WFDB_tools
A Matlab interface to the WFDB library

Jonas Carlson

June 2, 2005

The most recent versions of the software described here may be freely downloaded from PhysioNet (http://www.physionet.org/). An HTML version of this guide is available at http://www.physionet.org/physiotools/matlab/wfdb_tools/.

Contents

Preface iii

1 Installing the WFDB_tools wrappers 1
- 1.1 Preparation 1
- 1.2 Installing WFDB_tools in a user directory 2
- 1.3 Installing WFDB_tools as a MATLAB toolbox 2
- 1.4 The WFDB library and applications 2
- 1.5 Sample, signal, and annotator numbers 2

2 Using the WFDB_tools library 3
- 2.1 Reading signals 3
- 2.2 Reading annotations 4
- 2.3 Creating an annotation file 6
- 2.4 Creating a signal file 7

3 WFDB_tools library functions 11
- 3.1 Selecting Database Records 11
- 3.2 Special Input Modes 14
- 3.3 Reading and Writing Signals and Annotations 15
- 3.4 Non-Sequential Access to WFDB Files 17
- 3.5 Conversion Functions 18
- 3.6 Calibration Functions 21
- 3.7 Miscellaneous Functions 22
- 3.8 Creating structures 27

Preface

The Waveform Database interface library (the WFDB library) is a package of C-callable functions that provide clean and uniform access to digitized, annotated signals stored in a variety of formats. These functions, although originally designed for use with databases of electrocardiograms, are useful for dealing with any similar collection of digitized signals, which may or may not be annotated. The WFDB library has evolved to support the development of numerous databases that include signals such as blood pressure, respiration, oxygen saturation, EEG, as well as ECGs. Thus the WFDB library is considerably more than an ECG database interface.

This guide documents WFDB_tools, a set of MATLAB functions that enables the MATLAB user to take full advantage of the WFDB library and explore or create databases containing a wide variety of signals.

The WFDB_tools functions are not self-contained; rather, they are 'wrappers' for the WFDB library functions (i.e. the WFDB library must be installed for the MATLAB functions to work). The wrappers work much in the same way as the WFDB library functions, and the effort is to keep them as true to their counterparts as possible. The usual way to work in MATLAB is to get all results from one function, and therefore the wrappers may seem 'low-level' in comparison. Combining just a few of the wrappers in an m-file, however, would produce the 'high-level' way of working in MATLAB while keeping the full control of data handling that can only be obtained from the 'low-level' C-like wrappers.

This guide includes several short tutorial examples that illustrate how to read and write signals and annotations using these wrappers. It also contains descriptions of all the wrapper functions available to the MATLAB user. Note that MATLAB help files for all of the wrappers are included in the WFDB_tools package, so it is always possible within MATLAB to use a command such as

```
help WFDB_sampfreq
```

to obtain information about how to use any of these wrappers.

The set of wrappers is nearly complete. WFDB library functions for which no wrappers currently exist are wfdbinit, ungetann, sample, sample_valid, setannstr, setanndesc, setecgstr, calopen, getcal, putcal, newcal, flushcal, setmsheader, setwfdb, setibsize, and setobsize.

Chapter 1

Installing the WFDB_tools wrappers

Important: the WFDB_tools wrappers have been developed and tested with MATLAB *R13.* It is highly unlikely that they can be made to work with an older version of MATLAB. Initial development and testing was under Red Hat Linux 8.0, using WFDB library version 10.3.2; the wrappers have also been tested under Red Hat Linux 9.0 and WFDB library version 10.3.11. More recently, they have also been tested under MS-Windows XP, also with WFDB library version 10.3.11. It may be possible to recompile these wrappers and to use them with MATLAB R13 on other platforms such as Mac OS/X or Solaris, but doing so is currently unsupported.

1.1 Preparation

Before installing the WFDB_tools wrappers, install the WFDB Software Package (http://www.physionet.org/physiotools/wfdb.shtml), and verify that it is working properly on your system.

Download the WFDB_tools package (http://www.physionet.org/physiotools/matlab/WFDB_tools.tar.gz) and unpack it using a command such as

```
tar xfvz WFDB_tools.tar.gz
```

This creates a directory named `WFDB_tools`, which contains the wrapper source files (`src/*.c`), help files (`help/*.m`), precompiled binary files (`linux/*.mexglx`, for x86 GNU/Linux, and `windows/*.dll`, for MS-Windows), documentation (in `doc/`), and tutorial examples (in `examples/`).

