structures of `wfdb.h`
`WfdbFile.kt`, `Wfdbio.kt`	files, the input and output directories, network access, error messages (`lib/wfdbio.c`)
`Signal.kt`, `SignalTime.kt`, `Flac.kt`	headers, signal files in every format, time and date conversions (`lib/signal.c`), the FLAC codec
`Annot.kt`, `Calib.kt`, `Wfdbinit.kt`, `Ecgcodes.kt`	annotation files (`lib/annot.c`), the calibration list (`lib/calib.c`), `wfdbinit`, the annotation codes and `ecgmap.h`
`Record.kt`, `Check.kt`, `Metadata.kt`	the high-level API (Section 12.3)
`wfdbx.cstd`	the C standard library emulation (`printf`, `scanf`, `strtol`, `rand`, the standard streams, `exit`, `latin1`) and the platform layer
`wfdbx.apps`	the command-line tools, one file per tool, as functions `fun Wfdb.rdsampMain(argv: Array<ByteArray>): Int`, ...
`wfdbx.cli`	the command-line driver `wfdbx.cli.MainKt` (JVM)

```
Seginfo(recname: ByteArray = ByteArray(0), nsamp: Long = 0, samp0: Long = 0)
Anninfo(name: ByteArray = ByteArray(0), stat: Int = 0)      // also Anninfo(name: String, stat:
    ↪ Int)
Annotation(time: Long = 0, anntyp: Int = 0, subtyp: Int = 0, chan: Int = 0, num: Int = 0, aux:
    ↪ ByteArray? = null)
Calinfo(low: Double = 0.0, high: Double = 0.0, scale: Double = 0.0, sigtype: ByteArray? = null,
        units: ByteArray? = null, caltype: Int = 0)
```

Table 12.9 The classes of the C structures

Class (C structure)	Properties
`Siginfo` (`WFDB_Siginfo`)	`fname`, `desc`, `units`, `gain`, `initval`, `group`, `fmt`, `spf`, `bsize`, `adcres`, `adczero`, `baseline`, `nsamp`, `cksum` (Table 12.6)
`Seginfo` (`WFDB_Seginfo`)	`recname` (the name of the segment), `nsamp` (its number of samples), `samp0` (the number of its first sample in the record)
`Anninfo` (`WFDB_Anninfo`)	`name` (the annotator name, the suffix of the file), `stat` (`WFDB_READ` or `WFDB_WRITE` for the MIT format, `WFDB_AHA_READ` or `WFDB_AHA_WRITE` for the AHA format)
`Annotation` (`WFDB_Annotation`)	`time` (in sample intervals), `anntyp` (the annotation code), `subtyp`, `chan`, `num`, `aux` (`null`, or bytes whose first byte is the length of the text that follows, as in C)
`Calinfo` (`WFDB_Calinfo`)	`low`, `high` (the levels of the calibration pulse, in physical units), `scale` (physical units per cm), `sigtype` (matched against the beginning of a signal description), `units`, `caltype`

12.4.4 Examples

Reading the samples and the annotations of a record, as a C program does, on a `Wfdb` object of its own (the input directory is that of the examples):

```
import wfdbx.*
import wfdbx.cstd.latin1
```

```
val w = Wfdb()                                  // a library state of its own
w.wfdbx_set_input_dir(wfdbxGetInputDir()!!)
val si = Array(2) { Siginfo() }                 // filled with the signal specifications
val nsig = w.isigopen("100s", si, 2)            // open 2 signals
println("$nsig ${si.map { "${it.desc?.latin1()} ${it.fmt} ${it.gain}" }}")   // 2 [MLII 212 200.0,
    ↪ V5 212 200.0]
println(w.sampfreq(null as ByteArray?))         // 360.0: the frequency of the open record

val v = IntArray(nsig)                          // receives one sample per signal
w.isigsettime(w.strtim("0:30"))                 // skip to 30 seconds
w.getvec(v)
println("${v.toList()} ${(0 until nsig).map { w.aduphys(it, v[it]) }}")   // [947, 968] [-0.385,
    ↪ -0.28]

val frames = IntArray(360 * nsig)
println(w.getvecs(frames, 360))                 // 360: the next second, frame by frame

val a = Annotation()
w.annopen("100s", arrayOf(Anninfo("atr", WFDB_READ)), 1)
var n = 0
while (w.getann(0, a) == 0) {                   // 0: an annotation was read; -1: the end
    if (n < 2) println("${w.mstimstr(a.time).latin1().trim()} ${w.annstr(a.anntyp).latin1()}")
    ↪ // 0:00.050 +, 0:00.214 N
    n++
}
println("$n annotations")                       // 75 annotations

w.wfdbquiet()                                   // no messages on the standard error
println("${w.isigopen("nosuchrecord", null, 0)} ${w.wfdberror().latin1().trim()}")
// -1 init: can't open header for record nosuchrecord
w.wfdbverbose()
w.wfdbquit()                                    // close everything
```

Writing a record and an annotation file, sample by sample and annotation by annotation:

```
import wfdbx.*
import wfdbx.cstd.latin1

val w = Wfdb()
w.wfdbx_set_output_dir(wfdbxGetOutputDir()!!)
val out = arrayOf(Siginfo(fname = "ramp.dat".latin1(), desc = "ramp".latin1(), units = "mV".latin1
    ↪ (),
                          gain = 200.0, fmt = 16, adcres = 12))
println(w.osigfopen(out, 1))                    // 1: one output signal
w.setsampfreq(250.0)
for (k in 0 until 500) w.putvec(intArrayOf(k))
println(w.newheader("ramp"))                    // 0: ramp.hea is written
w.annopen("ramp", arrayOf(Anninfo("qrs", WFDB_WRITE)), 1)
for (t in longArrayOf(100, 350)) w.putann(0, Annotation(time = t, anntyp = NORMAL))
w.wfdbquit()                                    // completes ramp.dat and ramp.qrs

println(readAnnotations("ramp", "qrs").samples.toList())    // [100, 350]
```

12.5 The command-line tools in Kotlin

The 33 command-line tools (`rdsamp`, `rdann`, `wrsamp`, `xform`, ...) run on the JVM, through the launcher `kotlin/bin/wfdbx` (a POSIX shell script), followed by the name of the tool and its arguments. The tools, their options, their output and their exit status are those of the C tools, and are documented in Chapter 14. The launcher needs the jar built by `./gradlew fatJar` (Section 12.1). In the directory `kotlin`:

```
export WFDBX_INPUT_DIR=../tests/data
bin/wfdbx rdsamp -r 100s -t 10 -p      # the first 10 seconds of 100s, in physical units
bin/wfdbx rdann -r 100s -a atr         # its annotations
bin/wfdbx rdsamp --legacy -r 100s      # legacy mode: --legacy as the first argument
```

The command also takes these options, instead of a tool:

Table 12.10 The options of the launcher `bin/wfdbx`

Command	Effect
`bin/wfdbx TOOL [ARGUMENTS ...]`	runs the tool `TOOL`, and exits with its status
`bin/wfdbx --list`	lists the names of the tools, one per line
`bin/wfdbx --version`	prints the version, `WFDB-x 1.2.3`
`bin/wfdbx --help`, `bin/wfdbx -h`	prints the usage

Without arguments, the command prints the usage and exits with status 1; an unknown tool is an error ("wfdbx: unknown tool 'TOOL' (try 'wfdbx –list')", status 1), and an uncaught exception is reported as `wfdbx: error: MESSAGE` (status 1). When an output file cannot be written completely (a full disk, for example), or the standard output cannot be written ("TOOL: write error on standard output"), a tool exits with status 1; in legacy mode, it keeps the original status, as in C.

The files that a tool reads or creates itself are found where the library finds its own files: the `-i` file of `wrsamp`, `edf2mit`, `wav2mit`, `a2m` and `ad2m`, the input files of `ahaecg2mit`, the `-S` script of `xform` and the signal file that `wfdbdesc` measures are read under the input directory, and the WAV file of `mit2wav`, the EDF file of `mit2edf`, the master header of `snip -m` and the `.des`, `.ecg`, `.key` and `.txt` files of `parsescp` are created under the output directory (with `wfdbx_input_path` and `wfdbx_output_path`, Section 12.4). Such a file cannot be read from a URL. In legacy mode, they are read as named, relative to the current directory, and the output directory, if set, is a prefix of their names.

The launcher. `bin/wfdbx` runs `java -cp build/libs/wfdbx-all.jar wfdbx.cli.MainKt TOOL ...`, or the jar named by the environment variable `WFDBX_KOTLIN_JAR`. It starts the JVM with options for a short start-up (its first compiler only, the serial collector, and the class data sharing archive `wfdbx-all.jsa` that `./gradlew fatJar` makes next to the jar), and moves `JAVA_TOOL_OPTIONS` to the command line. A link to `bin/wfdbx` named after a tool runs that tool. Without the launcher, the jar runs the tools directly:

```
java -cp build/libs/wfdbx-all.jar wfdbx.cli.MainKt rdsamp -r 100s -t 10
```

The tools as functions. Each tool is a function of the package `wfdbx.apps`, such as

```
fun Wfdb.rdsampMain(argv: Array<ByteArray>): Int
```

where `argv` is C's `argv`, the program name first, and the result is the exit status. A tool may end the process (with `exit`, as C programs do), so an app uses the high-level functions or the library functions instead.

Environment variables. The tools, like the rest of the library, use the environment variables of Table 12.11. On the JVM, which cannot change its own environment, `setenv` (in `wfdbx.cstd`) keeps values that the library and the processes it starts see.

Table 12.11 The environment variables used by the Kotlin library

Variable	Meaning
`WFDBX_INPUT_DIR`	the input directory (a directory, a ~ path or a URL), when `wfdbxSetInputDir` does not set one
`WFDBX_OUTPUT_DIR`	the output directory, when `wfdbxSetOutputDir` does not set one
`WFDBX_NETWORK`	network access is on if it is a nonzero number, when `wfdbxSetNetwork` does not set it
`WFDBANNSORT`	`0` to keep annotation files in the order written (no sorting when they are closed)
`WFDBGVMODE`	the default getvec mode (`setgvmode` with a negative mode)
`WFDBCAL`	the calibration file read by `calopen(null)` (default `wfdbcal`)
`WFDB`	the WFDB path, used in legacy mode only (outside legacy mode, it is ignored, with a message)
`WFDB_FLAC_CHECK_MD5`	if set, the MD5 signature of FLAC signal files is verified when they are read
`WFDBX_KOTLIN_JAR`	the jar that `bin/wfdbx` runs (default `build/libs/wfdbx-all.jar`)

12.6 Testing

The unit tests of the Kotlin library are in `kotlin/src/commonTest` (run on every target) and `kotlin/src/jvmTest` (JVM only). In the directory `kotlin`:

```
./gradlew jvmTest                            # the unit tests, on the JVM
./gradlew compileCommonMainKotlinMetadata    # the common code uses only common APIs
./gradlew jvmTest -Pwfdbx.linux=true linuxX64Test   # also on Linux/Native, with its toolchain
```

The unit tests check the library functions with expected values printed by the C library, the high-level functions, the bulk paths against the per-frame paths, the header metadata functions against the cases shared by the seven languages (in `tests/metadata`: results, warnings and the headers, byte for byte), the examples of `kotlin/README.md`, and `checkRecord` against the reports of the Python implementation on a corpus of damaged records (this test is skipped, with a message, when `python3` with NumPy is not available). Network access is tested against a local HTTP server (127.0.0.1). The tests copy `tests/data` to a scratch directory, and never change the current directory.

The cross-language conformance suite compares the output of the tools with that of the original C programs, case by case: 1155 cases, and 886 in legacy mode, which runs each tool with `--legacy`. It runs the launcher `kotlin/bin/wfdbx`, so the jar is built first. From the root of the repository:

```
(cd kotlin && ./gradlew fatJar)
PYTHONPATH=$PWD/python/src python3 tests/conformance/harness.py check --impl kotlin -j8
PYTHONPATH=$PWD/python/src python3 tests/conformance/harness.py check --impl kotlin -j8 --legacy
```

PART III

Reference

CHAPTER 13

The WFDB library functions

Besides its high-level functions, which read and write whole records in the idiom of each language, WFDB-x provides the functions of the C library of the WFDB Software Package 10.7.0 (`libwfdb`, declared in `<wfdb/wfdb.h>`) in all seven languages. They keep the C names (`isigopen`, `getvec`, `getann`, `strtim`, ...), the C arguments and their order, the C return values and error codes, the messages that the C library prints, and its global state: the open record, signals and annotators, the sampling frequency and base time used by the time conversion functions, the calibration list, the tables of annotation mnemonics. A program written for the C library, or a passage of the *WFDB Programmer's Guide*, carries over call by call; the command-line tools of WFDB-x (Chapter 14) are written with these functions, as the C tools are with `libwfdb`.

The behavior is that of the C library, edge cases included, with the deliberate changes of WFDB-x described in Chapter 4: no built-in sampling frequency, gain, ADC resolution or units; files found only where they are named (an input directory, an output directory, network access off by default) instead of being searched for in the WFDB path; out-of-order annotation files sorted in the same process, without losing annotations; an output signal file never one of the input files; write errors counted. Legacy mode (`wfdbx_set_legacy`) restores the original behavior. Each entry below says what changes, if anything. WFDB-x adds a few functions of its own, whose names begin with `wfdbx_` (Section 13.11), and a reader of many sample vectors at once (`getframes`, or `getvecs`).

Since the names, the arguments and the behavior are the same in the seven languages, each function is documented once, in C notation: the first listing of each entry is the prototype of the C library (for the WFDB-x additions, the declaration of the C++ implementation, `cpp/include/wfdbx/wfdb.hpp`, which is in the same style). Section 13.1 shows how the C conventions (arrays filled by a function, output arguments, `NULL`, strings, structures) appear in each language. The record `100s` used in the examples is the sample record included with WFDB-x (the first minute of record 100 of the MIT-BIH Arrhythmia Database, in `tests/data`).

> **Note** These functions are for programs that need the C library's model: reading sample by sample, writing headers and annotation files piece by piece, or porting an existing C program. To read or write a whole record or annotation file, the high-level functions of each language (`read_record`, `write_annotations`, ...; Chapters 6 to 12) are simpler and faster.

13.1 Calling the library functions from each language

In every language the functions have their C names and take their arguments in the C order. What differs is how a language expresses C's pointers: arrays that a function fills, output arguments, `NULL`, and strings. Table 13.1 summarizes these conventions; the notes and the example that follow give the details for each language. Each example opens the record `100s` (from the current directory, the default input directory) with `isigopen`, reads one sample of each signal with `getvec`, and prints the description of signal 0 and the two samples (`MLII 995 1011`).

The conventions common to all languages:

- Signal, group and annotator numbers start at 0, as in C, also in the languages whose arrays start at 1 (Julia, R, MATLAB/Octave).
- Errors are reported as in C: by the return value, with a message on the standard error output, not by exceptions. `wfdberror` returns the last message, and `wfdbquiet` stops the messages from being printed.
- The state of the library is kept between calls, as in C. `wfdbquit` closes the files and resets the record's state; the input and output directories and the network setting are kept.
- Text is handled as bytes, one character per byte (ISO-8859-1 where a language's strings are Unicode), so that every byte of a header or an annotation file is kept.
- An annotation's `aux` field is either absent (C's `NULL`) or a sequence of bytes whose first byte is the length of the data that follow, as in C.

Table 13.1 The C conventions in each language.

Language	Calling	Arrays and output arguments	NULL	Strings returned
Python	`wfdbx.isigopen(...)` (module `wfdbx`)	a list passed by the caller is filled; an object (`Annotation`, `Calinfo`) passed by the caller is filled	`None`	`str`; `aux` is `bytes`
C++	`wfdbx::isigopen(...)` (namespace `wfdbx`, header `<wfdbx/wfdb.hpp>`)	as in C: arrays and structures allocated by the caller, passed by pointer	`nullptr`	`std::string`, or `std::optional<std::string>` where C may return `NULL`
Rust	`wfdbx::isigopen(...)`, or the method `w.isigopen(...)` of a `Wfdb` value	a mutable slice or reference passed by the caller is filled	`None` (`wfdbx::NULL`)	`Vec<u8>`, or `Option<Vec<u8>>` where C may return `NULL`
Julia	`isigopen(...)` after `using WFDBX`, or `WFDBX.isigopen(...)`; optionally a `Wfdb` object first	a vector or mutable structure passed by the caller is filled; a `Ref` receives a pointer result	`nothing`	`Vector{UInt8}`; `String` for `timstr`, `mstimstr`, `datstr`
R	`isigopen(...)` (package `wfdbx`)	returned with the C result in a list (`r$nsig`, `r$si`; `g$stat`, `g$v`)	`NULL`	character strings; `aux` is a raw vector
MATLAB/ Octave	`wfdbx.isigopen(...)` (package `+wfdbx`)	returned as extra output values (`[nsig, si] = ...`)	`[]`	`char` row vectors; `aux` is `uint8`
Kotlin	`isigopen(...)` (package `wfdbx`), or the extension function `w.isigopen(...)` of a `Wfdb` object	an array or object passed by the caller is filled	`null`	`ByteArray`, or `ByteArray?` where C may return `NULL`

13.1.1 Python

The functions, the structures and the constants are in the package `wfdbx` (`import wfdbx`). The C structures are the classes `Siginfo`, `Anninfo`, `Annotation`, `Calinfo` and `Seginfo`, with the C field names; their constructors take the fields as keyword arguments, with C's zero values, except that `Siginfo.spf` is 1 by default.

- Where C fills an array of structures (`isigopen`, `osigopen`), the caller passes a list, which is emptied and receives one `Siginfo` per signal. Where C fills an array of samples (`getvec`, `getframe`), the caller passes a list with room for every value (`[0] * nsig`), whose first elements are replaced. `putvec` reads a list (or any sequence).
- Where C fills a structure (`getann`, `getcal`), the caller passes an object (`Annotation()`, `Calinfo()`), whose fields are replaced.
- `getseginfo()` takes no argument and returns the list of `Seginfo` objects (empty if the record is not a multi-segment record), instead of a count and a pointer.
- The counts that only give the length of an array have defaults: `annopen(record, aiarray)`, `osigfopen(siarray)`, `setheader(record, siarray)` and `setmsheader(record, names)` use the length of the list; `isigopen` and `osigopen` take `siarray=None, nsig=0`, `getinfo` and `calopen` take `None`.
- Strings are `str`; an annotation's `aux` is `None` or `bytes`, and `Annotation.aux_string()` returns its text. `wfdb_error(fmt, *args)` formats as C's `printf`.
- `getframes(n)` reads up to `n` sample vectors into a NumPy array (Section 13.11).
- The functions of `<wfdb/ecgmap.h>` (`isann`, `isqrs`, `map1`, ...) and the annotation codes (`NORMAL`, `PVC`, ...) are in the package; `isann` returns `True` or `False` (C: 1 or 0).

```
import wfdbx

si = []
nsig = wfdbx.isigopen("100s", si, 2)    # si receives 2 Siginfo objects
v = [0] * 2
if nsig == 2 and wfdbx.getvec(v) > 0:   # v receives one sample per signal
    print(si[0].desc, v[0], v[1])       # MLII 995 1011
wfdbx.wfdbquit()
```

13.1.2 C++

`<wfdbx/wfdb.hpp>` declares the functions, types, structures and constants in namespace `wfdbx`; `<wfdbx/ecgcodes.hpp>` defines the annotation codes, and `<wfdbx/ecgmap.hpp>` the macros of `<wfdb/ecgmap.h>` as inline functions (`isann`, `isqrs`, ..., which call `wfdb_isann`, `wfdb_isqrs`, ...).

- Arrays and structures are allocated by the caller and passed by pointer, as in C. String arguments are `const char *`, and `nullptr` is accepted where C accepts `NULL`.
- Strings are returned by value, as `std::string`, or as `std::optional<std::string>` where the C function may return `NULL` (`annstr`, `ecgstr`, `anndesc`, `getinfo`, `wfdbfile`); unlike C's pointers to static buffers, they stay valid after later calls.
- The string fields of `WFDB_Siginfo`, `WFDB_Calinfo` and `WFDB_Annotation` (including `aux`) are `std::optional<std::string>`, since C distinguishes `NULL` from `""`; `WFDB_Anninfo::name` is a `std::string`. The structures are also available under the shorter names `Siginfo`, `Anninfo`, `Annotation`, `Calinfo` and `Seginfo`.
- `WFDB_Time` is `long long` (64 bits); `getseginfo` takes a `const WFDB_Seginfo **`.

- `getvecs` reads many sample vectors at once (Section 13.11). The library is not thread-safe, as in C.

```
#include <cstdio>
#include <wfdbx/wfdb.hpp>
using namespace wfdbx;

int main() {
    WFDB_Siginfo si[2];
    WFDB_Sample v[2];
    int nsig = isigopen("100s", si, 2);
    if (nsig == 2 && getvec(v) > 0)
        std::printf("%s %d %d\n", si[0].desc->c_str(), v[0], v[1]);
    wfdbquit();
    return 0;
}
```

13.1.3 Rust

The state of the library is held by a `Wfdb` value, and each function is a method of it (`w.isigopen(...)`). Every function is also a free function of the crate with the same name and arguments (`wfdbx::isigopen(...)`), which uses a default `Wfdb` value, one per thread; separate `Wfdb` values share no state, so that several records can be open at once. The functions that use no state of their own are free functions only: `wfdbldflags`, `wfdbcflags`, `wfdbdefwfdb`, `wfdbdefwfdbcal`, the count of write errors (`wfdbx_write_errors`, `wfdbx_note_write_error`, which belongs to the process), `isann`, `ammap` and `mamap` (and their `wfdb_` forms), and the macros of `<wfdb/ecgmap.h>` that use the mapping tables (`isqrs`, `map1`, ...), which use those of the default value (the methods `w.wfdb_isqrs(...)`, `w.wfdb_map1(...)`, ... use those of `w`).

- Where C fills an array, the caller passes a mutable slice (`&mut si`, `&mut v`); where C fills a structure, a mutable reference (`&mut annot`). C's `NULL` array, as in `isigopen(record, NULL, 0)`, is an empty slice.
- String arguments accept `&str`, `&[u8]` or `b"..."`; where C accepts `NULL`, they also accept `Option`s, and `wfdbx::NULL` is a typed `None`. Strings are returned as `Vec<u8>`, or `Option<Vec<u8>>` where C may return `NULL` (`anndesc`, `getinfo`, `wfdbfile`, ...).
- C's `int` is `i32`, `long` and `WFDB_Time` are `i64`, `WFDB_Signal`, `WFDB_Group` and `WFDB_Annotator` are `u32`. The structures (`Siginfo`, `Anninfo`, `Annotation`, `Calinfo`, `Seginfo`) have the C field names; `Default::default()` gives C's zero values; the `char` fields of `Annotation` are `i8` (`chan`: `u8`).
- `getseginfo(&mut segs)` takes an `&mut Option<Vec<Seginfo>>`, which receives a copy of the segment table (`None` if there is none).
- `wfdb_error(msg)` takes a message already formatted (with the crate's `cformat!` macro, which formats as C's `printf`), and `wfdb_error_fmt(fmt, args)` formats it.
- `getvecs` and `getvecs_status` read many sample vectors at once (Section 13.11); `wfdbx_fclose_output` closes a file that a program wrote itself and counts a write error (Section 13.11).

```
use wfdbx::Siginfo;

fn main() {
    let mut si = vec![Siginfo::default(); 2];
    let nsig = wfdbx::isigopen("100s", &mut si, 2);
    let mut v = vec![0i32; 2];
    if nsig == 2 && wfdbx::getvec(&mut v) > 0 {
```

```
        let desc = si[0].desc.clone().unwrap_or_default();
        println!("{} {} {}", String::from_utf8_lossy(&desc), v[0], v[1]);
    }
    wfdbx::wfdbquit();
}
```

13.1.4 Julia

The functions are in the module `WFDBX`. `using WFDBX` makes the names of most of them available; those that it does not export are called with the module prefix: `WFDBX.wfdb_error`, `WFDBX.wfdb_me_fatal`, and the function forms of the macros of `<wfdb/ecgmap.h>`, `WFDBX.wfdb_isann`, `WFDBX.wfdb_isqrs`, ... (`isann`, `isqrs`, ... are exported). The state of the library is held by a `Wfdb` object: every function that uses it takes one as an optional first argument (`isigopen(w, "100s", si, 2)`) and otherwise uses the default object, `lib()`; `Wfdb()` creates an independent one. The functions that use no such state take none: `wfdbldflags`, `wfdbcflags`, `wfdbdefwfdb`, `wfdbdefwfdbcal`, `isann`, `ammap`, `mamap` (and their `wfdb_` forms), and `wfdbx_write_errors` and `wfdbx_note_write_error`, whose count belongs to the process.

- Where C fills an array, the caller passes a vector: of `Siginfo` structures (`[Siginfo() for _ in 1:n]`), or of samples, which must be a `Vector{Int32}`. Where C fills a structure, the caller passes a mutable structure (`Annotation()`). `getseginfo(r)` takes a `Ref`, which receives a vector of `Seginfo` (or `nothing`).
- C's `NULL` is `nothing`. Arguments may be `String`s or bytes. Text is returned as bytes (`Vector{UInt8}`; `String(x)` converts it), except the results of `timstr`, `mstimstr` and `datstr`, which are `String`s; the string fields of the structures are bytes, or `nothing`.
- The integer types are those of C: `WFDB_Sample` and `int` are `Int32`, `WFDB_Time` and `long` are `Int64`; the `char` fields of `Annotation` are `Int8` (`chan`: `UInt8`).
- `wfdb_error(fmt, args...)` formats as C's `printf`. `wfdbx_legacy_args(args)` takes the arguments without the program name (Section 13.11).
- `getvecs` and `getvecs_status` read many sample vectors at once (Section 13.11).

```
using WFDBX

si = [Siginfo() for _ in 1:2]
nsig = isigopen("100s", si, 2)          # si receives the specifications
v = zeros(Int32, 2)
if nsig == 2 && getvec(v) > 0           # v receives one sample per signal
    println(String(si[1].desc), " ", v[1], " ", v[2])
end
wfdbquit()
```

13.1.5 R

The functions are those of the package `wfdbx` (`library(wfdbx)`). R has no output arguments, so a function that fills an array or a structure in C returns it, together with the C return value, in a list; the argument that C fills is omitted.

- `isigopen(record, nsig)` and `osigopen(record, nsig)` return `list(nsig, si)`: the C result and the list of `siginfo` lists; `wfdbinit(record, aiarray, nsig)` (without `nann`) returns `list(stat, si)`; `getvec()` and `getframe()` return `list(stat, v)`; `getann(an)` returns `list(stat, annot)`; `getcal(desc, units)` returns `list(stat, cal)`; `getseginfo()` returns the list of `seginfo` lists; `wfdbx_legacy_args(argv)` returns `list(argv, legacy)`.

- The C structures are lists with the C field names, made by `siginfo()`, `anninfo()`, `annotation()`, `calinfo()` and `seginfo()` (`siginfo()` gives `spf` 1). Numbers are doubles; strings are character strings, and C's `NULL` is `NULL`; an annotation's `aux` is `NULL` or a raw vector, and `aux_string(aux)` returns its text.
- The C function `sample` is `wfdb_sample`, so that it does not mask R's `sample`. The functions of `<wfdb/ecgmap.h>` accept vectors of codes. `wfdb_error(fmt, ...)` formats as C's `printf`.
- `getframes(n)` reads up to `n` sample vectors into a matrix (Section 13.11); `putframes(m)` writes the rows of a matrix with `putvec`.

```
library(wfdbx)

r <- isigopen("100s", 2)           # r$nsig, and r$si: the specifications
g <- getvec()                      # g$stat, and g$v: one sample per signal
if (r$nsig == 2 && g$stat > 0)
  cat(r$si[[1]]$desc, g$v, "\n")
wfdbquit()
```

13.1.6 MATLAB/Octave

The functions are those of the package `+wfdbx`, called with the prefix `wfdbx.` (`wfdbx.isigopen(...)`). An array or structure that C fills is returned as an additional output value, and omitted from the arguments.

- `[nsig, si] = wfdbx.isigopen(record, nsig)` and `[nsig, si] = wfdbx.osigopen(record, nsig)` return the C result and a struct array of `wfdbx.siginfo` structures; `[stat, si] = wfdbx.wfdbinit(record, aiarray, nann, nsig)`; `[stat, v] = wfdbx.getvec()` and `[stat, v] = wfdbx.getframe()` (an initial vector may be passed: its elements are kept where C leaves them unchanged); `[stat, annot] = wfdbx.getann(an)`; `[stat, cal] = wfdbx.getcal(desc, units)`; `[stat, si] = wfdbx.setheader(record, si, nsig)`; `segs = wfdbx.getseginfo()` returns a struct array; `[r, argv] = wfdbx.wfdbx_legacy_args(argv)`.
- The C structures are MATLAB structures with the C field names, made by `wfdbx.siginfo(n)`, `wfdbx.anninfo(name, stat)`, `wfdbx.annotation()`, `wfdbx.calinfo()` and `wfdbx.seginfo(n)` (`siginfo` gives `spf` 1). Samples, times and counts are doubles. Strings are `char` row vectors (record names can also be MATLAB strings), and C's `NULL` is `[]`; an annotation's `aux` is `[]` or a `uint8` row vector.
- Constants are functions without arguments: `wfdbx.WFDB_READ()`, `wfdbx.NORMAL()`, ...; `wfdbx.ecgcodes()` returns all the annotation codes in one structure. The calibration constants (`WFDB_AC_COUPLED`, ...) and `WFDB_FMT_LIST` are not defined; their values are given in Table 13.2.
- `wfdbx.wfdb_error(fmt, ...)` formats as C's `printf`. `wfdbx.getframes(n)` reads up to `n` sample vectors into a matrix (Section 13.11), and `wfdbx.putframes(M)` writes the rows of a matrix.

```
[nsig, si] = wfdbx.isigopen('100s', 2);   % si: the specifications
[stat, v] = wfdbx.getvec();               % v: one sample per signal
if nsig == 2 && stat > 0
    fprintf('%s %d %d\n', si(1).desc, v(1), v(2));
end
wfdbx.wfdbquit();
```

13.1.7 Kotlin

The functions are in the package `wfdbx` (`import wfdbx.*`). The state of the library is held by a `Wfdb` object, and each function is an extension function of it (`w.isigopen(...)`); every function is also a top-level function with the same name and arguments (`isigopen(...)`), which uses a default `Wfdb` object, one per thread. Separate `Wfdb` objects share no state; one object must be used by one thread at a time. As in Rust, `wfdbldflags`, `wfdbcflags`, `wfdbdefwfdb`, `wfdbdefwfdbcal`, `wfdbx_write_errors`, `wfdbx_note_write_error`, `isann`, `ammap`, `mamap` (and their `wfdb_` forms) and the macros of `<wfdb/ecgmap.h>` that use the mapping tables (`isqrs`, `map1`, ..., on the default object; the extension functions are `w.wfdb_isqrs(...)`, ...) are top-level functions only.

- Where C fills an array, the caller passes an array (`Array<Siginfo>`, `IntArray`); where C fills a structure, an object (`Annotation()`). `getseginfo(list)` takes a `MutableList<Seginfo>?`, which receives the segments.
- Strings are bytes (`ByteArray`), as in C; most functions also accept a `String` (one character per byte), and `latin1()` (in `wfdbx.cstd`) converts in either direction. C's `NULL` is `null`; results that C may return as `NULL` are `ByteArray?`.
- C's `int` and `WFDB_Sample` are `Int`, `long` and `WFDB_Time` are `Long`, `double` is `Double`. The structures (`Siginfo`, `Anninfo`, `Annotation`, `Calinfo`, `Seginfo`) have the C field names and C's zero values; the `char` fields of `Annotation` are `Int` properties that keep C's range (`anntyp`, `subtyp`, `num`: -128 to 127; `chan`: 0 to 255).
- `wfdb_error(msg)` takes a message already formatted (with `cformat` of `wfdbx.cstd`, which formats as C's `printf`).
- `getvecs` and `getvecs_status` read many sample vectors at once, and `wfdbx_fclose_output` closes a file that a program wrote itself (Section 13.11).

```
import wfdbx.*
import wfdbx.cstd.latin1

fun main() {
    val si = Array(2) { Siginfo() }
    val nsig = isigopen("100s", si, 2)
    val v = IntArray(2)
    if (nsig == 2 && getvec(v) > 0)
        println("${si[0].desc!!.latin1()} ${v[0]} ${v[1]}")
    wfdbquit()
}
```

Functions that exist only in some languages Every function documented in this chapter exists in the seven languages (`docs/INVENTORY.md`). The reader of many sample vectors has two names: `getframes` in Python, R and MATLAB/Octave, and `getvecs` (with `getvecs_status`) in C++, Rust, Julia and Kotlin; it is documented under `getframes`. A few helpers have no counterpart in the Python implementation, and therefore no entry here: `putframes` (R, MATLAB/Octave), `wfdbx_fclose_output` (Rust, Kotlin; in Julia, `WFDBX.wfdbx_fclose_output`, not exported), `wfdb_error_fmt` (Rust; in Kotlin, the method `w.wfdb_error_fmt`), `aux_string` (R; a method of `Annotation` in Python) and the structure constructors of R and MATLAB/Octave. In R, `sample` is named `wfdb_sample`.

13.2 Opening and closing records

A record is opened for reading by `isigopen` (its signals) and `annopen` (its annotation files), or by `wfdbinit`, which calls both; output signals are opened by `osigopen` or `osigfopen`, and output annotation files by `annopen`. Programs that write files must call `wfdbquit` (or close the files with the functions given in each entry) before they exit.

A record name is found as Chapter 4 describes: a relative name under the input directory (for reading) or the output directory (for writing), an absolute name, a `~` name or a URL as given; a name that begins with `+` keeps the files of the same kind already open, and the record name `-` is the standard input or output. A trailing `.hea` is removed from a record name.

---

annopen Open annotation files

```
int annopen(char *record, const WFDB_Anninfo *aiarray, unsigned int nann);
```

Opens, for `record`, the `nann` annotation files described by `aiarray`: for reading if their `stat` is `WFDB_READ` or `WFDB_AHA_READ`, for writing (created, or truncated) if it is `WFDB_WRITE` or `WFDB_AHA_WRITE`. The file of annotator `atr` of record `100s` is `100s.atr`. Input and output annotators are numbered separately, from 0, in the order of `aiarray`: these are the numbers given to `getann` and `putann`. All the annotation files already open are closed first (output files are completed, and sorted if needed, as by `oannclose`), unless `record` begins with `+`, which is then removed. All the `stat` values are checked before any file is opened. An input file whose contents show the other format (MIT or AHA) than its `stat` says is read in the format it has, with a warning. The modification labels (mnemonics and descriptions defined at the beginning of an input file) are applied to the tables of `annstr` and `anndesc`; the time resolution recorded in an input file is used by `getann` (see `getiaorigfreq`). An output annotator name may contain only letters, digits, `_`, `~`, `-` and `/`.

With `nann` 0, `annopen` only closes the open annotation files (`annopen("", NULL, 0)`) and reports whether that succeeded.

`record`
: The record name.

`aiarray`
: An array of `nann` `WFDB_Anninfo` structures (Section 13.12): the annotator name (`name`) and the mode (`stat`) of each file.

`nann`
: The number of annotators in `aiarray`.

Returns 0 on success; -3 if an input file cannot be opened; -4 if an output file cannot be created or its annotator name is illegal; -5 if a `stat` is not one of the four values. With `nann` 0: 0, -6 if an output file closed by the call was out of order (whether or not it could be sorted), or -7 if one could not be written completely.

Errors `annopen: can't read annotator ATR for record REC`; `annopen: can't write annotator ATR for record REC`; `annopen: illegal stat N for annotator ATR, record REC`; `init: illegal character N in annotator name`.

WFDB-x: input files are read only where their name says, output files are created under the output directory, and out-of-order output files closed by the call are sorted in the same process (Chapter 4).

isigopen Open the input signals of a record

```
int isigopen(char *record, WFDB_Siginfo *siarray, int nsig);
```

Reads the header of `record` and opens its signal files for reading. The input signals already open are closed first, unless `record` begins with `+`, which is then removed (the new signals are numbered after those open). The header also sets the sampling and counter frequencies, the base counter value, the base time and date, and the length of the record, which the time conversion functions use.

With `nsig` greater than 0, at most `nsig` signals are opened, and the structures `siarray[0]` to `siarray[n-1]` are filled with their specifications (signal numbers are assigned in the order of the header). Signals stored in the same file (a signal group) are opened together: a group is skipped if its file cannot be opened, or if its signals do not fit in the remaining entries (a record whose signals are all in one file is therefore not opened at all if `nsig` is smaller than its number of signals).

With `nsig` 0 or negative, no signals are opened (and the open input signals are closed, as above): the header is read, up to `-nsig` structures of `siarray` are filled from it (`siarray` may be `NULL` when `nsig` is 0), and the number of signals named in the header is returned. `isigopen(record, NULL, 0)` thus gives the number of signals, to allocate `siarray` before opening them, and sets the time base of the record for `strtim` and `timstr`.

A record name whose last component contains a `.`, other than in a final `.hea`, names an EDF file, read as a record (Chapter 15). A multi-segment record opens its segments as they are read (see `getseginfo`).

record
: The record name.

siarray
: An array of at least `nsig` (or `-nsig`) `WFDB_Siginfo` structures, filled by the function.

nsig
: The maximum number of signals to open; 0 or a negative number to read the header only.

Returns With `nsig` > 0: the number of signals opened (possibly fewer than `nsig`), 0 if the record has no signals or none could be opened. With `nsig` <= 0: the number of signals of the record. In either case: -1 if the header cannot be read (probably an incorrect record name), -2 if it is not a valid header, -3 if the internal limits are exceeded (the total number of samples per frame, or the skew, too large), in which case all the input signals are closed.

Errors `init: can't open header for record REC`; `isigopen: record REC has no signals`; `isigopen: none of the signals for record REC is readable`; the messages of the header parser (`init: ...`).

WFDB-x (Chapter 4): the header is read only where the record name says, and the signal files and segments that it names only from the header's directory (`init: file F named in the header of record REC is not in the header's directory` for a name that leaves it); a header without a sampling frequency, ADC resolution or units leaves them unspecified (0, 0, `NULL`) instead of assuming 250 Hz, a resolution and millivolts; a header with a negative or impossibly large number of signals gives `init: unexpected EOF in header file for record REC` (C tries to allocate memory for that count).

osigopen

Open output signals as described by a header

```
int osigopen(char *record, WFDB_Siginfo *siarray, unsigned int nsig);
```

Reads the header of the existing record `record` and opens for writing (creating or truncating them) the signal files of its first `nsig` signals, with the specifications of the header; `siarray` receives them (with `nsamp` and `cksum` 0). The output signals already open are closed first, unless `record` begins with `+`, which is then removed. The samples are then written with `putvec`, and a header for the new signals with `newheader`. A header of an existing database record names that record's signal files, which would be overwritten: the header used here is usually written for the purpose (a "piped" record such as `8l`, whose signal files are `data0` and `data1`).

record
The record whose header specifies the output signals.

siarray
An array of at least `nsig` `WFDB_Siginfo` structures, filled by the function.

nsig
The number of output signals to open.

Returns The number of output signals opened (`nsig`, 0 when `nsig` is 0); -1 if the header cannot be read; -2 if it is not valid, or if the signals of a group differ in format or block size; -3 if the record has fewer than `nsig` signals or a signal file cannot be created.

Errors `osigopen: record REC has fewer signals than needed`; `osigopen: can't open FILE`; `osigopen: error in specification of signal N or M`.

The Programmer's Guide allows `nsig` 0 with `siarray` `NULL`, to close the output signals; the C function then reads before the start of its internal array. In WFDB-x, in every language, `osigopen(record, NULL, 0)` closes the output signals already open (unless `record` begins with `+`), reads the header (and returns -1 if it cannot be read, -2 if it is not valid), opens nothing, and returns 0 (Section 4.7). `osigfopen(NULL, 0)` also closes the output signals.

WFDB-x: a signal file is created under the output directory, and a signal file that is one of the signal files of the open input record is refused (`osigopen: output file FILE is also an input file`, -3; legacy mode does not check); see Chapter 4.

osigfopen

Open output signals as described by the caller

```
int osigfopen(const WFDB_Siginfo *siarray, unsigned int nsig);
```

Closes the open output signals, and opens `nsig` output signals as `siarray` describes them: file name, description, units, gain, format, samples per frame, block size, ADC resolution and zero, baseline (`initval`, `nsamp` and `cksum` are not used). Signals written to the same file (a multiplexed signal file) must be consecutive and have the same `fname` and `group`, and the same format and block size; each new file has the next group number, starting at 0. A `fname` of `-` is the standard output; a signal of format 0 with no file name (or `~`) is not written, and a signal of format 0 with a file name is written in format 16. An `spf` smaller than 1 is taken as 1.

`osigfopen(NULL, 0)` closes the output signals (and the header opened by `newheader`, `setheader` or `setmsheader`) and reports whether they were written successfully.

siarray
An array of `nsig` `WFDB_Siginfo` structures, filled by the caller.

nsig
The number of output signals.

Returns The number of output signals opened (`nsig`); -2 for an error in a specification (`fname` and `desc` together longer than 200 characters, a negative `bsize`, a format not in `WFDB_FMT_LIST`, a wrong group or file name, or signals of a group with different formats or block sizes); -3 if a signal file cannot be created. With `nsig` 0: 0, or -4 if a signal file or header closed by the call could not be written completely.

Errors `osigfopen: error in specification of signal N`; `osigfopen: incorrect file name or group for signal N`; `osigfopen: can't open FILE`; `osigclose: write error in signal group N`.

WFDB-x: as for `osigopen`, files are created under the output directory, and a signal file of the open input record is refused (`osigfopen: output file FILE is also an input file`); see Chapter 4.

---

wfdbinit — Open the annotation files and the input signals

```
int wfdbinit(char *record, const WFDB_Anninfo *aiarray, unsigned int nann,
             WFDB_Siginfo *siarray, unsigned int nsig);
```

Calls `annopen(record, aiarray, nann)` and, if it returns 0, `isigopen(record, siarray, nsig)`. The Programmer's Guide advises against using `wfdbinit` together with `setifreq`: `setifreq` must be called after the signals are opened and before the annotation files are, to apply to the annotation times.

record
The record name.

aiarray
The annotators, as for `annopen`.

nann
The number of annotators.

siarray
An array of at least `nsig` `WFDB_Siginfo` structures, as for `isigopen`.

nsig
The maximum number of input signals to open.

Returns The negative result of `annopen` if it fails; otherwise the result of `isigopen`: the number of signals opened (with `nsig` 0, the number of signals of the record, none being opened), 0 if none (no message if `nsig` is 0), or -1, -2, -3.

In R, `wfdbinit(record, aiarray, nsig)` has no `nann` argument, and returns `list(stat, si)`; in MATLAB/Octave, `[stat, si] = wfdbx.wfdbinit(record, aiarray, nann, nsig)`.

---

wfdbquit — Close all files and reset the library

```
void wfdbquit(void);
```

Closes all the annotation files (completing and, if needed, sorting the output files, as `oannclose`), the info file opened by `setinfo`, the input and output signal files (writing out the buffered samples) and the header being written, and releases the buffers of `sample` and the info strings of `getinfo`. It resets the sampling frequencies, the base time and date, the record length and the segments of the open record. It does not write a header (`newheader` or `setheader` must be called first), and, as in C, it does not reset the counter frequency and base counter value, the output annotation time resolution (`setafreq`), the buffer sizes (`setibsize`, `setobsize`), the calibration list, or the tables of mnemonics, descriptions and code mappings (Chapter 4). Programs that write signals or annotations must call it (or the closing functions of each kind of file) before they exit.

WFDB-x: the input directory, the output directory and the network setting are kept; in legacy mode, the WFDB path is restored as by `resetwfdb` (Chapter 4).

---

wfdbflush — Write the buffered output

```
void wfdbflush(void);
```

Writes the buffered output annotations and samples to their files, so that the files are up to date while they stay open.

13.3 Reading and writing signals

The input signals are read one sample vector at a time, sequentially, with `getvec` (one sample of each signal) or `getframe` (every sample of a frame); `isigsettime`, `isgsettime` and `tnextvec` move to another time, and `sample` gives random access. Samples are integers in ADC units; a missing or invalid sample is `WFDB_INVALID_SAMPLE` (-32768) unless the `WFDB_GVPAD` mode is set. Times are in sample intervals: of the frame, of the highest sampling frequency in `WFDB_HIGHRES` mode, or of the frequency set by `setifreq`.

---

getvec — Read a sample of each input signal

```
int getvec(WFDB_Sample *vector);
```

Reads the next sample of each input signal into `vector`: `vector[i]` receives the sample of signal `i`. In `WFDB_LOWRES` mode (the default), a signal with several samples per frame gives the mean of the samples of the frame (rounded to the nearest integer, halves upward; invalid if one of them is); in `WFDB_HIGHRES` mode, each call gives the next sample at the highest sampling frequency, and the other signals are repeated (see `setgvmode`). After `setifreq`, the signals are resampled at the chosen frequency by linear interpolation. Skewed signals are corrected for their skew. The checksum of a signal is verified when the number of samples given in the header has been read, unless the signals were positioned (`isigsettime`, `isgsettime`) before (it is not verified in multi-segment records, EDF files and signals of format 0).

vector
: An array with room for one sample per input signal, filled by the function.

Returns The number of input signals, on success; 0 if no input signals are open; -1 at the end of the data (the contents of `vector` are then not valid); -3 at an unexpected end of a signal file (or a FLAC decoding error); -4 on a checksum error, detected on the last frame (which is valid, unless a signal file ended in that frame: as in C, -4 then replaces -3).

Errors `getvec: unexpected EOF in signal N`; `getvec: checksum error in signal N`; `getvec: error opening segment SEG`.

---

getframe

Read a frame of the input signals

```
int getframe(WFDB_Sample *vector);
```

Reads the next frame: every sample of every input signal for one frame interval, the `spf` samples of signal 0 first, then those of signal 1, and so on. For a record whose signals all have one sample per frame, `getframe` and `getvec` (without `setifreq`) read the same vectors. The `WFDB_HIGHRES` mode and `setifreq` do not affect it (`WFDB_GVPAD` does, see `setgvmode`).

vector
An array with room for the sum of the `spf` fields of the input signals, filled by the function.

Returns As `getvec`.

---

putvec

Write a sample to each output signal

```
int putvec(const WFDB_Sample *vector);
```

Writes a frame to the output signals: `vector` holds the `spf` samples of output signal 0, then those of signal 1, and so on (one sample per signal when every `spf` is 1). The number of samples and the checksum of each signal are kept, for `newheader`. A sample equal to `WFDB_INVALID_SAMPLE` is written as the invalid-sample value of the format; a sample outside the range of the format is written as C writes it (its low-order bits). In format 8 (first differences), a difference outside -128 to 127 is limited to that range, and the following samples correct the level as soon as the differences allow.

vector
The samples, given by the caller.

Returns The number of output signals on success; 0 if a difference was too large in format 8; -1 on a write error.

Errors `putvec: write error in signal N`.

---

isigsettime

Skip to a time in all input signals

```
int isigsettime(WFDB_Time t);
```

Positions the input signals so that the next sample read is the one at time `t`; only the magnitude of `t` counts, so that a value returned by `strtim` for a time of day can be given as it is. Nothing is done if the signals are already at `t` (without `setifreq`). A pipe, such as the standard input, cannot be positioned backward.

t The time, in sample intervals (see the introduction of this section).

Returns 0 on success (also if no input signals are open); -1 at the end of the data or on an improper seek; -2 if a signal group is incorrect.

Errors `isigsettime: improper seek on signal group N`.

---

isgsettime

Skip to a time in one signal group

```
int isgsettime(WFDB_Group sgroup, WFDB_Time t);
```

As `isigsettime`, for the signals of group `sgroup` only (the signals stored in one file); the other groups keep their position. It serves programs that open several records at once (`isigopen("+REC", ...)`).

sgroup
: The signal group number.

t
: The time, in sample intervals.

Returns 0 on success; -1 at the end of the data or on an improper seek; -2 if `sgroup` is not a signal group.

Errors `isigsettime: improper seek on signal group N`; `isgsettime: incorrect signal group number N`.

WFDB-x: a group number beyond the groups of the open record is reported as incorrect (-2) also when the signals are already at `t`, where C reads past the end of its array (Chapter 4).

---

tnextvec

Find the next valid sample of a signal

```
WFDB_Time tnextvec(WFDB_Signal s, WFDB_Time t);
```

Finds the first time, at or after `t`, at which input signal `s` has a valid sample (not `WFDB_INVALID_SAMPLE`), and positions the input signals there, so that the next `getvec` reads it. It skips long gaps (in a multi-segment record, segments without the signal) efficiently.

s
: The input signal number.

t
: The time at which the search begins, in sample intervals.

Returns The time of the next valid sample (`>= 0`); -1 at the end of the data, on an improper seek, or if `s` is not an input signal (or the negative status of `getvec`).

Errors `nextvect: illegal signal number N` (the C library's spelling).

---

sample

Read a sample by its time

```
WFDB_Sample sample(WFDB_Signal s, WFDB_Time t);
```

Returns sample `t` of input signal `s`, in any order: the samples are read with `getvec` (which moves the input signals) into a buffer of the 4096 most recent sample vectors, from which they are returned. A negative `t` is taken as 0. `sample_valid` tells whether the value is valid. `wfdbquit` releases the buffer. In R, this function is `wfdb_sample`.

s The input signal number (0 to the number of input signals minus 1).

t The sample number.

Returns The sample, in ADC units; after the end of the record, the last sample read; `WFDB_INVALID_SAMPLE` if the signal has no valid sample at `t`, if `s` is not an input signal, or if the signals cannot be positioned.

---

sample_valid

Whether the last sample read by sample is valid

```
int sample_valid(void);
```

Tells whether the value most recently returned by `sample` is valid, so that `while (sample_valid())` reads to the end of the record and `sample_valid() > 0` selects the values usable in a computation.

Returns 1 if the value was valid; 0 if `t` was after the end of the record (the value is padding) or the signals could not be positioned; -1 if the value was `WFDB_INVALID_SAMPLE` (the signal is not available at `t`, or the signal number is not valid).

---

findsig

Find an input signal by name or number

```
int findsig(const char *signame);
```

Returns the number of the input signal designated by `signame`: a string of decimal digits that is the number of an open input signal is taken as that number; otherwise the lowest-numbered input signal whose description (`desc`) is exactly `signame` is returned. The descriptions are compared as the header gives them, spaces included (the high-level readers remove surrounding spaces from signal names; the library does not).

signame
A signal number or a signal description, such as `"MLII"`.

Returns The signal number, or -1 if there is no such signal.

---

getspf

Number of samples per signal per frame

```
int getspf(void);
```

Returns the number of sample vectors that `getvec` returns per frame: 1, except in `WFDB_HIGHRES` mode for a record with oversampled signals, where it is the largest number of samples per frame of the input signals (the sampling frequency divided by the frame frequency, rounded).

Returns The number of samples per signal per frame.

---

setgvmode

Set the mode of getvec

```
void setgvmode(int mode);
```

Sets how `getvec` reads records whose signals have different numbers of samples per frame, and how invalid samples are returned. `WFDB_LOWRES` (0, the default): one sample per signal per frame, the oversampled signals being averaged over the frame. `WFDB_HIGHRES` (1): every sample of the most oversampled signal, the other signals being repeated (zero-order hold); the sampling frequency returned by `sampfreq(NULL)`, and the times of the time conversion functions and of the annotations, are then in units of the high-resolution sample interval (for annotation files opened after the call). `WFDB_GVPAD` (2), added to either: an invalid or missing sample is replaced by the previous valid sample of its signal, in `getvec`, `getframe` and `sample`. A negative `mode` restores the default, read from the environment variable `WFDBGVMODE` (or `WFDB_LOWRES` if it is not set).

mode
: `WFDB_LOWRES` or `WFDB_HIGHRES`, optionally plus `WFDB_GVPAD`; or a negative value.

---

getgvmode — The mode of getvec

```
int getgvmode(void);
```

Returns the mode set by `setgvmode`: the `WFDB_HIGHRES` bit is set in high-resolution mode, the `WFDB_GVPAD` bit if invalid samples are replaced.

Returns The mode (0 to 3).