Follow the instructions given in the `00README` file (in the top-level `WFDB_tools` directory) to install the WFDB_tools on your system.

All functions have explanatory m-files available through MATLAB's `help` function. In the case of toolbox installation (described below) a list of all functions is found using `help WFDB_tools`.

1.2 Installing WFDB_tools in a user directory

The directory `WFDB_tools` can be put anywhere. All MATLAB programs must then use the command `addpath` to make the directory available (it is probably not a good idea to put other program files inside the `WFDB_tools` directory).

1.3 Installing WFDB_tools as a MATLAB toolbox

With sufficient write privileges, it should be possible to install the WFDB tools as a MATLAB toolbox. It should then be put in the directory:
matlabroot/toolbox/WFDB_tools
To make it available to MATLAB, this directory must be added to `pathdef.m` (in the `toolbox/local` directory) and, if the toolbox path cache is enabled, the command `rehash toolboxcache` should be issued when MATLAB is started.

1.4 The WFDB library and applications

The *WFDB Programmer's Guide* (http://www.physionet.org/physiotools/wpg/), which documents the C-language WFDB library, is recommended as a source of additional information and examples. The *WFDB Applications Guide* (http://www.physionet.org/physiotools/wag/) describes many stand-alone programs that use the WFDB library to read and write digitized signals and annotations. If the WFDB Software Package has been correctly installed, you can run these programs from a terminal window or from within MATLAB to perform a wide variety of signal processing and analysis tasks.

1.5 Sample, signal, and annotator numbers

Several WFDB library functions, and most of the stand-alone WFDB applications, accept arguments that specify a specific sample within a digitized signal (a *sample number*, a specific signal within a set of signals (a *signal number*, or a specific set of annotations (an *annotator number*). The first sample number in a signal has sample number 0, not 1; similarly, the first signal has signal number 0, and the first annotator has annotator number 0. The WFDB_tools functions use the same zero-based sample, signal, and annotator numbers as the WFDB library functions that they wrap. This point is a possible source of confusion if you become accustomed to thinking of these numbers as array indices (which, in C, is exactly what they are); it may be best to think of them simply as identification numbers for the objects with which they are associated.

Chapter 2

Using the WFDB_tools library

Using simple examples, this chapter illustrates how to use the WFDB_tools wrappers to read and write signals and annotations. Additional information about the wrappers in these examples, and about the other wrappers in the library, can be found in the next chapter.

2.1 Reading signals

Assuming that the WFDB Software Package has been installed correctly, the record "100s" should be available. Reading the first ten samples of this record using MATLAB would be done as:

```
>> S = WFDB_isigopen('100s')
S =
2x1 struct array with fields:
    fname
    desc
    units
    gain
    initval
    group
    fmt
    spf
    bsize
    adcres
    adczero
    baseline
    nsamp
    cksum
```

```
>> DATA = WFDB_getvec(length(S), 10)
DATA =
         995        1011
         995        1011
         995        1011
         995        1011
         995        1011
         995        1011
         995        1011
         995        1011
        1000        1008
         997        1008
```

The first command, `S = WFDB_isigopen('100s')`, reads the header file of record 100s and returns the information in a structure (`S`, in this case). The length of `S` equals the number of signals in the data file. The fields of `S` contain the signal settings. To access, for example, the gain of signal 0 and the description of signal 1, use the commands:

```
>> S(1).gain
ans =
   200
>> S(2).desc
ans =
V5
```

(Remember: the first signal is signal 0, not signal 1! Its attributes are found in the first structure, `S(1)`. This will matter in later examples.)

The second command, `DATA = WFDB_getvec(length(S), 10)`, reads data from the previously opened record. The two input parameters are the number of signals (found as `length(S)`) and the desired number of samples (if omitted, the whole record is read).

Finally, don't forget

```
>> WFDB_wfdbquit
```

to close all open files.

An elaborated version of this example is provided in the `examples` directory of the WFDB_tools package (look for `example1.m`).

2.2 Reading annotations

This example illustrates how to open an annotation file, how to read annotations from it, and how to translate them into their mnemonic and description strings.