---

setifreq — Resample the input signals

```
int setifreq(WFDB_Frequency freq);
```

Makes `getvec` return the input signals resampled at `freq` samples per second, by linear interpolation between the two nearest samples (`getframe` is not affected). It must be called after the signals are opened. From then on, the time conversion functions, `isigsettime` and `isgsettime` use sample intervals at `freq`, the annotators opened afterwards return times at `freq`, and the time resolution of the output annotation files opened afterwards is `freq` (it calls `setafreq(freq)`). The resampling uses the greatest common divisor of the two frequencies, treated as multiples of 0.005 Hz.

freq
: The sampling frequency wanted, in Hz (> 0).

Returns 0 on success; -1 if no record is open, if `freq` is not positive, or (WFDB-x) if the sampling frequency of the open record is not specified.

Errors `setifreq: no open input record`; `setifreq: improper frequency F (must be > 0)`; `setifreq: sampling frequency is not specified`.

WFDB-x: a record without a sampling frequency is an error (Chapter 4); in legacy mode, it is reported as `setifreq: no open input record`, as in C.

---

getifreq — The frequency of the samples returned by getvec

```
WFDB_Frequency getifreq(void);
```

Returns the sampling frequency of the samples returned by `getvec`: the one set by `setifreq`, or else that of the open record (as `sampfreq(NULL)`).

Returns The frequency, in Hz (0 if no record is open, or if its frequency is not specified).

`setibsize` Set the size of the input buffers

```
int setibsize(int input_buffer_size);
```

Sets the size of the buffers in which the input signal files are read, for the signals opened afterwards (a block size given in the header takes precedence). It cannot be used while input signals are open.

`input_buffer_size`
The size in bytes; 0 for the default (`BUFSIZ`, 8192 bytes).

Returns The new size; -1 if input signals are open; -2 if the size is negative.

Errors `setibsize: can't change buffer size after isigopen`; `setibsize: illegal buffer size N`.

`setobsize` Set the size of the output buffers

```
int setobsize(int output_buffer_size);
```

As `setibsize`, for the output signal files opened afterwards by `osigopen` and `osigfopen`; it cannot be used while output signals are open.

`output_buffer_size`
The size in bytes; 0 for the default (`BUFSIZ`, 8192 bytes).

Returns The new size; -1 if output signals are open; -2 if the size is negative.

Errors `setobsize: can't change buffer size after osig[f]open`; `setobsize: illegal buffer size N`.

13.4 Time and date conversion

Times are kept as `WFDB_Time` values, in sample intervals. The functions of this section convert them to and from text, using the sampling frequency (or the frequency set by `setifreq`), the base time and date and the counter frequency of the open record, set by `isigopen`, `wfdbinit` or `sampfreq` from a header, or by `setsampfreq`, `setbasetime` and `setcfreq`. A positive time is an elapsed time (an interval from the beginning of the record); a time of day is represented by the negated number of sample intervals from the beginning of the record (0 or negative).

WFDB-x (Chapter 4): when no sampling frequency is known, C took a sample interval to be one second; WFDB-x reports `FUNCTION: sampling frequency is not specified`, and `timstr` and `mstimstr` return the time as a sample number (`s` followed by its magnitude), `strtim` returns 0 for a string that needs the frequency (`wfdbx_time_needs_freq` tells which). Legacy mode keeps C's behavior.

`timstr` Format a time

```
char *timstr(WFDB_Time t);
```

Converts `t` to text. If `t` is positive, or if neither a base time nor a base date is defined, `|t|` is an elapsed time, written `h:mm:ss` right-aligned in 8 characters (more from 100 hours on), without the hours under one hour (`" 1:00"`, `" 1:02:05"`). Otherwise `-t` sample intervals after the base time are written as a time of day in brackets, with all the digits, followed by the date if a base date is defined, or else by the number of days after the base time if it is not 0: `"[12:00:01 01/01/2020]"`, `"[12:00:00 1]"`. The time is rounded to the millisecond, and the milliseconds are then dropped. A base time of midnight (0) without a base date counts as undefined.

t The time, in sample intervals.

Returns The text. In C, it is in a static buffer (shared with `mstimstr`) that the next call overwrites; in WFDB-x, it is a new string in every language.

Errors WFDB-x: `timstr: sampling frequency is not specified`, and the result `s` followed by `|t|` (`"s100"`), if the frequency is not known.

---

mstimstr

Format a time with milliseconds

```
char *mstimstr(WFDB_Time t);
```

As `timstr`, with the milliseconds (rounded): an elapsed time `h:mm:ss.mmm` right-aligned in 12 characters (`" 0:01.000"`), a time of day `"[hh:mm:ss.mmm dd/mm/yyyy]"`. `mstimstr(0)` gives the base time and date of the record (there is no `getbasetime`).

t The time, in sample intervals.

Returns The text, as for `timstr`.

Errors WFDB-x: `mstimstr: sampling frequency is not specified`, as for `timstr`.

---

strtim

Parse a time

```
WFDB_Time strtim(const char *time_string);
```

Converts a time string to sample intervals. Leading spaces, tabs, carriage returns and newlines are skipped; then, according to the first character:

s*N* sample number *N* (`s12345`);

c*X* counter value *X*, converted with the counter frequency and the base counter value (`c350.5`); without a counter frequency, *X* is a sample number;

f*N* frame number *N*, converted to sample intervals;

e the end of the record (its length; 0 if the header does not give it), in high-resolution intervals in `WFDB_HIGHRES` mode;

i the time of the next sample of the input signals;

o the time of the next sample of the output signals;

[*time date*] a time of day, `[hh:mm:ss]`, followed optionally by a date `dd/mm/yyyy` or by a number of days after the base date (`[13:6:0]`, `[8:0:0 1]`, `[12:0:0 1/3/1992]`): the result is the negated

number of sample intervals from the base time to that time (0 or negative); without the closing bracket, the result is 0;

otherwise an elapsed time `h:m:s`, `m:s` or `s`, each part with optional decimals (`2:14.875`, `143`, `4:02:01`), rounded to the nearest sample interval.

For records with sampling frequencies below 1 kHz, `strtim(mstimstr(t))` is `t`.

`time_string`
The time string.

Returns A positive number of sample intervals for an elapsed time, a sample number, a counter value, `e`, `i` or `o`; the negated number of sample intervals for a time of day; 0 for a string that is not a time (indistinguishable from a time of 0).

Errors WFDB-x: `strtim: sampling frequency is not specified`, and 0, if the frequency is not known and the string needs it.

`datstr` — Format a date

```
char *datstr(WFDB_Date d);
```

Converts a Julian date (a day number; these Julian dates begin at midnight local time, not at noon) to text, `" dd/mm/yyyy"`, with a leading space, which the Programmer's Guide does not mention (`datstr(2374674)` is `" 14/07/1789"`). The Gregorian calendar applies from 15 October 1582; years before 1 A.D. are negative (there is no year 0).

`d` The Julian date.

Returns The text (a static buffer in C; a new string in WFDB-x).

`strdat` — Parse a date

```
WFDB_Date strdat(const char *date_string);
```

Converts a date `dd/mm/yyyy` to a Julian date. The year is taken as written: `15/3/89` is in the year 89, not 1989. The number of days between two dates is the difference of their Julian dates.

`date_string`
The date.

Returns The Julian date; 0 if the string does not contain two `/`, the day is not 1 to 31, the month is not 1 to 12, or the year is 0.

`sampfreq` — Sampling frequency of a record

```
WFDB_Frequency sampfreq(char *record);
```

With a record name, reads the header of `record` and returns its sampling frequency; the frequency, base time and date and counter frequency of the header then become those used by the time

conversion functions (as after `isigopen(record, NULL, 0)`, without closing the open signals). With `NULL`, returns the sampling frequency in use. For a record with oversampled signals, the result is the frame frequency in `WFDB_LOWRES` mode and the highest sampling frequency in `WFDB_HIGHRES` mode.

record
The record name, or `NULL`.

Returns The sampling frequency in Hz; -1 if the header cannot be read; -2 if it is not valid; WFDB-x: 0 if the header does not give a frequency (C: 250, `WFDB_DEFFREQ`; see Chapter 4). `sampfreq(record)` first sets the frequency to 0, so that a frequency set by `setsampfreq` does not apply to it.

---

setsampfreq — Set the sampling frequency

```
int setsampfreq(WFDB_Frequency sampling_frequency);
```

Sets the frame frequency used by the time conversion functions and written by `newheader` and `setheader`: it is the sampling frequency of a record whose signals have one sample per frame (in `WFDB_HIGHRES` mode, the sampling frequency becomes the frame frequency times the largest number of samples per frame). A header read afterwards by `isigopen`, `osigopen` or `wfdbinit` replaces it, unless the header gives no frequency: `setsampfreq(250)` before opening a record whose header gives none reproduces C's default (Chapter 4).

sampling_frequency
The frame frequency, in Hz (0 or more).

Returns 0 on success; -1 if the frequency is negative.

Errors `setsampfreq: sampling frequency must not be negative.`

---

setbasetime — Set the base time and date

```
int setbasetime(char *time_string);
```

Sets the base time (the time of day of sample 0) and the base date, used by the time conversion functions and written by `newheader` and `setheader`. The string is a time `hh:mm:ss` (with optional decimals), optionally followed by a space and a date `dd/mm/yyyy`; leading spaces are skipped. Only a space separates the date (not a tab, which the Programmer's Guide also allows). An empty string or `NULL` sets the current local time and date.

time_string
The time and date, or `""` or `NULL`.

Returns 0 on success; -1 if neither a time nor a date can be read from the string (and it does not begin with `[`).

Errors `setbasetime: incorrect time format, 'STRING'.`

---

getcfreq — Counter frequency

```
WFDB_Frequency getcfreq(void);
```

Returns the counter frequency: the rate at which a counter (of a tape recorder, or the pages of a chart) changes, as given by the header or by `setcfreq`, which `strtim` uses for `c` times. If none is defined (0 or negative), it returns the frame frequency.

Returns The counter frequency, in Hz.

---

setcfreq — Set the counter frequency

```
void setcfreq(WFDB_Frequency counter_frequency);
```

Sets the counter frequency, used by `strtim` and written by `newheader` and `setheader`; 0 or a negative value means none. A header read afterwards replaces it. `wfdbquit` does not reset it.

counter_frequency
The counter frequency, in Hz.

---

getbasecount — Base counter value

```
double getbasecount(void);
```

Returns the counter value of sample 0, as given by the header or by `setbasecount` (0 if neither gave one).

Returns The base counter value.

---

setbasecount — Set the base counter value

```
void setbasecount(double count);
```

Sets the counter value of sample 0, used by `strtim` and written by `newheader` and `setheader`. A header read afterwards replaces it. `wfdbquit` does not reset it.

count
The base counter value.

13.5 Amplitude conversion

These functions convert between ADC units and physical units with the gain (and, for `aduphys` and `physadu`, the baseline) of an open input signal. `adumuv` and `muvadu` convert differences of level (the baseline is not used), between ADC units and thousandths of the physical unit (microvolts for a signal in millivolts).

WFDB-x (Chapter 4): for an uncalibrated signal (gain 0) or a number that is not that of an open input signal, C assumed a gain of 200 (`WFDB_DEFGAIN`) and a baseline of 0; WFDB-x reports `FUNCTION: signal N is uncalibrated` or `FUNCTION: invalid signal number N`, and returns `WFDB_INVALID_SAMPLE` (`aduphys`: NaN). Legacy mode keeps C's behavior.

aduphys

ADC units to physical units

```
double aduphys(WFDB_Signal s, WFDB_Sample a);
```

Returns `(a - baseline) / gain` for input signal `s`.

s The input signal number.
a The sample value, in ADC units.

Returns The value in physical units; WFDB-x: NaN on error.

physadu

Physical units to ADC units

```
WFDB_Sample physadu(WFDB_Signal s, double v);
```

Returns `v * gain`, rounded to the nearest integer (halves away from zero), plus the baseline of input signal `s`.

s The input signal number.
v The value, in physical units.

Returns The sample value in ADC units; WFDB-x: `WFDB_INVALID_SAMPLE` on error.

adumuv

ADC units to microvolts

```
int adumuv(WFDB_Signal s, WFDB_Sample a);
```

Returns `a * 1000 / gain` for input signal `s`, rounded to the nearest integer (halves away from zero).

s The input signal number.
a The difference of level, in ADC units.

Returns The difference in thousandths of the physical unit; WFDB-x: `WFDB_INVALID_SAMPLE` on error.

muvadu

Microvolts to ADC units

```
WFDB_Sample muvadu(WFDB_Signal s, int microvolts);
```

Returns `microvolts * gain / 1000` for input signal `s`, rounded to the nearest integer (halves away from zero).

s The input signal number.
microvolts
The difference of level, in thousandths of the physical unit.

Returns The difference in ADC units; WFDB-x: `WFDB_INVALID_SAMPLE` on error.

13.6 Annotations

Annotation files are opened by annopen (or wfdbinit), read sequentially with getann, and written with putann, in MIT format (WFDB_READ, WFDB_WRITE) or AHA format (WFDB_AHA_READ, WFDB_AHA_WRITE); Chapter 15 describes both formats and lists the annotation codes. An annotation is a WFDB_Annotation structure (Section 13.12).

---

getann Read the next annotation

```
int getann(WFDB_Annotator a, WFDB_Annotation *annot);
```

Reads the next annotation of input annotator a into annot (an annotation pushed back by ungetann first, if there is one). Its time is in sample intervals of the input sampling frequency in effect when the annotator was opened (that of setifreq, or of the record, in high-resolution intervals in WFDB_HIGHRES mode), converted from the time resolution recorded in the file if it records one, or in the units set by setiafreq. In C, aux points to a buffer of the annotator that later calls overwrite.

a The input annotator number (0 for the first input annotator of annopen).

annot
The annotation structure, filled by the function.

Returns 0 on success; -1 at the end of the file (*annot is not valid); -2 if a is not an open input annotator; -3 at an unexpected end of the file.

Errors getann: can't read annotator N; getann: unexpected EOF in annotator ATR; getann: unexpected annot number in annotator ATR.

---

ungetann Push an annotation back

```
int ungetann(WFDB_Annotator a, const WFDB_Annotation *annot);
```

Makes *annot the next annotation that getann returns for input annotator a. It need not have been read from that annotator. Only one annotation can be pushed back at a time for each annotator.

a The input annotator number.

annot
The annotation to push back.

Returns 0 on success; -1 if an annotation is already pushed back (this one is not); -2 if a is not an input annotator.

Errors ungetann: annotator N is not initialized; ungetann: pushback buffer is full.

---

putann Write an annotation

```
int putann(WFDB_Annotator a, const WFDB_Annotation *annot);
```

Writes *annot to output annotator a; all its fields must be set. Annotations may be written in any order: an annotation is in order if its time is later than that of the previous annotation, or

its chan or its num is greater, or both are at time 0; a file that received one out of order is sorted when it is closed (see oannclose). The first putann to a file writes, before the annotation, the time resolution set by setafreq and the modification labels of the mnemonics and descriptions changed by setannstr and setanndesc. In an AHA file, at most 6 bytes of aux are written. As in C, the annotation code is not checked: an annotation written with one of the codes 59 to 63, the control codes of the MIT format (Section 15.3.1), is read back as a control word, which corrupts the rest of the file, and a code above 63 loses its high bits; in an MIT file, every time other than WFDB_TIME_MIN and WFDB_TIME_MAX is written, with as many SKIP words as its distance from the previous annotation needs. The high-level writers (write_annotations) refuse these codes and times before they write anything (Section 4.8).

a The output annotator number (0 for the first output annotator of annopen).

annot
The annotation, filled by the caller.

Returns 0 on success; -1 on a write error, or (MIT format) for a time equal to WFDB_TIME_MIN or WFDB_TIME_MAX; -2 if a is not an open output annotator.

Errors `putann: can't write annotation file N`; `putann: write error on annotation file ATR`; `putann: time overflow in annotation file N`.

---

iannsettime — Skip to a time in all input annotators

```
int iannsettime(WFDB_Time t);
```

Positions every input annotator at its first annotation at or after time |t| (rewinding the file if needed, which a pipe cannot do), so that a value of strtim for a time of day can be given as it is. WFDB_TIME_MIN rewinds to the very beginning, before annotations at negative times. A pushed-back annotation is discarded if the file is rewound.

t The time, in the units of the annotators' times.

Returns 0 if at least one annotator is positioned on an annotation; -1 if every annotator reached the end of its file (or none is open), or on an improper seek. (The Programmer's Guide also lists -3, which the C code never returns.)

Errors `iannsettime: improper seek`.

---

ianndclose — Close an input annotator

```
void ianndclose(WFDB_Annotator a);
```

Closes the file of input annotator a. The input annotators that follow are renumbered: annotator a+1 becomes a, and so on. Nothing is done if a is not open.

a The input annotator number.

---

oannclose — Close an output annotator

```
void oannclose(WFDB_Annotator a);
```

Completes and closes the file of output annotator `a` (an end marker in an MIT file, padding to a 1024-byte block in an AHA file); the output annotators that follow are renumbered. If annotations were written out of order, the file is sorted, unless the environment variable `WFDBANNSORT` is 0: the result is that of `sortann -r. REC -a ATR`, with the messages `Rearranging annotations for output annotator ATR ...done!`; if it is not sorted, the advice `Use the command: sortann -r REC -a ATR` is printed. The status is kept for `annopen(..., 0)`: -7 after a write error, -6 after an out-of-order file.

a The output annotator number.

Errors `oannclose: write error on annotation file ATR`; `Annotations still need to be rearranged.`

WFDB-x (Chapter 4): the file is sorted in the same process (C ran the program `sortann` through the shell), and an annotation at time 0, number 0 and channel 0 that arrives after later ones is no longer lost, nor are those written after it; legacy mode runs the `sortann` tool of the same language, with C's result (without a shell; R and MATLAB/Octave quote its arguments instead). A write error is counted (`wfdbx_write_errors`). The high-level `write_annotations` can decide for its call whether the file is sorted.

---

setafreq

Set the time resolution of output annotation files

```
void setafreq(WFDB_Frequency f);
```

Sets the time resolution, in ticks per second, of the output annotation files to which no annotation has been written yet (contrary to the Programmer's Guide, this includes those already open): the resolution is recorded at the beginning of the file, and the times given to `putann` are in ticks. 0 (the default) records none: the times are then in sample intervals of the record. `setifreq` calls it. `wfdbquit` does not reset it.

f The time resolution, in Hz; 0 for none.

---

getafreq

Time resolution of output annotation files

```
WFDB_Frequency getafreq(void);
```

Returns the value set by `setafreq` (or by `setifreq`).

Returns The time resolution in Hz; 0 if none was set.

---

setiafreq

Set the time units of an input annotator

```
void setiafreq(WFDB_Annotator a, WFDB_Frequency f);
```

Makes `getann` return the times of input annotator `a` in units of `1/f` second (the times read and pushed back so far are converted too). `annopen` sets it to the input sampling frequency. With `f` 0 or negative, the times are those of the file, unconverted. Nothing is done if `a` is not an open input annotator.

a The input annotator number.

f The frequency, in Hz.

---

getiafreq

Time units of an input annotator

```
WFDB_Frequency getiafreq(WFDB_Annotator a);
```

Returns the frequency of the time units that `getann` uses for input annotator `a` (see `setiafreq`).

Returns The frequency in Hz; -2 if `a` is not an open input annotator.

---

getiaorigfreq

Time resolution recorded in an annotation file

```
WFDB_Frequency getiaorigfreq(WFDB_Annotator a);
```

Returns the time resolution recorded in the file of input annotator `a` by the program that wrote it (see `setafreq`).

Returns The time resolution in Hz; 0 if the file records none (its times are then in sample intervals of the record's frames); -2 if `a` is not an open input annotator.

---

annstr

Mnemonic of an annotation code

```
char *annstr(int annotation_code);
```

Returns the mnemonic of an annotation code (`"N"` for `NORMAL`), from the table that `setannstr` and the modification labels of the input annotation files change (Chapter 15 lists the standard mnemonics). The codes of the range 0 to `ACMAX` that are not defined have mnemonics such as `"[15]"`.

annotation_code
The annotation code.

Returns The mnemonic; for a code outside 0 to `ACMAX`, the code in brackets (`"[55]"`).

---

strann

Annotation code of a mnemonic

```
int strann(const char *annotation_mnemonic_string);
```

Returns the code (1 to `ACMAX`) whose mnemonic, in the table of `annstr`, is `annotation_mnemonic_string`; if several codes have it, the lowest.

annotation_mnemonic_string
The mnemonic.

Returns The annotation code; `NOTQRS` (0) if no code has that mnemonic.

---

setannstr

Change the mnemonic of an annotation code

```
int setannstr(int annotation_code, const char *annotation_mnemonic_string);
```

Sets the mnemonic of a code in the table of `annstr` and `strann` (`NULL` is taken as `""`). With a positive code, the change is also marked to be recorded, as a modification label, in the output annotation files to which no annotation has been written yet (the labels are written by the first `putann`), so that other programs read the files with these mnemonics. With a negative code, the mnemonic of `-code` is changed for this program only. `wfdbquit` does not reset the table.

annotation_code
The code (or its negation).

annotation_mnemonic_string
The new mnemonic.

Returns 0 on success; -1 if `|code|` is greater than `ACMAX`.

Errors `setannstr: illegal annotation code N`.

---

anndesc — Description of an annotation code

```
char *anndesc(int annotation_code);
```

Returns the description of an annotation code (`"Normal beat"` for `NORMAL`), from the table that `setanndesc` and the modification labels of the input annotation files change.

annotation_code
The annotation code.

Returns The description; `NULL` for a code of the range 0 to `ACMAX` that has none; `"illegal annotation code"` for a code outside that range (the Programmer's Guide says `NULL`; the C code returns this string).

---

setanndesc — Change the description of an annotation code

```
int setanndesc(int annotation_code, const char *annotation_description);
```

As `setannstr`, for the table of `anndesc`: a positive code also marks the description to be recorded in the output annotation files to which no annotation has been written yet, a negative code changes the description of `-code` for this program only.

annotation_code
The code (or its negation).

annotation_description
The new description.

Returns 0 on success; -1 if `|code|` is greater than `ACMAX`.

Errors `setanndesc: illegal annotation code N`.

---

ecgstr — Standard mnemonic of an annotation code

```
char *ecgstr(int annotation_code);
```

As annstr, from a separate table that only setecgstr changes (not the modification labels of annotation files), for programs that need a fixed set of mnemonics.

annotation_code
The annotation code.

Returns The mnemonic; the code in brackets for a code outside 0 to ACMAX.

---

strecg Annotation code of a standard mnemonic

```
int strecg(const char *annotation_mnemonic_string);
```

As strann, with the table of ecgstr.

annotation_mnemonic_string
The mnemonic.

Returns The annotation code; NOTQRS (0) if no code has that mnemonic.

---

setecgstr Change a standard mnemonic

```
int setecgstr(int annotation_code, const char *annotation_mnemonic_string);
```

Sets the mnemonic of a code in the table of ecgstr and strecg (NULL is taken as ""). Nothing is recorded in annotation files, and negative codes are illegal.

annotation_code
The code, 0 to ACMAX.

annotation_mnemonic_string
The new mnemonic.

Returns 0 on success; -1 if the code is outside 0 to ACMAX.

Errors setecgstr: illegal annotation code N.

13.6.1 Classification and mapping of annotation codes

<wfdb/ecgmap.h> defines macros that classify annotation codes and map them to smaller sets, with tables that a program can change; the C library provides each of them also as a function, whose name begins with wfdb_ (wfdb_isqrs for isqrs). WFDB-x provides both names in every language, with the same results; the macro names are functions too (in C++, inline functions of <wfdbx/ecgmap.hpp>). A setter stores the new value as a C char, and returns it as stored: setisqrs(2, 300) returns 44. An argument that is not a legal code (1 to ACMAX) changes nothing. In R, these functions accept vectors of codes. isann (and wfdb_isann) returns a logical value in Python, R and MATLAB/Octave (True, TRUE, true), and 1 or 0 elsewhere.

---

isann Whether a code is a legal annotation code

```
int isann(int code);              /* a macro of <wfdb/ecgmap.h> */
```

Tells whether `code` is a legal annotation code, 1 to `ACMAX` (49).

code
: The annotation code.

Returns 1 if it is legal, 0 otherwise.

---

wfdb_isann

Function form of isann

```
int wfdb_isann(int code);
```

The function form of `isann`, with the same result.

code
: The annotation code.

Returns 1 if it is legal, 0 otherwise.

---

isqrs

Whether a code is a beat

```
int isqrs(int code);              /* a macro of <wfdb/ecgmap.h> */
```

Tells whether annotation code `code` denotes a QRS complex (a beat), from a table that `setisqrs` changes. Initially, the beat codes are 1 to 13, 25, 30, 31, 34, 35, 38 and 41.

code
: The annotation code.

Returns The value of the table (initially 1 for a beat, 0 otherwise); 0 for an illegal code.

---

wfdb_isqrs

Function form of isqrs

```
int wfdb_isqrs(int code);
```

The function form of `isqrs`, with the same result.

code
: The annotation code.

Returns As `isqrs`.

---

setisqrs

Change the result of isqrs

```
int setisqrs(int code, int newval);   /* a macro of <wfdb/ecgmap.h> */
```

Sets the value that `isqrs(code)` returns.

code
The annotation code.

newval
The new value.

Returns The value stored (`newval` as a C `char`); 0 for an illegal code.

---

wfdb_setisqrs — Function form of setisqrs

```
int wfdb_setisqrs(int code, int newval);
```

The function form of `setisqrs`, with the same result.

code
The annotation code.

newval
The new value.

Returns As `setisqrs`.

---

map1 — Map a code to the five basic classes

```
int map1(int code);              /* a macro of <wfdb/ecgmap.h> */
```

Maps an annotation code to one of `NOTQRS`, `NORMAL`, `PVC`, `FUSION` and `LEARN`, from a table that `setmap1` changes (for example, the bundle branch block beats map to `NORMAL`, ventricular escape beats to `PVC`).

code
The annotation code.

Returns The mapped code; `NOTQRS` for an illegal code.

---

wfdb_map1 — Function form of map1

```
int wfdb_map1(int code);
```

The function form of `map1`, with the same result.

code
The annotation code.

Returns As `map1`.

---

setmap1 — Change the result of map1

```
int setmap1(int code, int newval);    /* a macro of <wfdb/ecgmap.h> */
```

Sets the value that `map1(code)` returns.

code
The annotation code.

newval
The new value.

Returns The value stored (as a C `char`); `NOTQRS` for an illegal code.

---

wfdb_setmap1

Function form of setmap1

```
int wfdb_setmap1(int code, int newval);
```

The function form of `setmap1`, with the same result.

code
The annotation code.

newval
The new value.

Returns As `setmap1`.

---

map2

Map a code to the six basic classes

```
int map2(int code);              /* a macro of <wfdb/ecgmap.h> */
```

As `map1`, with `SVPB` as a sixth class: the supraventricular premature beats (`ABERR`, `NPC`, `APC`, `SVPB`) map to `SVPB` instead of `NORMAL`. `setmap2` changes its table.

code
The annotation code.

Returns The mapped code; `NOTQRS` for an illegal code.

---

wfdb_map2

Function form of map2

```
int wfdb_map2(int code);
```

The function form of `map2`, with the same result.

code
The annotation code.

Returns As `map2`.

---

setmap2

Change the result of map2

```
int setmap2(int code, int newval);    /* a macro of <wfdb/ecgmap.h> */
```

Sets the value that `map2(code)` returns.

code
The annotation code.

newval
The new value.

Returns The value stored (as a C `char`); `NOTQRS` for an illegal code.

---

wfdb_setmap2

Function form of setmap2

```
int wfdb_setmap2(int code, int newval);
```

The function form of `setmap2`, with the same result.

code
The annotation code.

newval
The new value.

Returns As `setmap2`.

---

ammap

Map an AHA annotation code to an MIT code

```
int ammap(int code);              /* a macro of <wfdb/ecgmap.h> */
```

Maps an AHA annotation code, a character (its character code, such as that of `'V'`), to an MIT annotation code: `'N'` to `NORMAL`, `'V'` to `PVC`, `'F'` to `FUSION`, `'E'` to `VESC`, `'P'` to `PACE`, `'Q'` to `UNKNOWN`, `'R'` to `RONT`, `'O'` to `NOTE`, `'U'` to `NOISE`, `'[` to `VFON`, `']'` to `VFOFF`. The table is fixed.

code
The AHA code.

Returns The MIT code; `NOTQRS` for any other character.

---

wfdb_ammap

Function form of ammap

```
int wfdb_ammap(int code);
```

The function form of `ammap`, with the same result.

code
The AHA code.

Returns As `ammap`.

---

mamap

Map an MIT annotation code to an AHA code

```
int mamap(int code, int subtype);     /* a macro of <wfdb/ecgmap.h> */
```

Maps an MIT annotation code, with its subtype, to an AHA code (a character code), from a fixed table: NOISE maps to 'U' only with subtype -1 (unreadable signals), and otherwise to 'O', as do the codes without an AHA equivalent.

code
The MIT annotation code.

subtype
Its subtype.

Returns The AHA code; 'O' for an illegal code.

---

wfdb_mamap — Function form of mamap

```
int wfdb_mamap(int code, int subtype);
```

The function form of mamap, with the same result.

code
The MIT annotation code.

subtype
Its subtype.

Returns As mamap.

---

annpos — Display position of an annotation code

```
int annpos(int code);             /* a macro of <wfdb/ecgmap.h> */
```

Returns where programs that display annotations place those of code code, from a table that setannpos changes: APSTD, APHIGH, APLOW, APATT, APAHIGH, APALOW or APUNDEF (Table 13.2).

code
The annotation code.

Returns The position code; APUNDEF for an illegal code.

---

wfdb_annpos — Function form of annpos

```
int wfdb_annpos(int code);
```

The function form of annpos, with the same result.

code
The annotation code.

Returns As annpos.

setannpos Change the result of annpos

```
int setannpos(int code, int newval);  /* a macro of <wfdb/ecgmap.h> */
```

Sets the value that `annpos(code)` returns.

code
The annotation code.

newval
The new position code.

Returns The value stored (as a C `char`); `APUNDEF` for an illegal code.

wfdb_setannpos Function form of setannpos

```
int wfdb_setannpos(int code, int newval);
```

The function form of `setannpos`, with the same result.

code
The annotation code.

newval
The new position code.

Returns As `setannpos`.

13.7 Calibration

The calibration functions manage the *calibration list*, kept in memory: for types of signals, the levels and shape of their calibration pulses and their customary plotting scale (`WFDB_Calinfo`, Section 13.12). A calibration file has one entry per line: the signal type (`*` for any), a tab, then the low and high levels (- for an AC-coupled signal's low level; - as the high level, which `newcal` writes for a high level equal to the low level, makes both levels 0), the pulse shape (`square`, `sine`, `sawtooth`, or anything else for an undefined shape), the scale and the units, separated by spaces or tabs; lines that begin with # and incomplete lines are ignored.

calopen Read a calibration file

```
int calopen(const char *calibration_filename);
```

Reads a calibration file into the calibration list, which is emptied first, unless the name begins with +, which is then removed (the entries are appended). `NULL` reads the file named by the environment variable `WFDBCAL`, or else `wfdbcal`; - reads the standard input.

calibration_filename
The name of the calibration file, or `NULL`.

Returns 0 on success; -2 if the file cannot be opened. (The Programmer's Guide also lists -1, for a lack of memory.)

Errors `calopen: can't read calibration file FILE`.

WFDB-x: the file is read only where its name says, under the input directory for a relative name, not searched for in the WFDB path (Chapter 4).

---

getcal

Find the calibration of a signal

```
int getcal(const char *description, const char *units, WFDB_Calinfo *cal);
```

Copies into `*cal` the first entry of the calibration list whose signal type is a prefix of `description` (or equal to it), and whose units are `units`. A `NULL` argument matches every entry.

description
: The signal description (`desc` of a `WFDB_Siginfo`), or `NULL`.

units
: The physical units, or `NULL`.

cal The structure filled by the function.

Returns 0 if an entry matches; -1 otherwise (also if the list is empty).

In R and MATLAB/Octave, `getcal` takes `description` and `units` only, and returns the entry with the status: `list(stat, cal)` in R, `[stat, cal]` in MATLAB/Octave.

---

putcal

Add an entry to the calibration list

```
int putcal(const WFDB_Calinfo *cal);
```

Appends a copy of `*cal` to the end of the calibration list.

cal The entry.

Returns 0. (The Programmer's Guide also lists -1, for a lack of memory.)

---

newcal

Write the calibration list to a file

```
int newcal(const char *calibration_filename);
```

Writes the calibration list, which is not changed, to a new file in the format read by `calopen`, with `undefined` for an undefined pulse shape. The name may contain only letters, digits, `_`, `~`, `-` and `/`.

calibration_filename
: The name of the file.

Returns 0 on success; -1 if the name is illegal, the file cannot be created, or it cannot be written completely.

Errors `newcal: can't create calibration file FILE`; `newcal: write error in calibration file`; `init: illegal character N in calibration file name`.

WFDB-x: the file is created under the output directory; a write error is counted (`wfdbx_write_errors`).

flushcal Empty the calibration list

```
void flushcal(void);
```

Empties the calibration list. `wfdbquit` does not.

13.8 Header and info strings

A header (`.hea` file) describes a record: `newheader` writes it for the signals written with `putvec`, `setheader` from signal specifications, `setmsheader` for a multi-segment record. The header being written stays open, so that `putinfo` can add info strings to it, until another header is written or `osigfopen(NULL, 0)` (or `wfdbquit`) closes it. Info strings are the comment lines of a header that follow the record line, and the lines of an optional `.info` file; Chapter 5 describes the `Key: value` entries that WFDB-x reads and writes in them.

WFDB-x (Chapter 4): headers are created under the output directory; a header written under a path (`out/rec`) names the record, its signal files and its segments relative to its own directory (`rec`, `rec.dat`), and only the record's own name (the last component) must be made of letters, digits, _, - and ~. Legacy mode checks the whole name and writes the names as given.

newheader Write the header of the output signals

```
int newheader(char *record);
```

Writes the header of `record` for the open output signals, as `setheader` does, with the numbers of samples and the checksums that `putvec` computed: it is called after the samples are written, before the output signals are closed. The frequencies and the base time are those set by `setsampfreq`, `setcfreq`, `setbasecount` and `setbasetime`, and the skews and prolog lengths those set by `wfdbsetskew` and `wfdbsetstart`. The record name - writes the header to the standard output. A record name that begins with + is not accepted (the Programmer's Guide says that the + is removed; the code rejects the character).

record
: The record name.

Returns 0 on success; -1 if the record name is illegal or the header cannot be created.

Errors `init: illegal character N in record name`; `newheader: can't create header for record REC`.

setheader Write a header from signal specifications

```
int setheader(char *record, const WFDB_Siginfo *siarray, unsigned int nsig);
```

Writes (or rewrites) the header of `record` for the `nsig` signals described by `siarray`, with the frequencies, base time, skews and prolog lengths as for `newheader`; the length is the `nsamp` of the first signal. It serves to change a header, for example the gain of a signal: the specifications are read with `isigopen(record, siarray, -nsig)`, changed and written back; the info strings must be copied explicitly (`getinfo`, `putinfo`). As in C, and although the argument is declared `const`, the

`units` of each structure are cut at their first blank and the `desc` at their first line break, in the caller's structures (in MATLAB/Octave, the changed structures are returned as a second output value, `[stat, si]`; in R, `setheader` returns the status only, and the caller's lists are not changed).

record
The record name.

siarray
The signal specifications.

nsig
The number of signals.

Returns 0 on success; -1 if the record name is illegal or the header cannot be created.

Errors `init: illegal character N in record name`; `newheader: can't create header for record REC`.

---

setmsheader

Write the header of a multi-segment record

```
int setmsheader(char *record, char **segnames, unsigned int nsegments);
```

Closes the input signals, and writes the header of the multi-segment record `record`, made of the `nsegments` existing records named in `segnames`, in that order. The segments' signal files are neither copied nor changed; the readers of the record (`getvec`, `isigsettime`, ...) go from one segment to the next. The segments must have the same number of signals and the same sampling frequency, and headers that give their length; the record takes the counter frequency, base counter value and base time and date of the first segment.

record
The name of the multi-segment record.

segnames
The names of the segments.

nsegments
The number of segments.

Returns 0 on success; -1 if the record name is illegal, there is no segment, or the header cannot be created; -2 if a segment name is longer than `WFDB_MAXRNL` (50) characters; -3 if the header of a segment cannot be read (or is itself that of a multi-segment record); -4 if a segment's length is not given, or its number of signals or sampling frequency differs from those of the first segment.

Errors `setmsheader: record must contain at least one segment`; `setmsheader: can't read segment SEG header`; `setmsheader: length of segment SEG must be specified`; `setmsheader: incorrect number of signals in segment SEG`; `setmsheader: incorrect sampling frequency in segment SEG`; `setmsheader: can't create header file for record REC`.

---

getseginfo

The segments of the open record

```
int getseginfo(WFDB_Seginfo **segments);
```

Makes `*segments` point to the array of `WFDB_Seginfo` structures (Section 13.12) that describe the segments of the open multi-segment record: the name, the length and the first sample number

of each. A segment named ~ is a gap, in which no signal is available. If no multi-segment record is open, `*segments` is not changed.

segments
The address of a pointer, set by the function.

Returns The number of segments; 0 if the open record is not a multi-segment record.

The form differs between languages: in C++, `getseginfo(const WFDB_Seginfo **)` gives a pointer to the library's table; in Rust, Julia and Kotlin, the argument (`&mut Option<Vec<Seginfo>>`, a `Ref`, a `MutableList<Seginfo>?`) receives a copy, and the number of segments is returned; in Python, R and MATLAB/Octave, `getseginfo()` takes no argument and returns the list (a struct array in MATLAB/Octave) of segments, empty if the record is not a multi-segment record.

---

getinfo

Read the info strings of a record

```
char *getinfo(char *record);
```

With a record name, reads the info strings of `record` (the comment lines of its header after the record line, then the lines of its `.info` file that begin with `#`, if it has one) and returns the first; with `NULL`, returns the next one, reading those of the current record (the last one opened or named) on the first call. An info string is the line without its `#` and its line ending.

record
The record name, or `NULL`.

Returns The info string; `NULL` if there are no more.

Errors `getinfo: caller did not specify record name.`

WFDB-x: the header and the `.info` file are read only where the record name says (Chapter 4). The record line is found as the header parser (`readheader`) finds it: the first line that has a token (separated by spaces, tabs, carriage returns and line feeds) that does not begin with `#`; the info strings of the header are the later lines that begin with `#`, and a header without a record line has none. In legacy mode, the record line is, as in C, the first line that does not begin with `#`, so that an empty or blank first line is taken for it and the comments that follow it before the real record line are returned as info strings (Section 4.6).

---

putinfo

Write an info string

```
int putinfo(const char *info);
```

Writes `info` as an info string (a line `#`*info*, with a CR-LF line ending): to the `.info` file opened by `setinfo`, or else to the header being written (after `newheader`, `setheader` or `setmsheader`). It should not contain a line break; WFDB-x does not limit its length (the Programmer's Guide gives 254 characters).

info
The info string.

Returns 0 on success; -1 if no info file or header is open.

Errors `putinfo: caller has not specified a record name.`

---

setinfo

Open the info file of a record

```
int setinfo(char *record);
```

Closes the info file opened by an earlier `setinfo`, if any, and opens the `.info` file of `record` for writing with `putinfo`, appending to it if it exists. Such a file holds information about the record without changing its header. `setinfo(NULL)` only closes the info file.

record
: The record name, or `NULL`.

Returns 0 on success; -1 if the record name is illegal or the file cannot be opened; with `NULL`, -2 if the info file could not be written completely.

Errors `setinfo: can't create info file for record REC`; `setinfo: write error in info file`.

---

wfdb_freeinfo

Release the info strings

```
void wfdb_freeinfo(void);
```

Releases the info strings read by `getinfo`: the next `getinfo` reads them again, from the first. `wfdbquit` calls it. (The Programmer's Guide says that it also closes the file opened by `setinfo`; the code does not, and neither does WFDB-x: `setinfo(NULL)` and `wfdbquit` close it.)

---

wfdbgetskew

Skew of an input signal

```
int wfdbgetskew(WFDB_Signal s);
```

Returns the skew of input signal `s` recorded in the header, in frames (the header gives it in samples). `getvec` and `getframe` already correct the skew; the function serves programs that rewrite a header and must keep the skews (with `wfdbsetskew`).

s The input signal number.

Returns The skew in frames; 0 if `s` is not an input signal.

---

wfdbsetiskew

Change the skew of an input signal

```
void wfdbsetiskew(WFDB_Signal s, int skew);
```

Changes the skew, in frames, with which input signal `s` is read (a function of the C library that the Programmer's Guide does not document). The new skew is applied only if it is between 0 and the largest skew of the record's header; otherwise nothing changes.

s The input signal number.

skew
: The skew, in frames.

---

wfdbsetskew

Set the skew of an output signal

```
void wfdbsetskew(WFDB_Signal s, int skew);
```

Sets the skew, in frames, that `newheader` and `setheader` write for output signal `s` (in samples). It does not affect the reading of input signals. Nothing is done if `s` is not an output signal.

s
: The output signal number.

skew
: The skew, in frames.

---

wfdbgetstart

Prolog length of an input signal file

```
long wfdbgetstart(WFDB_Signal s);
```

Returns the length of the prolog (the bytes before sample 0) of the signal file of input signal `s`, as the header records it; the readers already skip it. With no input signals open, signal 0 gives the prolog of the first signal of the last header read.

s
: The input signal number.

Returns
: The prolog length in bytes; 0 if `s` is not an input signal.

---

wfdbsetstart

Set the prolog length of an output signal

```
void wfdbsetstart(WFDB_Signal s, long bytes);
```

Sets the prolog length that `newheader` and `setheader` write for the signal file of output signal `s`; it does not affect the reading of input signals. The value also applies to every signal of the next header written by `setheader` whose file has no prolog length of its own (as when no output signals are open).

s
: The output signal number.

bytes
: The prolog length, in bytes.

---

wfdbputprolog

Write a prolog to an output signal file

```
int wfdbputprolog(const char *prolog, long bytes, WFDB_Signal s);
```

Writes the first `bytes` bytes of `prolog` to the signal file of output signal `s` (normally before any sample), and records the number of bytes written with `wfdbsetstart`.

prolog
The bytes of the prolog.

bytes
Their number.

s The output signal number.

Returns 0 on success; -1 if fewer bytes were written.

Errors `wfdbputprolog: only N of M bytes written`.

13.9 Error handling and settings

The library reports an error by its return value and by a message, printed on the standard error output unless `wfdbquiet` was called; the last message is kept for `wfdberror`.

---

wfdbquiet Stop printing error messages

```
void wfdbquiet(void);
```

Stops the printing of the library's messages on the standard error output; they are still kept for `wfdberror`.

---

wfdbverbose Print error messages again

```
void wfdbverbose(void);
```

Restores the printing of the library's messages (the default).

---

wfdberror The last error message

```
char *wfdberror(void);
```

Returns the last message of the library, printed or not.

Returns The message, with its final newline; if there has been none, the library version, `"WFDB library version 10.7.0 (Jun 10 2022).\n"` (the build date of the C release reproduced).

---

wfdb_error Report an error as the library does

```
void wfdb_error(const char *format_string, ...);
```

Formats a message as `printf` does, keeps it for `wfdberror` and prints it on the standard error output unless `wfdbquiet` was called. The library reports its own errors with it; a program may use it for its own messages.

format_string
A `printf` format.

... The values to format.

In Python, `wfdb_error(fmt, *args)`; in Rust and Kotlin, `wfdb_error(msg)` takes a message already formatted (Rust also has `wfdb_error_fmt(fmt, args)`, and Kotlin the method `w.wfdb_error_fmt(fmt, args)`); in Julia, `WFDBX.wfdb_error(fmt, args...)` (not exported). The formatting is that of the GNU C library in every language.

---

wfdbmemerr — Behavior on memory errors

```
void wfdbmemerr(int exit_on_error);
```

In C, chooses whether a failure to allocate memory ends the program (nonzero, the default) or lets the function continue (0). WFDB-x keeps the value, for `wfdb_me_fatal`, but has no other use for it: the languages report memory exhaustion themselves.

exit_on_error
: Nonzero to end the program on a memory error.

---

wfdb_me_fatal — Whether memory errors are fatal

```
int wfdb_me_fatal(void);
```

Returns the value set by `wfdbmemerr`.

Returns The value; 1 by default.

---

wfdbversion — Version of the library

```
const char *wfdbversion(void);
```

Returns the version of the WFDB library that WFDB-x reproduces (the version of WFDB-x itself is that of its packages, 1.2.3).

Returns `"10.7.0"`.

---

wfdbldflags — Linker flags of the C library

```
const char *wfdbldflags(void);
```

In C, returns the linker options with which programs are linked with the library (fixed when it is built). WFDB-x has none.

Returns `""`.

---

wfdbcflags — Compiler flags of the C library

```
const char *wfdbcflags(void);
```

In C, returns the compiler options with which programs that use the library are compiled. WFDB-x has none.

Returns `""`.

---

wfdbdefwfdb Default WFDB path

```
const char *wfdbdefwfdb(void);
```

Returns the default WFDB path of the C library, used in legacy mode when the environment variable `WFDB` is not set (that of a C release installed in `/usr/local`).

Returns `". /usr/local/database http://physionet.org/physiobank/database"`.

---

wfdbdefwfdbcal Default calibration file

```
const char *wfdbdefwfdbcal(void);
```

Returns the name of the calibration file that `calopen(NULL)` reads when the environment variable `WFDBCAL` is not set.

Returns `"wfdbcal"`.

13.10 File and path functions

The C library finds its input files by searching the WFDB path, a list of directories and URLs. WFDB-x finds a file only where its name says, under the input directory for a relative name (Chapters 3 and 4): the functions of this section keep their C names and arguments, and set and return the input directory; in legacy mode, they work on the WFDB path as in C, without network access.

---

setwfdb Set the input directory (the WFDB path)

```
void setwfdb(const char *database_path_string);
```

WFDB-x: sets the input directory. The string is split as a WFDB path is (at spaces, tabs, newlines, `;`, and `:` outside a URL), and must have at most one nonempty component, which becomes the input directory (a directory, a ~ path or a URL); with none (`""`, `";"`, `NULL`), the default is restored (`WFDBX_INPUT_DIR`, or the current directory). With more than one, nothing changes and an error is reported. A directory whose name contains a space must be set with `wfdbx_set_input_dir`.

In legacy mode, as in C: sets the WFDB path, the list of locations searched for input files (`@file` reads it from a file, up to ten levels deep); `NULL` restores the value of the environment variable `WFDB`, or the default (`wfdbdefwfdb`); the value is also exported as the environment variable `WFDB`, for the programs started afterwards; URL components are not used.

database_path_string
: The input directory (legacy mode: the WFDB path), or `NULL`.

Errors `setwfdb: WFDB-x reads from a single input directory; use wfdbx_set_input_dir (or legacy mode for the WFDB search path).`

---

getwfdb — The input directory (the WFDB path)

```
char *getwfdb(void);
```

WFDB-x: returns the input directory. In legacy mode, as in C: returns the WFDB path, read on the first call from the environment variable `WFDB` (or the default), `@file` indirections evaluated. The environment variable `WFDB` is ignored outside legacy mode, and the library says so once (`WFDB-x: the WFDB environment variable is ignored; set WFDBX_INPUT_DIR instead (or use legacy mode)`).

Returns The input directory, or `"."` if none is set (legacy mode: the WFDB path).

---

resetwfdb — Restore the input directory (the WFDB path)

```
void resetwfdb(void);
```

WFDB-x: restores the input directory set by the last `setwfdb`, if any (`wfdbx_set_input_dir` may have changed it since). In legacy mode, as in C: restores the WFDB path to the value that the first `getwfdb` of the process returned, if `getwfdb` was called before `setwfdb`; `wfdbquit` calls it (in legacy mode only, in WFDB-x).

---

wfdbfile — The path of a WFDB file

```
char *wfdbfile(const char *file_type, char *record);
```

Returns the name under which the library opens the file of `record` with suffix `file_type` (`"hea"`, `"atr"`, ...), if it can be opened: `wfdbfile("hea", "100s")` gives `100s.hea`, or the path under the input directory. A file whose name does not contain a record name is given by `file_type` with `record` `NULL`. As in C, a second name, with the suffix cut to 3 characters, is tried. With both arguments `NULL`, it returns the name of the file most recently opened by the library.

file_type
: The file type (suffix), or a file name.

record
: The record name, or `NULL`.

Returns The path name or URL; `NULL` if the file cannot be opened.

WFDB-x: the name is the one location that the name designates (an absolute name or URL as given, a relative name under the input directory), not the first found in the WFDB path; a URL is opened only with network access (Chapter 4).

13.11 WFDB-x additions

These functions are not in the C library. Their names begin with `wfdbx_`, except the reader of many sample vectors; they are the same in the seven languages, and the signatures below are

those of the C++ implementation (`cpp/include/wfdbx/wfdb.hpp`). Chapter 4 explains the settings they control; the environment variables `WFDBX_INPUT_DIR`, `WFDBX_OUTPUT_DIR` and `WFDBX_NETWORK` give their initial values, which a call overrides.

`wfdbx_set_input_dir` — Set the input directory

```
std::optional<std::string> wfdbx_set_input_dir(const char *dir);
```

Sets the input directory: every file named by a relative name (`100s`, `mitdb/100`, `../other/100`) is read from it; an absolute name, a ~ name or a URL is used as given. It may be a local directory, a ~ path, or the `http://` or `https://` URL of a directory, such as `https://physionet.org/files/mitdb/1.0.0/` (which also needs network access, `wfdbx_set_network`). The name is used as it is (it may contain spaces). The setting applies to the library functions, the high-level functions (whose `input_dir` option sets it for the duration of a call) and the tools, also for the files that they read themselves (`wfdbx_input_path`); `wfdbquit` does not change it; legacy mode does not use it (it searches the WFDB path).

`dir` The input directory; `NULL` or `""` restores the default: the value of `WFDBX_INPUT_DIR` if it is set, otherwise the current directory.

Returns The previous input directory (that of `WFDBX_INPUT_DIR` if it was in effect); `NULL` (`std::nullopt`, `None`, `nothing`, ...) if none was set.

`wfdbx_get_input_dir` — The input directory

```
std::optional<std::string> wfdbx_get_input_dir();
```

Returns the input directory set by `wfdbx_set_input_dir`, `setwfdb` or `WFDBX_INPUT_DIR`.

Returns The input directory; `NULL` if none is set (files are then read from the current directory; `getwfdb` returns `"."`).

`wfdbx_set_output_dir` — Set the output directory

```
std::optional<std::string> wfdbx_set_output_dir(const char *dir);
```

Sets the output directory: every file written under a relative name (a signal file, a header, an annotation, info or calibration file) is created in it; missing directories are created. An absolute name or a ~ name is used as given. The names recorded in headers do not change. The setting applies to the library functions, the high-level writers (whose `dir` option sets it for the duration of a call) and the tools, also for the files that they create themselves (`wfdbx_output_path`); `wfdbquit` does not change it. In legacy mode it is a plain prefix of every output name, absolute names included.

`dir` A local directory or a ~ path; `NULL` or `""` restores the default: the value of `WFDBX_OUTPUT_DIR` if it is set, otherwise the current directory. A URL (any name that begins with a URL scheme, such as `https://` or `ftp://`) is refused.

Returns The previous output directory; `NULL` if none was set. For a URL, the current output directory, unchanged.

Errors `wfdbx_set_output_dir: the output directory must be a local directory.`

---

wfdbx_get_output_dir — The output directory

```
std::optional<std::string> wfdbx_get_output_dir();
```

Returns the output directory set by `wfdbx_set_output_dir` or `WFDBX_OUTPUT_DIR`.

Returns The output directory; `NULL` if none is set (files are then written in the current directory).

---

wfdbx_set_network — Allow or forbid network access

```
int wfdbx_set_network(int on);
```

Allows or forbids reading files from `http://` and `https://` URLs; it is forbidden by default. A remote file is read in full the first time it is opened and kept for the rest of the process (a download that fails is not kept: the file is requested again the next time it is opened); a download gives up if it cannot connect, or receives nothing, for 30 seconds. A name with another URL scheme, such as `ftp://`, is refused, whatever this setting (Section 3.4). With network access forbidden, the first attempt to open a URL prints `WFDB-x: network access is disabled; set WFDBX_NETWORK=1 or call wfdbx_set_network(1) to open URL`, and the file cannot be opened. Legacy mode reads no remote files.

on 1 (or any positive value; `True` in Python) to allow, 0 to forbid; a negative value (or `None` in Python) restores the default: allowed if `WFDBX_NETWORK` is a nonzero number.

Returns The previous setting: 1 (allowed) or 0.

---

wfdbx_get_network — Whether network access is allowed

```
int wfdbx_get_network();
```

Returns 1 if network access is allowed, 0 otherwise.

---

wfdbx_set_legacy — Turn legacy mode on or off

```
int wfdbx_set_legacy(int on);
```

Turns legacy mode on or off for the whole library. In legacy mode, every deliberate change of WFDB-x is undone (Chapter 4): the built-in sampling frequency, gain and ADC resolution, the WFDB search path (without network access), sorting by the `sortann` program, output files that may be input files, and C's checks and names in headers. It is off by default, and is meant only to reproduce results of the original package.

on Nonzero to turn legacy mode on, 0 to turn it off.

Returns The previous setting: 1 (on) or 0.

---

wfdbx_get_legacy

Whether legacy mode is on

```
int wfdbx_get_legacy();
```

Returns 1 if legacy mode is on, 0 otherwise.

---

wfdbx_legacy_args

Recognize the –legacy option of a tool

```
int wfdbx_legacy_args(int &argc, char **&argv);
```

The function that every tool calls first: if the first argument after the program name (`argv[1]`) is exactly `--legacy`, it is removed (the program name stays first) and legacy mode is turned on. `--legacy` elsewhere is not recognized. In C++, Rust and Kotlin, the call also arranges the exit status of the program as for the tools: at exit, outside legacy mode, a failure to write the standard output is reported (`PROGRAM: write error on standard output`), and an exit status of 0 becomes 1 after a write error (see `wfdbx_write_errors`).

argc
: The number of arguments, decremented if `--legacy` is removed.

argv
: The arguments, program name first.

Returns 1 if `--legacy` was found (and removed), 0 otherwise.

The argument list is that of each language: in Python, a list (`sys.argv`), changed in place; in Rust and Kotlin, a mutable list of byte strings; in R, a character vector, and the result is `list(argv, legacy)`; in MATLAB/Octave, a cell array, with `[r, argv] = wfdbx.wfdbx_legacy_args(argv)`. In Julia, `wfdbx_legacy_args(args)` takes the arguments *without* the program name (Julia's `ARGS`), and looks at `args[1]`.

---

wfdbx_time_needs_freq

Whether strtim needs the sampling frequency

```
int wfdbx_time_needs_freq(const char *time_string);
```

Tells whether `strtim` needs a sampling frequency to convert `time_string`, so that a program can report a missing frequency in its own terms.

time_string
: The time string.

Returns 0 if the string (after leading spaces, tabs, carriage returns and newlines) begins with `e`, `i`, `o` or `s`, or with `c` while no counter frequency is defined; 1 otherwise.

---

wfdbx_write_errors

Number of write errors

```
int wfdbx_write_errors();
```

Returns the number of write errors since the program started: the output files (signal files, headers, annotation, info and calibration files, and the standard output opened as -) that the library could not write, flush or close, found when they are closed (by `wfdbquit`, `osigfopen(NULL, 0)`, `oannclose`, ...), and those counted by `wfdbx_note_write_error`. The count belongs to the process, not to a library state object (Rust, Julia, Kotlin). The tools exit with status 1 when it is not 0, and the high-level writers raise an error when they meet a write error (Chapter 4).

Returns The number of write errors.

---

wfdbx_note_write_error

Count a write error

```
void wfdbx_note_write_error();
```

Adds one to the count of `wfdbx_write_errors`, for a program that writes a file itself and could not write it. (Rust and Kotlin, and Julia without exporting it, also have `wfdbx_fclose_output(file, pname, name)`, which closes such a file, and reports `PNAME: write error on NAME` and counts it on failure.)

---

wfdbx_input_path

Where a program reads a file itself

```
std::optional<std::string> wfdbx_input_path(const char *name);
```

Returns the path under which a program opens a file that it reads itself, rather than through the library, so that the file is found where `wfdb_open` finds a file of that name (with an empty file type). In normal mode, a relative name is found under the input directory, if one is set (`wfdbx_set_input_dir`), a ~ name is expanded, and an absolute name is used as given; in legacy mode, the name is used as given, as the original programs did. The tools use it for the files named on their command line that they read themselves, such as the `-i` file of `wrsamp` (Section 3.3.2). For example, with the input directory `data`, `wfdbx_input_path("samples.txt")` returns `data/samples.txt`.

name
: A file name. `""` and `"-"` (the standard input, if the program treats it so) are returned as given.

Returns The path to open; `NULL` (`std::nullopt`, `None`, `nothing`, ...) if `name` is `NULL` or a URL, or becomes a URL when the input directory is prefixed to it (in normal mode, any name that begins with a URL scheme, such as `ftp://`): such a file cannot be opened in this way.

---

wfdbx_output_path

Where a program creates a file itself

```
std::optional<std::string> wfdbx_output_path(const char *name);
```

Returns the path under which a program creates a file that it writes itself, rather than through the library, so that the file is created where `wfdb_open` creates a file of that name (with an empty file type). In normal mode, a relative name is created under the output directory, if one is set (`wfdbx_set_output_dir`), a ~ name is expanded, and an absolute name is used as given; in legacy

mode, the output directory, if one is set, is a plain prefix of any name, absolute names included. When the output directory has been prefixed to the name, the directories missing in the resulting path are created, as for the output files of the library. The tools use it for the files that they create themselves, such as the WAV file of `mit2wav` (Section 3.3.2). For example, with the output directory `out`, `wfdbx_output_path("notes.txt")` returns `out/notes.txt`.

`name`
: A file name. `""` and `"-"` (the standard output, if the program treats it so) are returned as given.

Returns The path to create; `NULL` (`std::nullopt`, `None`, `nothing`, ...) if `name` is `NULL` or a URL (in normal mode, any name that begins with a URL scheme), which cannot be written.

`getframes` — Read many sample vectors at once

```
long long getvecs(WFDB_Sample *out, long long n, int *status = nullptr);
// The same function in each language:
//   C++, Rust, Julia, Kotlin:  getvecs(out, n), and in Rust, Julia and Kotlin
//                              getvecs_status(out, n), which also returns status
//   Python:                    frames, status = getframes(n)
//   R:                         r <- getframes(n, minstat = 0)   # r$v, r$stat
//   MATLAB/Octave:             [v, status, last] = wfdbx.getframes(n, minstat)
```

Reads up to `n` sample vectors, with exactly the result of `n` calls of `getvec` that stop at the first negative status, but much faster: whole blocks of the signal files are decoded at once, for every format, for multi-frequency records in both modes of `getvec`, and for multi-segment records; the frames that need special handling (the one at which a checksum is verified, a change of segment, skewed or resampled signals) go through `getvec`. The vectors follow one another in `out`, one value per signal in each (in Python, R and MATLAB/Octave, a matrix with one row per vector and one column per signal).

`out` (C++, Rust, Julia, Kotlin) An array with room for `n` times the number of values of a vector, filled by the function.

`n` The maximum number of vectors to read (`Inf` in R and MATLAB/Octave reads to the end).

`status`
: (C++, optional) Receives the value returned by the last `getvec`: the negative status of the first vector that could not be read (-1 at the end of the data, -3 for an unexpected end of file, ...), or the number of signals if all `n` were read.

`minstat`
: (R, MATLAB/Octave, optional) Stop at the first vector for which `getvec` would return less than `minstat` (as a C loop `while (getvec(v) >= minstat)`); 0 by default.

Returns C++, Rust, Julia, Kotlin: the number of vectors read (`>= 0`), or the negative status of `getvec` if none could be read; `getvecs_status` returns the number and the status. Python: a NumPy array (`int64`, one row per vector) and the status. R: `list(v, stat)`, a matrix and the status. MATLAB/Octave: the matrix, the status, and the vector filled by the `getvec` call that stopped the reading (the last frame of the record for a checksum error, -4).

13.12 Constants and structures

The constants of <wfdb/wfdb.h> have their C names and values in every language (Table 13.2); in MATLAB/Octave they are functions without arguments (wfdbx.WFDB_READ()), and the calibration constants and WFDB_FMT_LIST are not defined there. The annotation codes (NORMAL, PVC, ..., ACMAX) of <wfdb/ecgcodes.h> are listed in Chapter 15.

Table 13.2 The constants of the library.

Constant	Value	Meaning
WFDB_READ	0	stat of an MIT-format input annotation file
WFDB_WRITE	1	stat of an MIT-format output annotation file
WFDB_AHA_READ	2	stat of an AHA-format input annotation file
WFDB_AHA_WRITE	3	stat of an AHA-format output annotation file
WFDB_APPEND	4	mode of an output info file (opened for appending)
WFDB_INVALID_SAMPLE	-32768	value of an invalid or missing sample, in getvec, getframe and sample, unless WFDB_GVPAD is set; written by putvec as the invalid value of the format
WFDB_LOWRES	0	getvec mode: one sample per signal per frame
WFDB_HIGHRES	1	getvec mode: every sample of the oversampled signals
WFDB_GVPAD	2	getvec flag: invalid samples replaced by the previous valid sample
WFDB_DEFFREQ	250.0	C's default sampling frequency (Hz); not used by WFDB-x outside legacy mode
WFDB_DEFGAIN	200.0	C's default gain (adu per physical unit); not used outside legacy mode
WFDB_DEFRES	12	C's default ADC resolution (bits); not used outside legacy mode
WFDB_MAXRNL	50	maximum length of a record name (of a segment name); not exported in R
WFDB_MAXUSL	50	maximum length of a units string
WFDB_MAXDSL	100	maximum length of a desc string
WFDB_FMT_LIST	0, 8, 16, 61, 80, 160, 212, 310, 311, 24, 32, 508, 516, 524	the signal storage formats (Chapter 15)
WFDB_AC_COUPLED	0	caltype: AC-coupled signal
WFDB_DC_COUPLED	1	caltype: DC-coupled signal
WFDB_CAL_SQUARE	2	caltype: square calibration pulse
WFDB_CAL_SINE	4	caltype: sine calibration pulse
WFDB_CAL_SAWTOOTH	6	caltype: sawtooth calibration pulse
WFDB_CAL_UNDEF	8	caltype: undefined pulse shape
APUNDEF, APSTD, APHIGH, APLOW, APATT, APAHIGH, APALOW	0 to 6	display positions (annpos): undefined, standard, above, below, attached to the signal chan, above and below that

The simple types of the C library are integers and floating-point numbers: WFDB_Sample (int, 32 bits) for samples in ADC units, WFDB_Time (64 bits on the reference platform) for times in sample intervals, WFDB_Date (long) for Julian dates, WFDB_Frequency and WFDB_Gain (double), and

WFDB_Group, WFDB_Signal and WFDB_Annotator (unsigned int). Section 13.1 gives the corresponding types of each language (doubles in R and MATLAB/Octave).

The structures have the C field names in every language (Tables 13.3 to 13.7); their names are WFDB_Siginfo, ..., in C++ (also Siginfo, ...), Siginfo, Anninfo, Annotation, Calinfo and Seginfo in Python, Rust, Julia and Kotlin, and lists or structures made by siginfo(), anninfo(), annotation(), calinfo() and seginfo() in R and MATLAB/Octave (wfdbx.siginfo(n), ...). A string field that C may leave NULL is absent in the language's terms (None, std::nullopt, nothing, NULL, [], null).

Table 13.3 WFDB_Siginfo: the specification of a signal.

Field	Meaning
char *fname	the name of the signal file, relative to the header's directory (WFDB-x); - is the standard input or output
char *desc	the description of the signal (MLII), at most WFDB_MAXDSL characters, without line breaks; a header that gives none gives record REC, signal N
char *units	the physical units, at most WFDB_MAXUSL characters, without blanks; NULL if not specified (WFDB-x: never taken to be millivolts)
WFDB_Gain gain	ADC units per physical unit; 0 if uncalibrated
WFDB_Sample initval	the value of sample 0 (used by format 8)
WFDB_Group group	the signal group: the signals of a group are stored in one file, and have consecutive numbers
int fmt	the storage format (WFDB_FMT_LIST, Chapter 15)
int spf	samples per frame (more than 1 for an oversampled signal)
int bsize	the block size in bytes, for character special files (0 for ordinary files)
int adcres	the ADC resolution in bits; 0 if not specified (WFDB-x: not inferred from the format)
int adczero	the ADC output for an input at the center of its range
int baseline	the ADC value of 0 physical units
long nsamp	the number of samples per signal (of frames, for an oversampled signal); 0 if not specified
int cksum	the 16-bit checksum of the samples, verified by getvec at the end of the record

The constructors give C's zero values, except that spf is 1 in Python, R and MATLAB/Octave (0 in C++, Rust, Julia and Kotlin; osigfopen takes an spf below 1 as 1).

Table 13.4 WFDB_Anninfo: an annotator to open.

Field	Meaning
char *name	the annotator name, the suffix of the annotation file (atr is the reference annotation file of a database record); - is the standard input or output
int stat	WFDB_READ, WFDB_WRITE, WFDB_AHA_READ or WFDB_AHA_WRITE

Table 13.5 WFDB_Annotation: an annotation.

Field	Meaning
WFDB_Time time	the time, in sample intervals from the beginning of the record (or in the units of setiafreq and setafreq)

Field	Meaning
`char anntyp`	the annotation code, 1 to `ACMAX` (Chapter 15)
`signed char subtyp`	the subtype (-128 to 127); for noise annotations, the signals affected
`unsigned char chan`	the channel (0 to 255), usually the signal to which the annotation refers
`signed char num`	the annotator number (-128 to 127), free for the program's use
`unsigned char *aux`	`NULL`, or the auxiliary data: its first byte is the number of bytes that follow (at most 255), such as the rhythm of a rhythm change (`N`, or the text of a comment; it may contain any bytes

Table 13.6 `WFDB_Calinfo`: an entry of the calibration list.

Field	Meaning
`double low`	the low level of the calibration pulse, in physical units (0 for an AC-coupled signal)
`double high`	the high level of the pulse (the pulse amplitude for an AC-coupled signal)
`double scale`	the customary plotting scale, in physical units per centimeter
`char *sigtype`	the signal type, matched against the beginning of a signal description (empty: any signal)
`char *units`	the physical units
`int caltype`	`WFDB_AC_COUPLED` or `WFDB_DC_COUPLED`, plus `WFDB_CAL_SQUARE`, `WFDB_CAL_SINE` or `WFDB_CAL_SAWTOOTH`

Table 13.7 `WFDB_Seginfo`: a segment of a multi-segment record.

Field	Meaning
`char recname[WFDB_MAXRNL+1]`	the name of the segment (a record), or ~ for a gap without signals
`WFDB_Time nsamp`	the length of the segment, in samples
`WFDB_Time samp0`	the number of samples before the segment in the multi-segment record (the sample number of its first sample)

CHAPTER 14

The command-line tools

WFDB-x includes the 33 command-line tools of the WFDB Software Package 10.7.0 that read, write and convert records and annotation files. Each of the seven implementations has all of them, with the options, output, messages and exit status of the C tools, so that a script written for the WFDB tools runs unchanged with WFDB-x. The exceptions are the deliberate divergences of Chapter 4: no built-in signal parameters (hence the options `-F`, `-G`, `-R` and `-U` added to some tools), files found only where they are named, annotation files sorted without loss, an output signal file never written over an input file, write errors reported in the exit status, and the record line of a header found by `getinfo` as the header parser finds it (the `Notes` of `wfdbdesc`). The original behavior remains available with `--legacy` (Section 14.2.4).

Every implementation is checked against the C tools by the same conformance suite (1155 cases): the standard output, the standard error, the exit status and the files written are compared byte for byte. The examples of this chapter were produced by the Python implementation, run in a copy of the directory `tests/data` of the repository, which holds the sample record `100s` (the first minute of record 100 of the MIT-BIH Arrhythmia Database) and the other test inputs; every other implementation gives the same output.

14.1 Running the tools

The tools are separate programs in C++ and Rust, and subcommands of a launcher in the other languages, as in Table 14.1. The arguments that follow the tool's name are those of the C tool: `wfdbx rdsamp -r 100s -t 10` (Python, Julia, R, MATLAB/Octave and Kotlin, with the alias `wfdbx`) and `rdsamp -r 100s -t 10` (C++ and Rust) do the same thing. The tool sees its own name as its program name, so its messages begin with `rdsamp:`, as those of the C tool do.

Table 14.1 How each implementation runs a tool (here `rdsamp`).

Language	Command
Python	`wfdbx rdsamp ...`: the command `wfdbx` is installed with the package (`pip install`); without it, `python -m wfdbx.cli rdsamp ...` or `python -m wfdbx.apps.rdsamp ...`. In a program, `wfdbx.apps.rdsamp.main(argv)` runs the tool (`argv` is the whole argument list, the program name first) and returns its exit status.
C++	`rdsamp ...`: one program per tool, built in `cpp/build/bin/` (or the `bin` directory of the installation).
Rust	`rdsamp ...`: one program per tool, installed by `cargo install --path rust`.
Julia	`julia --startup-file=no --project=JULIADIR JULIADIR/bin/wfdbx.jl rdsamp ...`, where `JULIADIR` is the directory `julia` of the repository; a shell script named `wfdbx` that runs this command makes it `wfdbx rdsamp ...`. In a session, `WFDBX.run_tool("rdsamp", ["-r", "100s"]; dir=...)` returns the exit status and the bytes written to the standard output and the standard error, and `WFDBX.tools()` the names of the tools.

Language	Command
R	`Rscript r/inst/bin/wfdbx.R rdsamp ...` (in an installed package, the script is `system.file("bin", "wfdbx.R", package = "wfdbx")`); with the alias `alias wfdbx='Rscript .../wfdbx.R'`, `wfdbx rdsamp ...`. In a session, `wfdbx_tool("rdsamp", c("-r", "100s"))` runs the tool and returns its exit status, and `wfdbx_tools()` lists the tools.
MATLAB/Octave	Octave: `octave-cli --no-gui --norc -q matlab/bin/wfdbx_cli.m rdsamp ...` (with an alias `wfdbx`, `wfdbx rdsamp ...`); MATLAB: `matlab/bin/wfdbx_matlab.sh rdsamp ...`, which runs `matlab -batch` (or the program named by the environment variable `WFDBX_MATLAB`). In a session, `status = wfdbx.apps.rdsamp({'-r', '100s'});` `help wfdbx.apps.rdsamp` shows the usage.
Kotlin	`kotlin/bin/wfdbx rdsamp ...`, a POSIX shell script that runs `java -cp build/libs/wfdbx-all.jar wfdbx.cli.MainKt rdsamp ...` (the jar is built by `./gradlew fatJar`); a link to `bin/wfdbx` named after a tool runs that tool.

The launchers also take a few arguments of their own, in place of the tool's name: `--list` prints the names of the tools, one per line (Python, R, MATLAB/Octave and Kotlin); `--version` prints the version of WFDB-x, as `WFDB-x` followed by the version number (Python, Julia, R, MATLAB/Octave and Kotlin); `-h` or `--help` prints the launcher's usage (Python, R, MATLAB/Octave and Kotlin), and the Python launcher also has `--examples` *DIR*, which copies the example notebooks to *DIR* (by default `wfdbx-examples`). Without arguments, or with an unknown tool name, a launcher prints a message on the standard error and exits with status 1. The programs of C++ and Rust, like the C tools, have no `--version` option.

14.2 Conventions

14.2.1 Records and files

A record is named as in Chapter 2, without the suffix of its header (`100s`, not `100s.hea`), and an annotation file by its record and its annotator (`-r 100s -a atr` is the file `100s.atr`). The tools find every file that they read through the library, as Chapter 3 describes: a relative name is taken relative to the *input directory*, set by the environment variable `WFDBX_INPUT_DIR` (by default the current directory), an absolute name, a ~ name or a URL is used as given, and nothing is searched for. A URL (only `http://` and `https://` URLs are read) is opened only when network access is turned on, by `WFDBX_NETWORK=1`. Files written through the library (headers, signal files, annotation files) are created in the *output directory*, `WFDBX_OUTPUT_DIR` (by default the current directory). For example, to read record 100 of the MIT-BIH Arrhythmia Database from PhysioNet:

```
export WFDBX_INPUT_DIR=https://physionet.org/files/mitdb/1.0.0/
export WFDBX_NETWORK=1
wfdbx rdsamp -r 100 -t 0:05
```

A few files are opened by the tools themselves, not through the library: the input files named by `-i` in `wrsamp`, `edf2mit`, `wav2mit`, `a2m` and `ad2m`, the input files of `ahaecg2mit`, the script of `xform -S`, and the signal file whose size `wfdbdesc` measures; and the output files that a tool writes itself: the WAV file of `mit2wav`, the EDF file of `mit2edf`, the master header of `snip -m`, and the `.des`, `.ecg`, `.key` and `.txt` files of `parsescp`. They follow the same rules as the files of the library (Section 3.3.2): a relative name is read from the input directory, or created in the output directory (with its missing directories), a ~ name is expanded, an absolute name is used as given, and - keeps its meaning. A URL is not accepted for these files, also when the input directory is a URL: the tool reports that it cannot read, or create, the file. In legacy mode, the input files are opened as

named, relative to the current directory, and the output directory, if one is set, is a plain prefix of the names of the output files.

14.2.2 Time arguments

Options that take a time (`-f`, `-t`, `-l` and others) accept the time strings of the library function `strtim`:

- an elapsed time from the beginning of the record, `hh:mm:ss`, `mm:ss` or `ss`, with an optional fraction of a second (`1:30.5`, `10`);
- a sample number, `s` followed by the number (`s21600`);
- a time of day, in brackets: `[hh:mm:ss]`, `[hh:mm:ss dd/mm/yyyy]`, or `[hh:mm:ss d]` with a number of days after the start (it needs the base time of the record);
- `e`, the end of the record; `c` followed by a counter value; `f` followed by a frame number.

Elapsed times and times of day need the sampling frequency of the record. When a record does not give one, WFDB-x reports an error instead of assuming 250 Hz (Chapter 4); sample numbers and `e` need no frequency. The annotation tools `rdann`, `ann2rr`, `sortann`, `a2m` and `m2a` take `-F` *FREQ* for that case, and `rr2ann` already had it.

14.2.3 Help, options and arguments

The options are those of the C tools: one letter after a hyphen, given separately (`-v -p`, not `-vp`), their arguments as the next command-line arguments. As in C, only the first letter after the hyphen is read in most tools, so that `-pd` is `-p` with the time format `d`, and an argument that is not an option where an option is expected is an error (`unrecognized argument`). Options that take a list (`-s` *SIGNAL* `[`*SIGNAL* `...]`, `-p` *TYPE* `...`) take every following argument that does not begin with a hyphen.

`-h` prints the tool's usage summary on the standard error. A tool that needs an argument that was not given (often the record name) prints the same summary. The exit status after `-h` is that of the C tool, 0 or 1 depending on the tool (each entry says which); as in C, `wfdbdesc`, `sampfreq` and `revise` take `-h` alone as a record name. The help texts of `ann2rr` and `wrsamp` pause after 23 lines, print `--More--` and read a line from the standard input, as the C tools do. The usage line of each tool is given at the start of its entry below.

14.2.4 Legacy mode

`--legacy`, as the *first* argument of a tool (`rdsamp --legacy -r 100s`, `wfdbx rdann --legacy -r 100s -a atr`), is removed before the options are read, and the tool then behaves exactly as the original one: the options added by WFDB-x (such as `-F` of `rdann` or `-R` of `wrsamp`) are unrecognized, the help texts and messages are those of the original, the built-in values of the original (250 Hz, 200 adu/mV, ...) are used, files are found through the WFDB path (the environment variable `WFDB`; the files that a tool reads itself are opened as named), and the exit status is the original one. `--legacy` anywhere else is not recognized: it is treated as any other argument. Legacy mode is meant for reproducing results of the original package; Chapter 4 explains why it is not the default.

14.2.5 Standard input and output

Text output goes to the standard output, and messages, prompts and help texts to the standard error. Some tools read the standard input (`wrsamp` without `-i`, `edf2mit -i -`, `wrann`, `rr2ann`, `parsescp`, `readid`) and some write binary data to the standard output (`wfdbcat`, `makeid`, `parsescp -a`, and `wrsamp` without `-o`).

14.2.6 Exit status

The exit status is that of the C tool: 0 on success, 1 for an error in the arguments (including a required option that is missing), 2 when the record or a file cannot be read, and, in some tools, other values that their entries give. In addition, WFDB-x follows the convention of the standard Unix tools: when an output file or the standard output cannot be written (a full disk, for example), a tool that would have exited with status 0 exits with status 1, and reports a standard output that could not be written as `PROGRAM: write error on standard output`, and a file that `snip`, `mit2edf` or `parsescp` writes itself as `PROGRAM: write error on FILE` (Chapter 4). The original tools exited with status 0 in both cases; legacy mode does too.

14.3 Summary of the tools

Table 14.2 The 33 tools of WFDB-x.

Tool	What it does
Signal tools (Section 14.4)	
`rdsamp`	print the samples of a record (text, CSV or XML; ADC or physical units)
`wrsamp`	write a record from a text file of samples
`wfdbdesc`	describe the signals of a record (the contents of its header)
`wfdbtime`	convert times to sample numbers, elapsed times and times of day
`time2sec`	convert a time to seconds
`sampfreq`	print the sampling frequency of a record
`signame`	print the names of the signals of a record
`signum`	print the numbers of the signals with given names
`sigamp`	measure signal amplitudes (peak-to-peak near beats, or RMS)
`snip`	copy a segment of a record to a new record
`xform`	convert a record: storage format, sampling frequency, gain, signals
`wfdbcat`	copy WFDB files to the standard output
`wfdbwhich`	print the path of a WFDB file
Annotation tools (Section 14.5)	
`rdann`	print the annotations of an annotation file
`wrann`	write an annotation file from text in the format of `rdann`
`sortann`	sort an annotation file into canonical order
`ann2rr`	print the intervals between annotations (RR intervals)
`rr2ann`	write an annotation file from a list of RR intervals
Format converters (Section 14.6)	
`edf2mit`	convert an EDF or EDF+ file to a WFDB record
`mit2edf`	convert a WFDB record to an EDF or EDF+ file
`rdedfann`	print the annotations of an EDF+ file as `rdann` text
`wav2mit`	make a WAV file readable as a WFDB record
`mit2wav`	convert a WFDB record to a WAV file
`wfdb2mat`	convert a WFDB record to a MATLAB/Octave `.mat` file
`a2m`	convert an AHA-format annotation file to WFDB (MIT) format
`m2a`	convert a WFDB annotation file to AHA format
`ad2m`	convert an AHA-format signal file (tape or CD-ROM) to a WFDB record
`md2a`	convert a WFDB record to an AHA-format signal file

Tool	What it does
`ahaecg2mit`	convert AHA DVD files (binary or text) to WFDB records
`makeid`	write an AHA-format record ID block
`readid`	read an AHA-format record ID block
`revise`	convert an obsolete-format header to the current format
`parsescp`	read an SCP-ECG file, describe it, and convert it to a record

14.4 Signal tools

rdsamp

Print the samples of a record

```
usage: rdsamp -r RECORD [OPTIONS ...]
```

Reads the signals of a record and prints one line per sample interval: the sample number (or, with `-p`, the time) and the sample of each selected signal, in ADC units, or in physical units with `-p`. The output is plain text in columns (tab-separated), CSV (`-c`) or WFDB-XML (`-X`), which `wrsamp` reads back. An invalid sample is printed as - in physical units, and as the value -32768 in ADC units.

-r The input record, *RECORD* (required).

-c Use the CSV (comma-separated value) output format: the column headings of `-v` are quoted (`'sample #','MLII','V5'`), and the columns are separated by commas.

-f Begin at the time that follows (Section 14.2.2); by default, the beginning of the record.

-h Print the usage summary, and exit with status 0.

-H Read multifrequency records in high-resolution mode: one line per sample of the signals with the most samples per frame, the samples of the other signals repeated (by default, one line per frame, the samples of a frame averaged).

-l Stop after the time interval that follows (`hh:mm:ss`, or `s` followed by a number of samples), counted from the beginning time; a `-t` time before the end of that interval still applies.

-p Print times and samples in physical units, with 3 decimal places (times in seconds by default). A letter appended to the option chooses the time column: `-pd` the time of day and the date, when the record has a base time (elapsed time otherwise), `-pe` the elapsed time as `hh:mm:ss.mmm`, `-ph` hours, `-pm` minutes, `-ps` seconds (the default), `-pS` the sample number.

-P As `-p`, with 8 decimal places (columns 15 characters wide); `-Pd`, `-Pe`, `-Ph`, `-Pm`, `-Ps` and `-PS` choose the time column.

-s Print only the signals that follow, given by number (0, 1, ...) or by name, in the order given: `-s` *SIGNAL* `[`*SIGNAL* `...]`. A signal that the record does not have is an error.

-S Begin at the first valid sample of the signal that follows (a number or a name), at or after the `-f` time.

-t Stop at the time that follows (by default, the end of the record).

-v Print column headings (the signal names, and with `-p` the units).

-X Print WFDB-XML (an XML prolog, the sampling frequency, the number of signals, the headings and the samples); implies `-c` and `-v`.

The first four samples of `100s`, in ADC units, then four samples from 1 second in physical units (`100s` gives no units, so they are shown as `nd`), and the CSV format:

```
$ rdsamp -r 100s -t s4
              0     995    1011
              1     995    1011
              2     995    1011
              3     995    1011
$ rdsamp -r 100s -f 1 -t s364 -p -v
   Elapsed time    MLII      V5
      (seconds)    (nd)    (nd)
          1.000  -0.535  -0.205
          1.003  -0.505  -0.080
          1.006  -0.415   0.015
          1.008  -0.300   0.065
$ rdsamp -r 100s -t s3 -c -v
'sample #','MLII','V5'
0,995,1011
1,995,1011
2,995,1011
```

Returns Exit status 0 when the samples have been printed (also when a read error or a checksum error ends the output early, as in C).

Errors Exit status 1 for an error in the arguments (or a missing `-r`); 2 if the record cannot be opened, a signal of `-s` or `-S` cannot be found, or the sampling frequency or a gain that the output needs is not specified.

Differences from the C tool (Chapter 4): the sampling frequency is required for time arguments other than sample numbers, for time columns (`-p` without `-pS`, or with `-v`) and for `-X`; the gain of each selected signal is required for physical units (`rdsamp: signal N of record R is uncalibrated`, status 2); unspecified units are shown as `nd` (C: `mV`). The C tool assumed 250 Hz and 200 adu/mV. The quotes and backslashes of signal names are escaped in CSV headings, where the C tool did not terminate.

---

wrsamp

Write a record from a text file of samples

```
usage: wrsamp [OPTIONS ...] [COLUMN ...]
```

Reads a text file with a sample of each signal on each line, and writes the samples to a signal file and, with `-o`, a header. Fields are separated by tabs, spaces, commas or any combination of these (consecutive separators count as one). If the first line contains letters, it gives the signal names; if the second line also contains letters, it gives the units (with or without parentheses). WFDB-XML, as written by `rdsamp -X`, is read as well. Non-numeric fields are written as invalid samples. Samples are rounded to the nearest integer after scaling (`-x`).

COLUMN
: The fields to copy, by number (the leftmost field is column 0), in the order of the output signals; by default, all fields in order (see `-z`).

-c Accepted for compatibility; it has no effect (as in the C tool).

-d Add dither (random noise of up to one unit) to the samples before rounding.

-f Start copying at the input line that follows (a number, ≥ 1; the first line of samples is line 0).

-F The sampling frequency to write in the header (Hz). Required with `-o`.

-G The gain (adu per physical unit) to write in the header; a quoted list of values (`-G "100 50"`) gives one per signal, the last applying to the remaining signals. A gain of 0 declares the signals uncalibrated. Required with `-o`.

-h Print the usage summary, and exit with status 1.

-i Read the input from the file that follows (by default, or with `-`, the standard input). The file is read from the input directory (Section 14.2.1).

-o Write the record that follows: the signal file *RECORD*`.dat` and the header *RECORD*`.hea`. Without `-o`, the samples are written to the standard output (in format 16 unless `-O` is given) and no header is written.

-O The storage format of the output (default 16); any format of Table 15.1.

-r The character that follows separates input lines (records), instead of the line feed.

-R The ADC resolution in bits; a quoted list gives one per signal, as for `-G`. Required with `-o`.

-s Any character of the string that follows separates input fields (by default: space, tab, carriage return or comma; only tab if the first line contains a tab).

-t Stop copying at the input line that follows (a number, ≥ 1).

-U The units of the signals; a quoted list gives one per signal, as for `-G`. Required with `-o` for calibrated signals, unless the input gives the units.

-x Multiply the input values by the scale factor that follows; a quoted list gives one per signal.

-z Do not copy column 0 (usually a sample number), unless it is selected by a *COLUMN* argument.

The obsolete option `-l` (and its argument) is ignored, with a message, as in C. For example, a record written from the output of `rdsamp` (column 0, the sample number, is skipped by `-z`):

```
$ rdsamp -r 100s -t s5 > s.txt
$ wrsamp -i s.txt -o rec -F 360 -G 200 -R 11 -U mV -z
$ cat rec.hea
rec 2 360 5
rec.dat 16 200/mV 11 0 995 4975 0 col 1
rec.dat 16 200/mV 11 0 1011 5055 0 col 2
$ wrsamp -i s.txt -o rec2
wrsamp: the sampling frequency must be specified (-F)
```

Returns Exit status 0 when the record has been written.

Errors Exit status 1 for an error in the arguments, a missing `-F`, `-G`, `-R` or `-U` (`wrsamp: the ... must be specified (-X)`), or an XML input without samples; 2 if the input file cannot be read or the output cannot be opened; 3 if the input contains no line separator.

Differences from the C tool (Chapter 4): when a record is written (`-o`), the sampling frequency (`-F`), the gains (`-G`), the ADC resolutions (the new option `-R`) and, for calibrated signals, the units (the new option `-U`) must be given; the C tool assumed 250 Hz, 200 adu/mV, a resolution based on the format, and millivolts. A header written with `-U mV` says `200/mV` where the C tool wrote `200`. The `-i` file is read from the input directory (C: from the current directory).

wfdbdesc — Describe the signals of a record

```
usage: wfdbdesc RECORD [-readable]
```

Prints the contents of a record's header in readable form: the info strings (`Notes`), the starting time, the length, the sampling frequency, and for each signal its file, description, gain and

units, initial value, storage format, ADC resolution and zero, baseline and checksum. For a multi-segment record, it also gives the layout (fixed or variable) and, for a fixed layout, the length of the first segment, whose signals it describes.

RECORD
: The record to describe.

-readable
: Describe only the signals whose files can be opened (by default, every signal named in the header, without opening the files). Any abbreviation of `-readable` is accepted.

```
$ wfdbdesc 100s
Record 100s
Notes
=====
 69 M 1085 1629 x1
 Aldomet, Inderal
Produced by xform from record 100, beginning at 0:0
=====

Starting time: not specified
Length:     1:00.000 (21600 sample intervals)
Sampling frequency: 360 Hz
2 signals
Group 0, Signal 0:
 File: 100s.dat
 Description: MLII
 Gain: 200 adu/nd
 Initial value: 995
 Storage format: 212
 I/O: can be unbuffered
 ADC resolution: 11 bits
 ADC zero: 1024
 Baseline: 1024
 Checksum: 21537
Group 0, Signal 1:
 ...
```

Returns Exit status 0.

Errors Exit status 1 without a record name; 2 if the header cannot be read, or the record has no signals.

Differences from the C tool (Chapter 4): an unspecified sampling frequency or ADC resolution is shown as `not specified` (lengths are then given in sample intervals only), a gain of 0 as `uncalibrated` (C: `uncalibrated; assume 200`), and unspecified units as `adu/nd`; the C tool assumed 250 Hz, 12 bits and millivolts. For a record whose header gives no length, the size of the first signal file, from which the length is computed, is measured in the header's directory, where the library reads the file (C: in the current directory). The notes are the info strings that `getinfo` returns, after the record line found as the header parser finds it (Section 4.6): for a header whose first line is empty, the comments before the record line are not printed (the C tool, and `wfdbdesc --legacy`, print them as notes).

wfdbtime Convert times and sample numbers

```
usage: wfdbtime [ -r RECORD [ TIME ] ]
```

Converts each time argument, at the sampling frequency of the record, and prints three columns: the sample number (`s` followed by the number), the elapsed time from the beginning of the record, and the time of day (the elapsed time again if the record has no base time).

`-r` The record, *RECORD*, whose sampling frequency and base time are used. A later `-r` switches to another record for the times that follow it.

`TIME`
A time to convert, in any form of Section 14.2.2; any number of times may follow the record.

```
$ wfdbtime -r 100s 1:00 s100
         s21600        1:00.000        1:00.000
           s100        0:00.278        0:00.278
```

Returns Exit status 0.

Errors Exit status 1 without `-r` (also for `-h`, which prints the usage); 2 if the record cannot be opened or its header gives no sampling frequency.

Differences from the C tool (Chapter 4): the record must give its sampling frequency (C: 250 Hz assumed).

`time2sec` — Convert a time to seconds

```
usage: time2sec [-r RECORD] TIME
```

Prints a time interval in seconds. Without `-r`, *TIME* is converted at 1000 samples per second and printed with up to 12 significant digits; with `-r`, it is converted at the sampling frequency of the record and printed with at most three decimals (milliseconds). A time of day in brackets (`[9:0:0]`) gives the elapsed time from the beginning of the record.

`-r` The record, *RECORD*, whose sampling frequency (and base time) are used.

`TIME`
The time to convert (Section 14.2.2).

```
$ time2sec 1:30.5
90.5
$ time2sec -r 100s s1000
2.778
```

Returns Exit status 0.

Errors Exit status 1 for other arguments (also for `-h`), after the usage summary. Exit status 2 with `-r` if the header of the record cannot be read, or gives no sampling frequency (`time2sec: sampling frequency of record RECORD is not specified`).

Differences from the C tool: with `-r`, a record whose header gives no sampling frequency (or cannot be read) is an error. The C tool converted the time at its default frequency of 250 Hz, a result unrelated to the record, because WFDB-x assumes no frequency that a record does not give (Section 4.1); until version 1.2.1, WFDB-x divided by the unspecified frequency (0) and printed `-9223372036854775808` with the exit status 0. With `--legacy`, the C behavior (250 Hz).

sampfreq

Print the sampling frequency of a record

```
usage: sampfreq [OPTION] RECORD
```

Prints the sampling frequency (frame rate) of a record, in Hz. With `-a`, prints the name and sampling frequency of each signal, tab-separated; with `-H`, the highest sampling frequency of a multifrequency record.

RECORD
: The record.

-a List all signals and their sampling frequencies.

-h Listed in the usage summary, which is printed (status 1) when no argument is given; as in C, `sampfreq -h` alone takes `-h` as a record name.

-H Print the maximum sampling frequency of a multifrequency record.

```
$ sampfreq 100s
360
$ sampfreq -a mfreq
slow    100
medium  200
fast    400
$ sampfreq -H mfreq
400
```

Returns Exit status 0; a header that cannot be read prints the library's message and `-1`.

Errors Exit status 1 without arguments.

Differences from the C tool (Chapter 4): for a header that does not give a sampling frequency, the library reports 0, which is printed (C: 250).

signame

Print the names of the signals of a record

```
usage: signame -r RECORD [OPTIONS ...]
```

Prints the name (description) of each signal of the record, one per line, or of the signals whose numbers follow `-s`. A number that is not that of a signal prints `[INVALID]`.

-r The record, *RECORD* (required).

-h Print the usage summary, and exit with status 0.

-s Print the names of the signals whose numbers follow: `-s` *SIGNAL* `[`*SIGNAL* `...]`.

```
$ signame -r 100s -s 1 0 2
V5
MLII
[INVALID]
```

Returns Exit status 0.

Errors Exit status 1 for an error in the arguments; 2 if the record cannot be opened or has no signals.

Differences from the C tool: none, apart from `--legacy` (the C tool also read past the end of its array for the number equal to the number of signals; WFDB-x prints `[INVALID]`).

---

signum

Print the numbers of the signals with given names

```
usage: signum -r RECORD [OPTIONS ...]
```

For each name that follows `-s`, prints the numbers of the signals of the record with that exact name (tab-separated, when several have it), or `X` if none has it, one line per name.

-r The record, *RECORD* (required).
-h Print the usage summary, and exit with status 0.
-s The signal names to look for (required): `-s` *SIGNAL* [*SIGNAL* ...].

```
$ signum -r 100s -s V5 MLII ABP
1
0
X
```

Returns Exit status 0.

Errors Exit status 1 for an error in the arguments, or without `-r` or `-s`; 2 if the record cannot be opened.

Differences from the C tool: none, apart from `--legacy`.

---

sigamp

Measure signal amplitudes

```
usage: sigamp -r RECORD [OPTIONS ...]
```

Measures the amplitude of each signal, up to *NMAX* times, and prints the trimmed mean of the measurements (the largest and smallest 5% excluded), one tab-separated column per signal. With an annotator (`-a`), each measurement is the peak-to-peak amplitude in a window around a beat annotation (an annotation that `map1` maps to `NORMAL`); without one, it is the RMS amplitude over consecutive windows of 1 second (`-w`). The amplitudes are in ADC units, or in physical units with `-p`.

-r The record, *RECORD* (required).
-a Measure peak-to-peak amplitudes near the beat annotations of the annotator that follows. `-a` and `-w` cannot be used together (`-w` is then ignored, with a message).
-d The window of the peak-to-peak measurements, *DT1 DT2*: the time before and the time after each annotation (default 0.05 seconds each); used only with `-a`.
-f Begin at the time that follows.
-h Print the usage summary, and exit with status 0.
-H Read multifrequency records in high-resolution mode.
-n Make up to the number of measurements that follows per signal (default 300).
-p Print the amplitudes in physical units. A letter appended chooses the time format of the individual measurements (`-q`, `-v`): `-pd` the time of day, `-pe` `hh:mm:ss`, `-ph` hours, `-pm` minutes, `-ps` seconds (the default), `-pS` sample numbers.

-q Quick mode: print the individual measurements only, each with its time.
-t Stop at the time that follows.
-v Verbose mode: print the individual measurements, then the trimmed means (`Trimmed mean`).
-w The window of the RMS measurements, *DTW* (default 1 second).

The environment variable `WFDBGVMODE` sets the resolution mode when `-H` is not given, as in C.

```
$ sigamp -r 100s
        32.97   23.7155
$ sigamp -r 100s -a atr -n 4 -v
0.213889        265     176
1.027778        295     193
1.838889        306     216
2.627778        301     216
Trimmed mean    291.75  200.25
```

Returns Exit status 0.

Errors Exit status 1 for an error in the arguments; 2 if the record or the annotator cannot be opened, the record gives no sampling frequency, or a signal is uncalibrated with `-p`.

Differences from the C tool (Chapter 4): the record must give its sampling frequency (the windows are in seconds), and, with `-p`, the gain of each signal (C: 250 Hz and 200 adu/mV assumed).

---

snip

Copy a segment of a record to a new record

```
usage: snip -i IREC -n NREC [OPTIONS ...]
```

Copies the samples of a record between two times to a new record, with the same signals, storage format and signal specifications, and optionally its annotations, with times relative to the beginning of the excerpt. The header of the new record keeps the info strings of the input and adds `Produced by snip from record IREC, beginning at TIME`. To change the sampling frequency, gains or signals, `xform` is the tool to use.

-i The input record, *IREC* (required; `-r` is accepted as a synonym).
-n The new record, *NREC* (required). It must differ from *IREC*: the header of the new record is written over that of a record of the same name.
-a Copy the annotations of the annotators that follow: `-a` *ANNOTATOR* [*ANNOTATOR* `...`].
-f Begin at the time that follows (default 0).
-h Print the usage summary, and exit with status 0.
-l Copy only the duration that follows (`hh:mm:ss` or `s` followed by a number of samples); overrides `-t`.
-m Keep the segments of a multi-segment input record, if possible. The master header of the new record is created in the output directory, as the other files are.
-O Write the samples in the storage format that follows (by default, the format of the input signals, or, if they differ, the best fit to the largest ADC resolution).
-s Do not write info strings in the new header (neither those of the input nor the `Produced by snip` line).
-t Stop at the time that follows (default: the end of the record).

```
$ snip -i 100s -n ex -f 10 -t 20 -a atr
```

```
$ cat ex.hea
ex 2 360 3600
ex.dat 212 200 11 1024 946 -16262 0 MLII
ex.dat 212 200 11 1024 969 18142 0 V5
# 69 M 1085 1629 x1
# Aldomet, Inderal
#Produced by xform from record 100, beginning at 0:0
#Produced by snip from record 100s, beginning at 10
```

Returns Exit status 0.

Errors Exit status 1 for an error in the arguments or an improper interval; 2 if a header cannot be read or written, or the input record gives no sampling frequency.

Differences from the C tool (Chapter 4): the input record must give its sampling frequency, for the new header (C: 250 Hz assumed); an output signal file is never one of the input files; when the last frame to copy cannot be read and no frame was read before it, 0 is written (C: uninitialized values); the master header of `-m` is created in the output directory (C: in the current directory).

`xform` — Convert a record (format, frequency, gain, signals)

```
usage: xform -i IREC [OPTIONS ...]
```

Copies the signals of a record to a new one, converting them: to another storage format, sampling frequency (by interpolation), gain, ADC resolution or zero level, and to a subset or another order of the signals; annotations can be copied too, with their times converted. The output is described either by an existing header (`-o`) or by answers to prompts (printed on the standard error and read from the terminal, the standard input, or a script given by `-S`): the output record name (unless `-n` gives it), the number of signals, the sampling frequency, the directory of the signal files, the storage format, whether all signals go in one file, and for each signal its gain, ADC resolution and ADC zero (and with `-N` its description and units). Samples out of the range of the output format wrap around unless `-c` is given. While it converts, it prints `Checking input signals ...` and progress dots.

-i The input record, *IREC* (required).
-a Copy the annotations of the annotators that follow: `-a` *ANNOTATOR* [*ANNOTATOR* ...].
-c Clip the output samples to the range of the format (by default, they wrap around).
-d Add dither to the input if the sampling frequency or the gain changes.
-f Begin at the time that follows.
-h Print the usage summary, and exit with status 0.
-H Open the input record in high-resolution mode.
-M Process a multifrequency record without changing the frequencies of its signals.
-n Create a header for the record *NREC* that follows, with the signal descriptions of *IREC*.
-N As `-n`, but with the signal descriptions of *OREC* (or asked for).
-o Write the output signal files as the existing header of the record *OREC* that follows describes them (no prompts).
-s Write only the input signals that follow, by number or name, in that order: `-s` *SIGNAL* [*SIGNAL* ...]. A signal that the record does not have is skipped, with a warning.

-S Take the answers to the prompts from the text file *SCRIPT* that follows, one answer per line (an empty line takes the default answer). The file is read from the input directory (Section 14.2.1).

-t Stop at the time that follows.

-u Adjust annotation times, if needed, to make them unique.

For example, a copy of the first 2 seconds of `100s` in format 16, with the answers in the file `ans.txt` (empty lines, except `16` on the fourth: the default number of signals, frequency and directory, then format 16, then the default answers):

```
$ xform -i 100s -n x16 -S ans.txt -t 2 2>/dev/null
$ cat x16.hea
x16 2 360 720
x16.dat 16 200 11 1024 995 -26770 0 MLII
x16.dat 16 200 11 1024 1011 -12260 0 V5
# 69 M 1085 1629 x1
# Aldomet, Inderal
#Produced by xform from record 100, beginning at 0:0
#Produced by xform from record 100s, beginning at 0:0
```

Returns Exit status 0.

Errors Exit status 1 for an error in the arguments, identical `-n` and `-o` names, `-M` with a change of frequency, an output frequency of 0, a missing ADC resolution, a `-t` time before the `-f` time, or the end of a script before the last answer; 2 if a record cannot be opened or written, a sampling frequency is not specified, or a requested rescaling is impossible.

Differences from the C tool (Chapter 4): the input record (and the record of `-o`) must give a sampling frequency, and an output frequency of 0 is refused (C: 250 Hz); rescaling an uncalibrated signal to a given gain, and rescaling by ADC resolution when a resolution is unspecified, are refused; an ADC resolution is offered as the default only if it comes from the input (or the previous output signal) and suits the output format, and must be given otherwise; unspecified units are shown as `nd` in the prompts; an output signal file that would be one of the input files is refused (`osigfopen: output file NAME is also an input file`). With `-M` and an output frame larger than the input frame, the samples beyond the input frame are 0 (C: a crash). The script of `-S` is read from the input directory (C: from the current directory).

---

wfdbcat

Copy WFDB files to the standard output

```
usage: wfdbcat FILE [FILE ...]
```

Copies each file, found as the library finds files (Section 14.2.1: relative to the input directory, or as a full path or a URL), to the standard output, byte for byte. A file that cannot be found is reported (`wfdbcat: 'FILE' not found`) and skipped.

FILE
A file to copy, named with its suffix (`100s.hea`).

```
$ wfdbcat 100s.hea
100s 2 360 21600
100s.dat 212 200 11 1024 995 21537 0 MLII
100s.dat 212 200 11 1024 1011 -3962 0 V5
# 69 M 1085 1629 x1
```

```
# Aldomet, Inderal
#Produced by xform from record 100, beginning at 0:0
```

Returns Exit status 0, also when a file was not found.

Errors Exit status 1 without arguments, or with `-h` alone.

Differences from the C tool (Chapter 4): each file is read from where it is named, not found by a search of the WFDB path.

---

wfdbwhich Print the path of a WFDB file

```
usage: wfdbwhich [-r RECORD ] FILENAME
 -or-  wfdbwhich FILE-TYPE RECORD
```

Prints the path by which the library opens a file: a file named by *FILENAME*, or the file of type *FILE-TYPE* (a suffix such as `hea`, `dat` or `atr`) of a record. A file that cannot be found is reported with the input directory (`(WFDB path = ...)`).

-r Open the record *RECORD* first.

FILENAME
The name of the file to find.

FILE-TYPE
The type (suffix) of the file of the record to find.

RECORD
The record.

```
$ wfdbwhich hea 100s
100s.hea
$ wfdbwhich 100x.dat
`100x.dat' not found
(WFDB path = .)
```

Returns Exit status 0 when the file is found.

Errors Exit status 1 for other arguments (also `-h`); 2 if the file is not found.

Differences from the C tool (Chapter 4): the file is looked for only where it is named (the input directory, a full path or a URL), not by a search of the WFDB path, and the path printed in the message is the input directory.

14.5 Annotation tools

---

rdann Print the annotations of an annotation file

```
usage: rdann -r RECORD -a ANNOTATOR [OPTIONS ...]
```

Prints one line per annotation: the time (elapsed time, or time of day in brackets when the record has a base time), the sample number, the mnemonic of the annotation type, the subtype, channel and number fields, and the auxiliary information (after a tab), if any. Options select the

annotations to print and the time format. This text is the input format of `wrann`. The old syntax of the C tool, `rdann` *ANNOTATOR RECORD* `[`*FROM* `[`*TO* `[`*TYPE* `...]]]`, is accepted too.

-r The record, *RECORD* (required).

-a The annotator, *ANNOTATOR* (required).

-c Print only the annotations with the channel number that follows (from −128 to 127).

-e Print the times as elapsed times, even for a record with a base time. Only one of `-e` and `-x` can be used (the last one given applies).

-f Start at the time that follows; also `a` followed by a number (from that annotation number), or `b` or `#` followed by a number (from that beat number).

-F The sampling frequency, used if the record has no header or its header gives none.

-h Print the usage summary, and exit with status 0.

-n Print only the annotations with the `num` value that follows.

-p Print only the annotations of the types that follow, given by their mnemonics: `-p` *TYPE* `[`*TYPE* `...]` (`-p N V`).

-s Print only the annotations with the subtype that follows.

-t Stop at the time that follows; also `a`, `b` or `#` followed by a number, as for `-f` (`#`: that number of beats).

-v Print column headings.

-x Print the times in seconds, minutes and hours (with enough decimal places to resolve the time resolution of the file).

```
$ rdann -r 100s -a atr -t 3 -v
      Time   Sample #  Type  Sub Chan  Num      Aux
    0:00.050       18     +    0    0    0      (N
    0:00.214       77     N    0    0    0
    0:01.028      370     N    0    0    0
    0:01.839      662     N    0    0    0
    0:02.628      946     N    0    0    0
$ rdann -r 100s -a atr -t 2 -x
    0.050   0.00083 0.0000139     +    0    0    0      (N
    0.214   0.00356 0.0000594     N    0    0    0
    1.028   0.01713 0.0002855     N    0    0    0
    1.839   0.03065 0.0005108     N    0    0    0
```

Returns Exit status 0.

Errors Exit status 1 for an error in the arguments, or if the sampling frequency is not specified (`rdann: sampling frequency of record R is not specified (use -F)`); 2 if the annotation file cannot be opened or a starting annotation or beat number is beyond its end.

Differences from the C tool (Chapter 4): the new option `-F` gives the sampling frequency of a record without a header (or whose header gives none), and is then required; the C tool assumed 250 Hz.

---

wrann — Write an annotation file from text

```
usage: wrann -r RECORD -a ANNOTATOR <TEXT-FILE
```

Reads annotations from the standard input, in the format printed by `rdann`, and writes the annotation file *RECORD*.*ANNOTATOR*. The time column is skipped and the sample number is

used, except on lines that begin with a time of day in brackets, whose time is converted. A mnemonic that is not an annotation code is written as a `NOTE` (comment) annotation whose auxiliary information begins with that mnemonic. Text after a tab is the auxiliary information. The time resolution recorded in the file is the sampling frequency of the record, if its header gives one. Annotations that are not in canonical order are sorted when the file is closed (Chapter 4).

-r The record, *RECORD* (required).
-a The output annotator, *ANNOTATOR* (required).

`-h`, not listed in the usage, prints it and exits with status 0.

```
$ rdann -r 100s -a atr | wrann -r 100s -a copy
```

Returns Exit status 0.

Errors Exit status 1 for an error in the arguments; 2 if the output file cannot be opened.

Differences from the C tool (Chapter 4): an out-of-order file is sorted by the library itself, without losing annotations, instead of by running `sortann`.

---

sortann — Sort an annotation file

```
usage: sortann -r RECORD -a ANNOTATOR [OPTIONS ...]
```

Rearranges the annotations of a file in canonical order: by time, then `num`, then `chan`. A later annotation with the same time, `num` and `chan` replaces an earlier one, and `NOTQRS` annotations are dropped. Without `-o`, the sorted file is written under the name of the input (in the output directory, so that it replaces the input when the input and output directories are the same), unless the input is already sorted: that is reported (`sortann: input is already ordered -- no output written`) and nothing is written.

-r The record, *RECORD* (required). As in C, `-r`*DIR* (the directory attached to the option) sets the input directory before the record name.
-a The input annotator, *ANNOTATOR* (required).
-f Copy only the annotations from the time that follows (requires `-o`).
-F The sampling frequency, used if the record has no header or its header gives none; then required for `-f` and `-t`.
-h Print the usage summary, and exit with status 0.
-o Write the output as the annotator *OUTANN* that follows (it must differ from the input annotator); required with `-f` or `-t`.
-t Copy only the annotations before the time that follows (requires `-o`).

```
$ sortann -r 100s -a atr
sortann: input is already ordered -- no output written
```

Returns Exit status 0, also for an input that is already sorted.

Errors Exit status 1 for an error in the arguments, a missing `-o`, identical input and output annotators, or a sampling frequency that is needed and not specified; 2 if an annotation file cannot be opened.

Differences from the C tool (Chapter 4): the new option `-F`; an out-of-order annotation at time 0 (with number and channel 0) is no longer lost, nor are those inserted after it (the C tool dropped them; legacy mode does too).

---

ann2rr

Print the intervals between annotations

```
usage: ann2rr -r RECORD -a ANNOTATOR [OPTIONS ...]
```

Prints the intervals between consecutive beat (QRS) annotations, one per line, in sample intervals by default, optionally with the times and types of the annotations that begin and end each interval. The first interval begins at the starting time.

-r The record, *RECORD* (required).

-a The annotator, *ANNOTATOR* (required).

-A Print all intervals between annotations, not only RR intervals (overrides `-c` and `-p`).

-c Print only intervals between consecutive valid annotations.

-f Start at the time that follows.

-F The sampling frequency, used if the record has no header or its header gives none; then required for times and for intervals in units of time.

-h Print the usage summary, and exit with status 0.

-i Print the intervals in the format that follows: `h` (hours), `m` (minutes), `s` (seconds), each optionally followed by a number of decimal places from 0 to 15 (default 3; `-i s8`), or `t` (`hh:mm:ss`).

-p Print only intervals that end with annotations of the types that follow: `-p` *TYPE* `[`*TYPE* `...]`, given by mnemonics such as `N` or `V`.

-P Print only intervals that begin with annotations of the types that follow.

-t Stop at the time that follows.

-v Print the time of the end of each interval, in the format that follows, if any: `h`, `m` or `s` (with decimal places, as for `-i`), `t`, or `T` (date and time, if available); without a format, in sample intervals.

-V Print the time of the beginning of each interval, as for `-v`.

-w Print the annotations that end the intervals.

-W Print the annotations that begin the intervals.

```
$ ann2rr -r 100s -a atr -t 4 -i s -V s -w
0.000   0.214   N
0.214   0.814   N
1.028   0.811   N
1.839   0.789   N
2.628   0.792   N
```

Returns Exit status 0.

Errors Exit status 1 for an error in the arguments, or a sampling frequency that is needed and not specified; 2 if the annotation file cannot be opened.

Differences from the C tool (Chapter 4): the new option `-F`; the C tool assumed 250 Hz.

rr2ann Write an annotation file from RR intervals

```
usage: rr2ann -r RECORD -a ANNOTATOR [OPTIONS ...] <TEXT-FILE
```

Reads a column of RR intervals (or of times of occurrence, with `-T`) in sample intervals from the standard input, and writes an annotation file of `N` (normal beat) annotations at the corresponding times, or of the types of a second column (`-w`). The first token of a line is the interval; empty lines, non-numeric tokens, and negative or zero intervals are skipped. If the record has no header, one is created.

-r The record, *RECORD* (required).
-a The output annotator, *ANNOTATOR* (required).
-F The sampling frequency, used (and required) if the header does not exist or does not give one; otherwise ignored, with a warning.
-h Print the usage summary, and exit with status 0.
-T The input contains times of occurrence, not RR intervals.
-x Multiply the input values by the factor that follows (default 1) to obtain sample intervals.
-w Take the annotation type of each line from its second column (a mnemonic).

```
$ ann2rr -r 100s -a atr | rr2ann -r 100s -a rr
$ rdann -r 100s -a rr -t 2
    0:00.214       77     N    0    0    0
    0:01.028      370     N    0    0    0
    0:01.839      662     N    0    0    0
```

Returns Exit status 0.

Errors Exit status 1 for an error in the arguments, or a sampling frequency that is needed and not given; 2 if the header cannot be created; 3 if the annotation file cannot be opened.

Differences from the C tool (Chapter 4): `-F` is required when the record has no header or its header gives no frequency (C: 250 Hz assumed); a header is created only when there is none.

14.6 Format converters

edf2mit Convert an EDF or EDF+ file to a WFDB record

```
usage: edf2mit [OPTIONS ...]
```

Reads a European Data Format file (EDF or EDF+) and writes a WFDB record with the same data: a header and a signal file in format 16, with the sampling frequencies, gains, baselines, units and start time of the EDF header. The signals `EDF Annotations` of an EDF+ file are copied as signals (`rdedfann` reads them). Unknown options are ignored.

-i The EDF file to read, *EDFILE* (required); - reads the standard input. It is read from the input directory (Section 14.2.1).
-b The input is in big-endian byte order (by default, little-endian).
-h Print the usage summary, and exit with status 0.

-r The record to create, *RECORD* (by default, the input file name up to its first dot, with every character other than a letter or digit replaced by _).

-s Copy only the signals that follow, by number (from 0): `-s` *SIGNAL* `[`*SIGNAL* `...]`.

-v Print the fields of the EDF header (debugging output) on the standard output.

```
$ edf2mit -i edfp.edf -r edfp2
$ cat edfp2.hea
edfp2 2 10 20 13:24:35 02/03/2002
edfp2.dat 16x10 4.095/uV 11 0 0 26823 0 EEG Fpz-Cz
edfp2.dat 16x3 32767.5 15 0 12331 14629 0 EDF Annotations
```

Patient data The patient and recording identification fields of an EDF header can identify a person; `-v` prints them.

Returns Exit status 0.

Errors Exit status 1 for a missing or unreadable input file or an invalid record name; 2 if the input is not EDF, or is shorter than the 256-byte fixed part of an EDF header (`edf2mit: incomplete EDF header in FILE`; no file is created).

Differences from the C tool (Chapter 4): the `-i` file is read from the input directory (C: from the current directory); otherwise none, apart from `--legacy`, except for input on which the C tool fails: a file shorter than the fixed part of the header (C: fields from uninitialized memory), and signals with 0 samples per data record (C: a crash).

---

mit2edf

Convert a WFDB record to an EDF file

```
usage: mit2edf -r RECORD [OPTIONS ...]
```

Reads a record and writes an EDF file with the same data: 16-bit samples in data records (blocks) of 10 seconds, or shorter if a block would exceed 61440 bytes. The record name is written as the patient identification, and the base date and time, if any, as the start date and time (`Startdate not recorded` otherwise). An empty line is printed on the standard output when the file is complete. Unknown options are ignored.

-r The input record, *RECORD* (required).

-h Print the usage summary, and exit with status 0.

-o The EDF file to write, *EDFILE* (default: *RECORD*`.edf`, without the directory part of the record name), created in the output directory (Section 14.2.1).

-p Write an EDF+ file (its name must end in `.edf` or `.EDF`); warnings say what EDF+ requires that the record does not give.

-v Verbose mode: print the sizes of the header, the blocks and the file, and a dot per minute of data.

```
$ mit2edf -r 100s -o 100s.edf -v
Converting record 100s to 100s.edf (EDF mode)
 Header block size: 768 bytes
 Data block size: 10 seconds (3600 frames or 14400 bytes)
 Recording length: 1:00 (7 data blocks, 25200 frames, 100800 bytes)
 Total length of file to be written: 101568 bytes
.
```

Returns Exit status 0.

Errors Exit status 1 for an error in the arguments or an invalid EDF+ file name; 2 if the record cannot be opened; 3 if its signals cannot be read, or the sampling frequency, a gain or an ADC resolution is not specified; 4 if the output file cannot be created; 5 if a frame does not fit in an EDF block.

Differences from the C tool (Chapter 4): the record must give the sampling frequency and the gain and ADC resolution of every signal (an EDF file stores physical and digital ranges); the C tool assumed 250 Hz and 200 adu/mV and guessed the resolution from the storage format. Unspecified units are written as `nd` (C: `mV`). The EDF file is created in the output directory (C: in the current directory).

`rdedfann` — Print the annotations of an EDF+ file

```
usage: rdedfann -r RECORD [OPTIONS ...]
```

Reads the `EDF Annotations` signal of an EDF+ file, read as a record (Chapter 2), and prints its annotations as text in the format of `rdann`, which `wrann` can convert to an annotation file. The annotation text is printed in place of the mnemonic (by default) or as the auxiliary information (`-x`), followed by the duration, if any.

`-r` The EDF+ file, read as a record, *RECORD* (required): `-r edfp.edf`.

`-F` Compute the sample numbers at the frequency *FREQ* that follows, in Hz (a value that is not positive gives 1).

`-h` Print the usage summary, and exit with status 0.

`-v` Print column headings.

`-x` Print the annotation text in the auxiliary information, with the type `"` (`NOTE`).

```
$ rdedfann -r edfp.edf -v
      Time   Sample #  Type  Sub Chan  Num      Aux
    0:00.500       50 Sleep_stage_W    0    0    0              duration: 1.5
    0:01.250      125 Arousal    0    0    0           duration: 1.5
    0:01.250      125 Spindle_onset    0    0    0              duration: 1.5
```

Returns Exit status 0.

Errors Exit status 1 for an error in the arguments; 2 if the record cannot be opened; 3 if it has no EDF annotations.

Differences from the C tool: none, apart from `--legacy`.

`wav2mit` — Make a WAV file readable as a record

```
usage: wav2mit -i FILE.wav -r RECORD [ OPTIONS ... ]
```

Writes a header for a WAV file (PCM, 8 or 16 bits per sample), so that the WAV file itself is read as the signal file of a record: format 16 (or 80 for 8-bit files) after a 44-byte prolog, with the sampling frequency, number of signals and resolution of the WAV file.

`-i` The WAV file, *FILE.wav* (required; the name must end in `.wav`), read from the input directory (Section 14.2.1).

-G The gain in adu per physical unit (required; 0 declares the signals uncalibrated).

-h Print the usage summary, and exit with status 1.

-r The record to create, *RECORD*: its header is *RECORD*`.hea` (default: the WAV file name without `.wav`).

-U The physical units (required unless the gain is 0).

For example, for the WAV file written by the example of `mit2wav` below:

```
$ wav2mit -i 100s.wav -r w2 -G 200 -U mV
$ cat w2.hea
w2 2 360 21600
100s.wav 16+44 200/mV 16 0 0 0 0
100s.wav 16+44 200/mV 16 0 0 0 0
```

Returns Exit status 0.

Errors Exit status 1 for an error in the arguments or a missing `-G` or `-U`; 2 if the WAV file cannot be opened; 3 if it is not a supported WAV file (not PCM, more than 16 bits per sample, inconsistent chunk lengths); 4 if the header cannot be written.

Differences from the C tool (Chapter 4): the new options `-G` (required) and `-U` (required unless the gain is 0); the C tool wrote 12.5 adu/mV for 8-bit and 6400 adu/mV for 16-bit files, in millivolts. The `-i` file is read from the input directory (C: from the current directory).

---

mit2wav

Convert a record to a WAV file

```
usage: mit2wav -o FILE.wav -r RECORD [ OPTIONS ... ]
```

Writes the signals of a record to a WAV file (16-bit PCM, one channel per signal, at the sampling frequency of the record), the samples shifted left to 16 bits (the WAV header gives the largest ADC resolution as the number of bits per sample). The samples are written through the library, and the WAV header is then put in front of them, in the same file. With `-n`, it also writes a header that describes the WAV file as a record, with the gains and baselines adjusted to the shift.

-o The WAV file to write, *FILE.wav* (required; the name must end in `.wav`), created in the output directory (Section 14.2.1).

-r The input record, *RECORD* (required).

-h Print the usage summary, and exit with status 1.

-n Write a header for the WAV file, as the record *NEWREC* that follows (it must differ from *RECORD*).

```
$ mit2wav -r 100s -o 100s.wav -n wavrec
$ cat wavrec.hea
wavrec 2 360 21600
100s.wav 16+44 6400 16 0 -928 -31712 0 MLII
100s.wav 16+44 6400 16 0 -416 4288 0 V5
```

Returns Exit status 0.

Errors Exit status 1 for an error in the arguments; 2 if the record cannot be opened; 3 if its signals cannot be read, it gives no sampling frequency, or the header of `-n` cannot be written; 4 if the output file cannot be opened.

Differences from the C tool (Chapter 4): the record must give its sampling frequency (C: 250 Hz assumed); a signal of more than 16 bits is shifted right to 16 bits and gets a gain multiplier of 0 (C: undefined shift, samples written as 0); the WAV header is put in front of the samples in the file that the library created, in the output directory (C: in the file of that name in the current directory).

---

wfdb2mat — Convert a record to a MATLAB/Octave file

```
usage: wfdb2mat -r RECORD [OPTIONS ...]
```

Writes the samples of a record (in ADC units, less the ADC zero) as a MATLAB/Octave `.mat` file (MATLAB 5 format) holding a matrix `val` of one row per signal, and a header that describes the `.mat` file as a record: format 80, 16 or 32 (the smallest that holds the ADC resolutions of the signals, or their storage formats where the resolution is not specified) after a 192-byte prolog. Both are named after the record with an `m` appended: `100sm.mat` and `100sm.hea` for `100s` (the help text, that of the C tool, says *RECORD*`.mat`). A summary of the conversion is printed on the standard output.

-r The input record, *RECORD* (required).
-f Begin at the time that follows.
-h Print the usage summary, and exit with status 0.
-H Read multifrequency records in high-resolution mode.
-l Stop after the time interval that follows (as in `rdsamp`).
-s Convert only the signals that follow, by number or name: `-s` *SIGNAL* `[`*SIGNAL* `...]`.
-S Begin at the first valid sample of the signal that follows, at or after the `-f` time.
-t Stop at the time that follows.

```
$ wfdb2mat -r 100s -t 2
Source: record 100s
val has 2 rows (signals) and 720 columns (samples/signal)
Duration:     0:02
Sampling frequency: 360 Hz  Sampling interval: 0.002777777778 sec
Row     Signal  Gain    Base    Units
1       MLII    200     0       nd
2       V5      200     0       nd

To convert from raw units to the physical units shown
above, call the 'rdmat.m' function from the wfdb-matlab
toolbox: https://physionet.org/physiotools/matlab/wfdb-app-matlab/
```

Returns Exit status 0.

Errors Exit status 1 for an error in the arguments, an empty interval (`wfdb2mat: no output`), an output larger than 2 GB, or an output file that cannot be opened; 2 if the record or a signal cannot be opened, or the record gives no sampling frequency.

Differences from the C tool (Chapter 4): the record must give its sampling frequency (C: 250 Hz); uncalibrated signals stay uncalibrated in the output header (C: 200 adu/mV); unspecified units are written as `nd` (C: `mV`); a one-character record name `m` gives the output `mm` (C wrote over the input header).

a2m

Convert AHA-format annotations to WFDB format

```
usage: a2m -i AHAFILE [ OPTIONS ... ]
```

Converts an AHA-format annotation file (Section 15.3.2) to an MIT-format annotation file. Four kinds of input are read: files in AHA format written by WFDB tools (type 0, times in sample intervals), short-format and long-format AHA tape files (types 1 and 2, times in milliseconds), and the compressed files of the AHA Database CD-ROMs and diskettes (type 3, 6 bytes per annotation, times in milliseconds). The annotation times are shifted by an offset (`-s`).

-i The AHA-format input file, *AHAFILE* (required). It is read from the input directory (Section 14.2.1).

-a The output annotator (default `atr`).

-F The sampling frequency, used (and then required) if the record header does not give one (AHA Database: 250).

-h Print the usage summary, and exit with status 1.

-r The output record, *RECORD* (default: the input file name without its directory and suffix).

-s Shift the annotations by the time that follows (default: 0 for type 0, 5:0 for type 1, 2:30:0 for type 2, and for type 3 5:0 or 2:30:0 for the records `x2xx` and `x3xx`, or `x0xx` and `x1xx`).

-t The type of the input, from 0 to 3, as above (default: 3 if the file name ends in `.ANO` or `.ano`, 0 otherwise).

```
$ a2m -i 1201.ANO -F 250
init: can't open header for record 1201
$ rdann -r 1201 -a atr -F 250 -t 5:3
init: can't open header for record 1201
    5:01.000    75250     N    0    0    0
    5:01.812    75453     N    0    0    0
    5:02.628    75657     V    0    0    0
```

Returns Exit status 0.

Errors Exit status 1 for an error in the arguments or a sampling frequency that is not specified; 2 if the input or the annotation files cannot be opened.

Differences from the C tool (Chapter 4): the new option `-F` (the C tool assumed 250 Hz with a header, or 1 sample per second without one), and annotation times in milliseconds (types 1 to 3) are converted at the record's sampling frequency (C: 4 ms per sample, whatever the frequency); the `-i` file is read from the input directory (C: from the current directory).

m2a

Convert WFDB annotations to AHA format

```
usage: m2a -r RECORD -a MIT-ANNOTATOR AHA-ANNOTATOR [ OPTIONS ... ]
```

Copies an MIT-format annotation file to an AHA-format annotation file of the same record (Section 15.3.2), the annotation types mapped by `mamap`. With `-s`, the annotations before the shift are dropped, and the times of the others are shifted back and converted to milliseconds, as in the AHA tape files.

-r The record, *RECORD* (required).

-a The input (MIT) and output (AHA) annotators, *MIT-ANNOTATOR AHA-ANNOTATOR* (required).

-F The sampling frequency, used if the record header gives none; then required for `-s`.

-h Print the usage summary, and exit with status 1.

-s Shift the annotation times back by the time that follows, and convert them to milliseconds.

```
$ m2a -r 100s -a atr aha
```

Returns Exit status 0 (the file `100s.aha` is written).

Errors Exit status 1 for an error in the arguments or a sampling frequency that is needed and not specified; 2 if an annotation file cannot be opened.

Differences from the C tool (Chapter 4): the new option `-F`; the C tool assumed 250 Hz.

---

ad2m

Convert an AHA-format signal file to a WFDB record

```
usage: ad2m -i AHAFILE [ OPTIONS ... ]
```

Converts the two signals of an AHA Database signal file, in the old tape format (format 16 with an end-of-data marker) or in the compressed format of the CD-ROMs (`-c`), to a WFDB record in format 212: *RECORD*`.dat` and *RECORD*`.hea`.

-i The AHA-format input file, *AHAFILE* (required). It is read from the input directory (Section 14.2.1).

-c The input is compressed (the default if the name ends in `.CMP` or `.cmp`; otherwise the old tape format is assumed).

-f Begin at the time that follows (default 0, the beginning of the input file).

-F The sampling frequency in Hz (required; AHA Database: 250).

-G The gain in adu per physical unit (required; AHA Database: 400; 0 declares the signals uncalibrated).

-h Print the usage summary, and exit with status 1.

-r The output record, *RECORD* (default: the input file name without its directory and suffix).

-R The ADC resolution in bits, from 1 to 32 (required; AHA Database: 12).

-t Stop at the time that follows (default: 35 minutes after the beginning for the records `x2xx` and `x3xx`, 3 hours for `x0xx` and `x1xx`, otherwise the end of the input, with a warning).

-U The physical units (required unless the gain is 0; AHA: `mV`).

```
$ ad2m -i 1201.CMP -F 250 -G 400 -R 12 -U mV
$ cat 1201.hea
1201 2 250 800
1201.dat 212 400/mV 12 0 0 32104 0
1201.dat 212 400/mV 12 0 90 -31586 0
```

Returns Exit status 0.

Errors Exit status 1 for an error in the arguments, a missing `-F`, `-G`, `-R` or `-U`, or an end time not after the start time; 2 if the input cannot be read; 3 if the output signal file cannot be opened.

Differences from the C tool (Chapter 4): the new options `-F`, `-G`, `-R` and `-U`, all required (`-U` unless the gain is 0); the C tool assumed the parameters of the AHA Database (250 Hz, 400 adu/mV, 12

bits, millivolts). The `-f` and `-t` times are converted at the `-F` frequency. The `-i` file is read from the input directory (C: from the current directory).

---

md2a

Convert a record to an AHA-format signal file

```
usage: md2a -o AHAFILE -r RECORD [ OPTIONS ... ]
```

Writes the signals of a record to an AHA-format signal file: format 16, the signals interleaved, followed by an end-of-data marker. With `-n`, it also writes a header that describes the output file as a record.

-o The output file, *AHAFILE* (required).

-r The input record, *RECORD* (required).

-h Print the usage summary, and exit with status 1.

-n Write a header for the output file, as the record *NEWREC* that follows.

```
$ md2a -r 100s -o 100s.ah -n ahrec
$ cat ahrec.hea
ahrec 2 360 21600
100s.ah 16 200 11 1024 995 21537 0
100s.ah 16 200 11 1024 1011 -3962 0
```

Returns Exit status 0.

Errors Exit status 1 for an error in the arguments; 2 if the record cannot be opened; 3 if the header of `-n` cannot be created; 4 if the output file cannot be opened.

Differences from the C tool: none, apart from `--legacy` (the samples of a frame beyond those of the record are 0, where the C tool used uninitialized values).

---

ahaecg2mit

Convert AHA DVD files to WFDB records

```
usage: ahaecg2mit [-s] -F FREQ -G GAIN -R BITS -U UNITS RECORD.XXX [RECORD.XXX]...
```

Converts the files of the AHA Database DVD, in binary (`.ecg`) or text (`.txt`) form, to WFDB records: for each input file *RECORD*`.XXX`, it writes *RECORD*`.hea`, *RECORD*`.dat` (format 212) and *RECORD*`.atr`, and reports `wrote ...` on the standard error. Files with another suffix are ignored, with a message. The options come first, in any order.

-s Make short-format (35-minute) records from long-format (2.5-hour) files: the first 145 minutes are skipped (located from the `-F` frequency), and the second digit of the output record name is incremented by 2.

-F The sampling frequency in Hz (required; AHA DVD: 250).

-G The gain in adu per physical unit (required; AHA DVD: 400).

-R The ADC resolution in bits, from 1 to 32 (required; AHA DVD: 12).

-U The physical units (required unless the gain is 0; AHA DVD: `mV`).

RECORD.XXX
An input file; *RECORD* is a 4-digit AHA Database record name, and *XXX* is `ecg` or `txt`. Several files may be converted in one run. The files are read from the input directory (Section 14.2.1).

```
$ ahaecg2mit -F 250 -G 400 -R 12 -U mV 1001.ecg
Rearranging annotations for output annotator atr ...done!
wrote 1001.atr, 1001.dat, and 1001.hea
$ cat 1001.hea
1001 2 250 700
1001.dat 212 400/mV 12 0 0 1213 0 ECG0
1001.dat 212 400/mV 12 0 300 -1026 0 ECG1
```

Returns Exit status 0 (also when an input file is ignored or cannot be opened).

Errors Exit status 1 without arguments, with `-h` as the first argument (the usage), or for a missing or invalid `-F`, `-G`, `-R` or `-U`.

Differences from the C tool (Chapter 4): the new options `-F`, `-G`, `-R` and `-U`, required (`-U` unless the gain is 0), which precede the input files in any order (the C tool had only `-s`, as the first argument, and assumed 250 Hz, 400 adu/mV, 12 bits and millivolts); the input files are read from the input directory (C: from the current directory).

---

makeid

Write an AHA-format record ID block

```
usage: makeid annotator record
```

Writes the 512-byte ID block of the AHA tape format for a record (with two signals) and an annotator to the standard output: the record ID (8 bytes), the number of annotations, the first and last samples of the annotated segment, the number of 512-byte blocks of samples, the times of the first and last annotations, and the number of blocks of annotations, as 16- and 32-bit integers. It describes these values on the standard error. The record name must have at most 8 characters.

annotator
: The annotator.

record
: The record.

```
$ makeid atr 100s > 100s.id
100s
75 annotations
Annotated segment begins at sample 0
Annotated segment ends at sample 21600
169 blocks of sample data
First annotation is at sample 18
Last annotation is at sample 21423
3 blocks of annotations
```

Returns Exit status 0.

Errors Exit status 1 with fewer than two arguments or a record name longer than 8 characters; 2 if the record or the annotator cannot be opened.

Differences from the C tool: none, apart from `--legacy`.

---

readid

Read an AHA-format record ID block

```
usage: readid data-block-size annot-block-size
```

Reads an ID block (as written by `makeid`) from the standard input, and prints the record ID, the number of data blocks and the number of annotation blocks, recounted in blocks of the given sizes (in bytes).

data-block-size
: The size of a block of samples, in bytes (at least 1).

annot-block-size
: The size of a block of annotations, in bytes (at least 1).

```
$ readid 512 16 < 100s.id
100s 169 96
```

Returns Exit status 0.

Errors Exit status 1 with fewer than two arguments.

Differences from the C tool: none, apart from `--legacy` (a record ID of more than 8 characters, which overflows a buffer in the C tool, is printed whole).

---

revise

Convert an obsolete-format header

```
usage: revise record
```

Reads the obsolete-format header of a record, *record*`.header` (one value per line, as written by early versions of the WFDB library), and writes a header in the current format, *record*`.hea`.

record
: The record (a `header.` prefix is removed).

Returns Exit status 0.

Errors Exit status 1 without an argument; 2 if the old header cannot be read.

Differences from the C tool (Chapter 4): a missing ADC resolution stays unspecified (C: 12 bits).

---

parsescp

Read and convert an SCP-ECG file

```
usage: parsescp -o RECORD [OPTIONS ...] <SCPFILE
```

Reads an SCP-ECG file from the standard input, checks its CRCs, decodes its sections (patient and acquisition data, Huffman tables, leads, QRS locations, reference beats, rhythm data, measurements, diagnoses) and the ECG, derives the missing limb leads, and writes: *RECORD*`.des`, a description (the patient data, acquisition data and measurements), *RECORD*`.key`, the patient's name and medical record number, and *RECORD*`.ecg`, the 12-lead ECG in a binary layout of its own; with `-w`, a WFDB record of 12 signals, *RECORD*`.hea` and *RECORD*`.dat` (format 16, 200 adu/mV); with `-v`, the samples as text, *RECORD*`.txt`, and an analysis of the file on the standard output. All these files are created in the output directory (Section 14.2.1).

-o The record name, *RECORD*, of at most 11 characters (default `ecg`).

-a Anonymize the input and copy it to the standard output (the CRC recomputed); no other output.

-f Parse a file with CRC errors.

-h Print the usage summary, and exit with status 1.

-l Low-pass filter (smooth) the output waveforms.

-v Print a (very) detailed analysis of the input on the standard output, and write *RECORD*`.txt`; given twice, the text file has column headings.

-w Write a WFDB-format record.

-z Make the signals zero-mean.

-b (Debugging) Show the baseline (residuals), without the reference beats.

-s (Debugging) Shift the reference beats by the number of samples that follows before adding them.

-S (Debugging) Skip the parsing of the section whose number follows (- for all sections).

-t (Debugging) Show the reference beats only, without the baseline.

-x (Debugging) Show hex dumps (implies `-v`).

```
$ parsescp -o scp -w < example.scp
$ head -3 scp.hea
scp 12 500 5000     0:22
scp.dat 16 200/mV 16 0 -1 -2500 0 I
scp.dat 16 200/mV 16 0 -3 -2034 0 II
```

Patient data The `.key` file holds the patient's name and medical record number, the `.des` file and the header the age and sex, and `-v` prints all the patient data of the file; `-a` writes a copy of the input with these fields removed.

Returns Exit status 0.

Errors Exit status 1 for an error in the arguments, an input shorter than 6 bytes, a CRC error (without `-f`), or a reference-beat subtraction zone larger than the template; 2 if the declared length of the input cannot be allocated; 3 if the input is shorter than its declared length.

Differences from the C tool (Chapter 4): the record is written with the sampling frequency given in the SCP file, and is not written if the file gives none (`parsescp: sampling frequency not specified; record R not written`); the C tool always wrote 500 Hz. The `.des`, `.ecg`, `.key` and `.txt` files are created in the output directory, as the record is (C: in the current directory). Signals and reference beats that cannot be allocated, and samples beyond the end of a signal shorter than lead 0, are read as absent or as zeros (undefined behavior in the C tool).

CHAPTER 15

Storage formats and annotation codes

This chapter is the reference of the binary files of a record: the storage formats of the signal files, named by the format numbers of the header (Chapter 2), the layout of the annotation files, and the annotation codes with their mnemonics, descriptions and mapping tables. These are the formats of the WFDB library 10.7.0; the files that WFDB-x writes are read by the C library, and those of the C library by WFDB-x, in all seven implementations.

15.1 Signal storage formats

A signal file holds the samples of one *signal group*: the signals of the header that name the same file. The samples are stored frame by frame: for each frame, the samples of the first signal of the group (one, or n for a signal with n samples per frame, written `16x`n in the header), then those of the second signal, and so on. The formats that pack several samples into a group of bytes (212, 310 and 311) pack consecutive samples of this sequence, whatever signals they belong to. The header can also give a byte offset (`16+44`: the samples begin after 44 bytes of prolog, which the library skips) and a skew for each signal (Chapter 2).

Each format represents a range of integer values (ADC units). The most negative value of a format is reserved: it marks an *invalid* sample (a sample that is missing or was not recorded). `putvec` writes the invalid sample `WFDB_INVALID_SAMPLE` (-32768) as that value, and `getvec` reads that value back as `WFDB_INVALID_SAMPLE` (or, in the `WFDB_GVPAD` mode of `setgvmode`, as the previous valid sample of the signal). A sample outside the range of the format is written modulo 2^{bits} (its lowest bits), except in format 8, which limits the change from one sample to the next. The formats are listed by the library constant `WFDB_FMT_LIST`, in the order of Table 15.1; every implementation reads and writes all of them.

Table 15.1 The signal storage formats (bits per sample, the bytes of a group of samples, the range of the values, and whether WFDB-x reads (R) and writes (W) them).

Format	Bits	Layout	Range (invalid value)	R/W
0	–	no signal file (a null signal)	every sample invalid	R/W
8	8	1 byte: the difference from the previous sample, two's complement	differences -128 to 127 (no invalid value)	R/W
16	16	2 bytes, two's complement, little-endian	-32768 to 32767 (-32768)	R/W
61	16	2 bytes, two's complement, big-endian	-32768 to 32767 (-32768)	R/W
80	8	1 byte, offset binary (value + 128)	-128 to 127 (-128)	R/W
160	16	2 bytes, offset binary (value + 32768), little-endian	-32768 to 32767 (-32768)	R/W
212	12	2 samples in 3 bytes	-2048 to 2047 (-2048)	R/W
310	10	3 samples in two 16-bit little-endian words (15 bits of each)	-512 to 511 (-512)	R/W

Format	Bits	Layout	Range (invalid value)	R/W
311	10	3 samples in one 32-bit little-endian word (30 bits)	-512 to 511 (-512)	R/W
24	24	3 bytes, two's complement, little-endian	-2^{23} to $2^{23}-1$ (-2^{23})	R/W
32	32	4 bytes, two's complement, little-endian	-2^{31} to $2^{31}-1$ (-2^{31})	R/W
508	8	FLAC stream, 8 bits per sample	-128 to 127 (-128)	R/W
516	16	FLAC stream, 16 bits per sample	-32768 to 32767 (-32768)	R/W
524	24	FLAC stream, 24 bits per sample	-2^{23} to $2^{23}-1$ (-2^{23})	R/W

The ADC resolution given in the header (Chapter 2) may be smaller than the storage width of the format (11-bit samples in format 212, as in `100s`); the format only has to hold the range of the values. The precise layout of each format follows. In the bit diagrams, bit 0 is the least significant bit, and s_0, s_1, s_2 are consecutive samples of the group.

15.1.1 Format 0: null signals

A signal of format 0 has no samples on disk (its file name in the header is usually ~). Every sample read is invalid (`WFDB_INVALID_SAMPLE`, or, in `WFDB_GVPAD` mode, the signal's initial value from the header); samples written to it are discarded. Format 0 is used by the layout segment and the null segments of multi-segment records.

15.1.2 Format 8: first differences

Each byte is the difference between a sample and the previous sample of the same signal, as a signed 8-bit integer; the first difference is taken from the initial value given in the header. The samples themselves can take any value, but they change by -128 to 127 from one sample to the next: `putvec` limits a larger difference to -128 or 127 and returns 0 (the slew rate is too high), and the following samples are then relative to the value actually written. The format has no invalid value, and an error in the file affects every following sample.

15.1.3 Formats 16 and 61: 16-bit amplitudes

Each sample is a 16-bit two's complement integer, least significant byte first (format 16) or most significant byte first (format 61).

15.1.4 Formats 80 and 160: offset binary

Each sample is stored as an unsigned integer, the value plus 2^{bits-1}: one byte, the value plus 128 (format 80), or two bytes, the value plus 32768, least significant byte first (format 160). Format 80 is that of 8-bit WAV files (`wav2mit`).

15.1.5 Format 212: 12-bit amplitudes

Two samples are packed into three bytes $b_0 b_1 b_2$:

- s_0: the low 8 bits in b_0, the high 4 bits in bits 0–3 of b_1;
- s_1: the high 4 bits in bits 4–7 of b_1, the low 8 bits in b_2.

Each sample is a 12-bit two's complement integer. When the number of samples written is odd, the last sample is written as the first two bytes of a group ($b_0 b_1$).

15.1.6 Format 310: 10-bit amplitudes in 16-bit words

Three samples are packed into two 16-bit little-endian words w_0 and w_1, bit 0 of each word being unused (0):

- s_0: bits 1–10 of w_0;
- s_1: bits 1–10 of w_1;
- s_2: its low 5 bits in bits 11–15 of w_0, its high 5 bits in bits 11–15 of w_1.

Each sample is a 10-bit two's complement integer. An incomplete group at the end of the file is written as the words that hold its samples (w_0 for one sample, $w_0 w_1$ for two).

15.1.7 Format 311: 10-bit amplitudes in 32-bit words

Three samples are packed into one 32-bit little-endian word w (four bytes), bits 30 and 31 being unused (0):

- s_0: bits 0–9 of w;
- s_1: bits 10–19;
- s_2: bits 20–29.

Each sample is a 10-bit two's complement integer. An incomplete group at the end of the file is written as its first two bytes (one sample) or its four bytes (two samples).

15.1.8 Formats 24 and 32: 24- and 32-bit amplitudes

Each sample is a two's complement integer of 3 bytes (format 24) or 4 bytes (format 32), least significant byte first. In format 32, the invalid value -2^{31} differs from `WFDB_INVALID_SAMPLE`: an invalid sample is written as -2^{31}, which is read back as `WFDB_INVALID_SAMPLE`.

15.2 FLAC formats

Formats 508, 516 and 524 store the samples of a signal group losslessly compressed, as a FLAC stream (Free Lossless Audio Codec) of $format - 500$ bits per sample (8, 16 or 24). Each signal of the group is a channel of the stream, so a FLAC file holds at most 8 signals, and all of them must have the same number of samples per frame (the library reports `cannot store N signals in a single FLAC file` or `every signal in a FLAC file must have the same sampling frequency`). The samples of a frame of a signal with several samples per frame are consecutive samples of its channel. The FLAC sample rate is nominal (96 kHz); the sampling frequency is the one given in the header.

The C library reads and writes these formats with libFLAC. WFDB-x needs no external library:

- **Reading** uses a translation of the stream decoder of libFLAC 1.4.3, in each implementation: every subframe type (constant, verbatim, fixed and linear prediction), stereo decorrelation, wasted bits, variable block sizes and CRC checks. Damaged files are decoded, resynchronized and reported as by the C library, and seeking (`isigsettime`) works as in C. The MD5 signature of the stream is checked when the input is closed if the environment variable `WFDB_FLAC_CHECK_MD5` is set (`isigclose: warning: incorrect MD5 hash in FLAC input`).
- **Writing** uses an encoder of WFDB-x's own: the stream begins with `fLaC`, a `STREAMINFO` block (block size 4096, the frame sizes, the number of channels and bits per sample, the total number of samples and the MD5 signature, all completed when the file is closed) and an empty `VORBIS_COMMENT` block (vendor `wfdb-x`). The samples are coded in frames of 4096 samples per channel (the last frame shorter), the channels independently, each as a

constant, verbatim or fixed-predictor (order 0 to 4) subframe, whichever is smallest, with Rice-coded residuals (partition orders 0 to 6). A sample is first reduced to the width of the format.

The files written are standard FLAC streams: libFLAC, and therefore the C library, reads them back with identical sample values. Their bytes differ from those that libFLAC writes (which depend on its version and on the processor), so a record written in a FLAC format by WFDB-x has the same samples and checksums as one written by the C library, but not the same signal file.

15.3 Annotation files

An annotation file belongs to a record and an annotator: `100s.atr` holds the annotations of the annotator `atr` for the record `100s`. Each annotation has a time (a sample number, or a number of ticks of the file's time resolution), an annotation code (`anntyp`, Section 15.4), three small integer fields (`subtyp`, `chan` and `num`) and optional auxiliary information (`aux`, a string of up to 255 bytes). The library reads two formats, MIT (the WFDB format, the default for writing) and AHA; it recognizes the format of a file when it opens it for reading.

15.3.1 MIT format

An MIT-format annotation file is a sequence of 16-bit words, each stored least significant byte first. The 6 high bits of a word are a code A (`(w >> 10) & 63`), and the 10 low bits a value I (`w & 1023`). An annotation is an *annotation word*, whose code A (1 to 49) is the annotation code and whose value I is the interval, in ticks, from the time of the previous annotation (from 0 for the first), followed by any number of *pseudo-annotation words* that modify it, with the codes 59 to 63 of Table 15.2. The 32-bit integers of the file are stored as two 16-bit words, the most significant word first, each least significant byte first (the byte order of the PDP-11: the bytes 2, 3, 0, 1 of a little-endian integer). The word 0 ($A = 0$, $I = 0$) ends the file.

Table 15.2 The pseudo-annotation codes of MIT-format annotation files.

A	Name	Meaning
59	`SKIP`	I is 0; the next two words are a signed 32-bit interval, added to the time. It precedes an annotation word whose interval does not fit in 10 bits (more than 1023 ticks, or a negative interval, in a file that is out of order), which then has the interval 0. Intervals beyond $\pm(2^{31}-1)$ take several `SKIP`s.
60	`NUM`	I is the `num` field of this annotation and of the following ones (written only when it changes; initially 0).
61	`SUB`	I is the `subtyp` field of this annotation only (written only when it is not 0).
62	`CHN`	I is the `chan` field of this annotation and of the following ones (written only when it changes; initially 0).
63	`AUX`	The low 8 bits of I are a byte count n; the next n bytes are the `aux` string of this annotation, followed by a null byte if n is odd, so that the file stays word-aligned.

The `subtyp` and `num` fields are signed 8-bit integers and `chan` an unsigned 8-bit integer, stored in the 10 bits of I. Codes 50 to 58 are not used. The codes 59 to 63 are reserved for the pseudo-annotation words, and cannot be annotation codes: an annotation word with one of them would be read as a control word, and would corrupt the rest of the file. The times `WFDB_TIME_MIN` and `WFDB_TIME_MAX` (-2^{63} and $2^{63}-1$) cannot be written either. The library function `putann` refuses these times (`time overflow`) but does not check the code, as in C (a code above 63 also loses its high bits); the high-level `write_annotations` accepts only the codes 0 to 58 and

refuses the others, and these times, before it writes anything (Section 4.8). An annotation of code 0 (NOTQRS), which a word of value 0 could not represent, is written as a SKIP of its interval minus 1 followed by a word of code 0 and interval 1.

At the beginning of a file, NOTE annotations at time 0 (with subtyp, chan and num 0) whose aux begins with ## carry information for the library, written by putann before the first annotation and read by annopen:

- ## time resolution: *F*: the times of the file are in ticks of *F* per second, not in sample intervals of the record (written when setafreq has set a resolution, as wrann does with the sampling frequency of the record; getiaorigfreq returns it);
- ## annotation type definitions, then one annotation per code whose mnemonic or description has been redefined (by setannstr or setanndesc with a positive code), with the aux *code mnemonic description*, then ## end of definitions. When it reads such a file, the library takes these definitions as the mnemonics and descriptions of annstr, strann and anndesc;
- after these, an annotation of code 0 at time 0 ends the table.

An MIT-format file written by putann need not be in canonical order (by time, then num, then chan); WFDB-x sorts an out-of-order file itself when it is closed, without losing annotations (Chapter 4), unless the environment variable WFDBANNSORT is 0.

15.3.2 AHA format

An AHA-format annotation file, as used by the AHA Database and written for it by m2a, has 16 bytes per annotation:

Table 15.3 An annotation of an AHA-format annotation file.

Bytes	Contents
0	0
1	the AHA annotation code, a character (Table 15.6); mapped to an annotation code by ammap when read, and from one by mamap when written
2–5	the time, a 32-bit integer in PDP-11 order (most significant word first)
6–7	the serial number of the annotation, a 16-bit little-endian integer, from 1
8	the subtyp field
9	the annotation code (anntyp) when written; read as the chan field
10–15	the aux string, up to 6 bytes, padded with null bytes

The file is padded with bytes 255 (0377) to a multiple of 1024 bytes; a byte 255 in place of the first byte of an annotation ends it. When it reads an AHA file, the library gives an AHA code U with a subtyp of 0 the subtyp -1, and mamap maps NOISE to U only with that subtype. A file opened for reading is taken to be in AHA format when its first byte is 0 and its second byte is an AHA code other than [and]; the library warns (file appears to be in AHA format) when the format it finds is not the one requested. The times are in the units written: sample intervals for files written by m2a without -s, and milliseconds in the AHA Database tape files (a2m -t 1 and -t 2). The compressed annotation files of the AHA Database CD-ROMs (.ANO), read only by a2m (-t 3), have 6 bytes per annotation: the AHA code, a 24-bit time in milliseconds (most significant byte first; 255 in its high byte marks the end), and a 16-bit serial number.

15.4 Annotation codes

Annotation codes are integers from 1 to ACMAX (49); 0, NOTQRS, is not an annotation code (isann is false for it), and codes 42 to 49 have no standard meaning (they can be defined by setannstr and setanndesc, and by the definitions of an annotation file). Each code has a symbolic name (a constant of the library: NORMAL, PVC, ...), a mnemonic (the text of rdann, returned by annstr and ecgstr) and a description (anndesc). Table 15.4 gives the standard values, those of the library when no annotation file has redefined them.

Table 15.4 The annotation codes: number, symbolic name, mnemonic and description.

Code	Name	Mnemonic	Description
0	NOTQRS	(space)	(not an annotation code)
1	NORMAL	N	Normal beat
2	LBBB	L	Left bundle branch block beat
3	RBBB	R	Right bundle branch block beat
4	ABERR	a	Aberrated atrial premature beat
5	PVC	V	Premature ventricular contraction
6	FUSION	F	Fusion of ventricular and normal beat
7	NPC	J	Nodal (junctional) premature beat
8	APC	A	Atrial premature beat
9	SVPB	S	Supraventricular premature or ectopic beat
10	VESC	E	Ventricular escape beat
11	NESC	j	Nodal (junctional) escape beat
12	PACE	/	Paced beat
13	UNKNOWN	Q	Unclassifiable beat
14	NOISE	~	Change in signal quality
15	–	[15]	(none)
16	ARFCT	|	Isolated QRS-like artifact
17	–	[17]	(none)
18	STCH	s	ST segment change
19	TCH	T	T-wave change
20	SYSTOLE	*	Systole
21	DIASTOLE	D	Diastole
22	NOTE	"	Comment annotation
23	MEASURE	=	Measurement annotation
24	PWAVE	p	P-wave peak
25	BBB	B	Bundle branch block beat (unspecified)
26	PACESP	^	(Non-captured) pacemaker artifact
27	TWAVE	t	T-wave peak
28	RHYTHM	+	Rhythm change
29	UWAVE	u	U-wave peak
30	LEARN	?	Beat not classified during learning
31	FLWAV	!	Ventricular flutter wave
32	VFON	[	Start of ventricular flutter/fibrillation
33	VFOFF	]	End of ventricular flutter/fibrillation
34	AESC	e	Atrial escape beat
35	SVESC	n	Supraventricular escape beat
36	LINK	@	Link to external data (aux contains URL)

Code	Name	Mnemonic	Description
37	NAPC	x	Non-conducted P-wave (blocked APC)
38	PFUS	f	Fusion of paced and normal beat
39	WFON, PQ	(	Waveform onset
40	WFOFF, JPT	)	Waveform end
41	RONT	r	R-on-T premature ventricular contraction
42	–	[42]	(none)
43	–	[43]	(none)
44	–	[44]	(none)
45	–	[45]	(none)
46	–	[46]	(none)
47	–	[47]	(none)
48	–	[48]	(none)
49	–	[49]	(none)

15.4.1 Mapping tables

The library classifies the annotation codes with tables that programs can query and change (Chapter 13): `isqrs(a)` is 1 for the codes of QRS complexes (beats); `map1(a)` maps a code to one of NOTQRS, NORMAL, PVC, FUSION and LEARN; `map2(a)` to one of those or SVPB; `mamap(a, s)` gives the AHA code (a character) of an annotation code with the subtype `s`; and `annpos(a)` the position at which an annotation is drawn (APSTD at the signal, APHIGH above it, APLOW below, APUNDEF undefined; the position codes APATT (4, attached to the signal given by `chan`), APAHIGH (5) and APALOW (6, above and below it) are defined but used by no standard code). `setisqrs`, `setmap1`, `setmap2` and `setannpos` change the tables. A code out of range gives 0, NOTQRS, NOTQRS, `O` and APUNDEF. Table 15.5 gives the tables for the codes 0 to 41 (codes 42 to 49 give 0, NOTQRS, NOTQRS, `O` and APUNDEF).

Table 15.5 The mapping tables of the annotation codes 0 to 41 (`mamap` of code 14 is `U` with the subtype -1, `O` otherwise).

Code	Mnemonic	isqrs	map1	map2	mamap	annpos
0		0	NOTQRS	NOTQRS	O	APUNDEF
1	N	1	NORMAL	NORMAL	N	APSTD
2	L	1	NORMAL	NORMAL	N	APSTD
3	R	1	NORMAL	NORMAL	N	APSTD
4	a	1	NORMAL	SVPB	N	APSTD
5	V	1	PVC	PVC	V	APSTD
6	F	1	FUSION	FUSION	F	APSTD
7	J	1	NORMAL	SVPB	N	APSTD
8	A	1	NORMAL	SVPB	N	APSTD
9	S	1	NORMAL	SVPB	N	APSTD
10	E	1	PVC	PVC	E	APSTD
11	j	1	NORMAL	NORMAL	N	APSTD
12	/	1	NORMAL	NORMAL	P	APSTD
13	Q	1	NORMAL	NORMAL	Q	APSTD
14	~	0	NOTQRS	NOTQRS	U	APHIGH
15	[15]	0	NOTQRS	NOTQRS	O	APUNDEF

Code	Mnemonic	isqrs	map1	map2	mamap	annpos
16	|	0	NOTQRS	NOTQRS	O	APHIGH
17	[17]	0	NOTQRS	NOTQRS	O	APUNDEF
18	s	0	NOTQRS	NOTQRS	O	APHIGH
19	T	0	NOTQRS	NOTQRS	O	APHIGH
20	*	0	NOTQRS	NOTQRS	O	APHIGH
21	D	0	NOTQRS	NOTQRS	O	APHIGH
22	"	0	NOTQRS	NOTQRS	O	APHIGH
23	=	0	NOTQRS	NOTQRS	O	APHIGH
24	p	0	NOTQRS	NOTQRS	O	APHIGH
25	B	1	NORMAL	NORMAL	N	APSTD
26	^	0	NOTQRS	NOTQRS	O	APHIGH
27	t	0	NOTQRS	NOTQRS	O	APHIGH
28	+	0	NOTQRS	NOTQRS	O	APLOW
29	u	0	NOTQRS	NOTQRS	O	APHIGH
30	?	1	LEARN	LEARN	Q	APSTD
31	!	1	PVC	PVC	O	APSTD
32	[	0	NOTQRS	NOTQRS	[	APSTD
33	]	0	NOTQRS	NOTQRS	]	APSTD
34	e	1	NORMAL	NORMAL	N	APSTD
35	n	1	NORMAL	NORMAL	N	APSTD
36	@	0	NOTQRS	NOTQRS	O	APHIGH
37	x	0	NOTQRS	NOTQRS	O	APHIGH
38	f	1	NORMAL	NORMAL	N	APSTD
39	(	0	NOTQRS	NOTQRS	O	APHIGH
40	)	0	NOTQRS	NOTQRS	O	APHIGH
41	r	1	PVC	PVC	R	APSTD

`ammap(c)`, the reverse of `mamap`, maps an AHA code (the character `c`, from `E` to `]`) to an annotation code, and gives `NOTQRS` for any other character:

Table 15.6 The AHA annotation codes and the annotation codes that `ammap` gives them.

AHA code	ammap
E	VESC
F	FUSION
G	NOTQRS
H	NOTQRS
I	NOTQRS
J	NOTQRS
K	NOTQRS
L	NOTQRS
M	NOTQRS
N	NORMAL
O	NOTE
P	PACE

AHA code	ammap
Q	UNKNOWN
R	RONT
S	NOTQRS
T	NOTQRS
U	NOISE
V	PVC
W	NOTQRS
X	NOTQRS
Y	NOTQRS
Z	NOTQRS
[	VFON
\	NOTQRS
]	VFOFF