First, we need to create an 'Anninfo' structure containing the annotator name and mode of the annotation file:

```
>> A = WFDB_Anninfo(1)
A =
    name: 'a1'
    stat: 'WFDB_READ'
```

This might seem like a complicated way to go, but it reflects the way the underlying WFDB library works. The record 100s has an annotator named `atr`, so we need to change the `name` field of `A` before issuing the command to open the file.

```
>> A.name = 'atr'
A =
    name: 'atr'
    stat: 'WFDB_READ'
>> WFDB_annopen('100s', A)
```

Now that the annotation file is open, we may read the first two annotations, and take a closer look at the second one.

```
>> ANNOTATION = WFDB_getann(0, 2)
ANNOTATION =
2x1 struct array with fields:
    time
    anntyp
    subtyp
    chan
    num
    aux
>> ANNOTATION(2)
ans =
      time: 77
    anntyp: 1
    subtyp: 0
      chan: 0
       num: 0
       aux: ''
```

The first argument of WFDB_getann is the input annotator number. Since we are reading only one annotation file, its annotator number is 0. (If we had opened a second annotation file for reading, its annotator number would be 1, etc.) Next, let's see what the annotation type (the `anntyp` field) means, in mnemonic and description:

```
>> WFDB_annstr(ANNOTATION(2).anntyp)
ans =
N
>> WFDB_anndesc(ANNOTATION(2).anntyp)
ans =
Normal beat
```

Finally,

```
>> WFDB_wfdbquit
```

An elaborated version of this example is provided in the `examples` directory of the WFDB_tools package (look for `example2.m`).

2.3 Creating an annotation file

In this example, we will create an annotation file that annotates the first two P-waves of record 100s. These are located at (roughly) sample numbers 315 and 610. First, let's find the annotation code for a P-wave:

```
>> WFDB_strann('p')
ans =
    24
```

The description is:

```
>> WFDB_anndesc(24)
ans =
P-wave peak
```

Create the two annotations:

```
>> ANN = WFDB_Annotation(2)
ANN =
1x2 struct array with fields:
    time
    anntyp
    subtyp
    chan
    num
    aux
>> ANN(1).time = 315;
>> ANN(1).anntyp = 24;
>> ANN(1).aux = 'First P-wave';
>> ANN(2).time = 610;
>> ANN(2).anntyp = 24;
>> ANN(2).aux = 'Second P-wave';
```

Now create an annotator structure:

```
>> A = WFDB_Anninfo(1)
A =
    name: 'a1'
    stat: 'WFDB_READ'
>> A.name = 'p';
>> A.stat = 'WFDB_WRITE';
```

Create an empty annotation file to hold the annotations:

```
>> WFDB_annopen('100s', A)
```

Write the annotations:

```
>> WFDB_putann(0, ANN)
```

The first argument of WFDB_putann is the output annotator number. Since we are writing only one annotation file, its annotator number is 0.
Close all open files:

```
>> WFDB_wfdbquit
```

The new annotation file, `100s.p`, will be located in MATLAB's current directory. The result may be verified using WAVE, which can be called from MATLAB using:

```
>> !wave -r 100s -a p &
```

(Remember the !-sign to call system functions from MATLAB and the &-sign to run WAVE as a background process; depending on your setup, using & may cause an error, however, and you may need to run WAVE as a foreground process without the final '&' in the command.)

An elaborated version of this example is provided in the `examples` directory of the WFDB_tools package (look for `example3.m`).

2.4 Creating a signal file

Creating an output signal file is made in three steps: create a signal information structure, write the output signal data, and create the header file.

Assume we have data from three signals. We need to create a signal information structure using:

```
>> S = WFDB_Siginfo(3)
S =
1x3 struct array with fields:
    fname
    desc
    units
    gain
    initval
    group
    fmt
    spf
    bsize
    adcres
    adczero
    baseline
```

Now these fields need to be filled with appropriate values. All of them have default values to avoid producing errors, but it is unlikely that they will fit our signals. For example, if signal 0 is the X-lead of a Frank-lead ECG, we may want its description to be:

```
>> S(1).desc = 'Frank X';
```

and so on for all other fields. (Remember: the first signal is signal 0; its attributes are in `S(1)`.) When done, we create an empty signal file in which to write the data, using

```
>> WFDB_osigfopen(S)
```

We also need to supply the sampling frequency, for example 1 kHz:

```
>> WFDB_setsampfreq(1000);
```

and the basetime of the recording (i.e. the time of sample number 0). Assuming the recording was started when my oldest daughter was born:

```
>> WFDB_setbasetime('02:19:00 02/09/1999')
```

(Dates used by WFDB_tools are always in DD/MM/YYYY format; 02/09/1999 is 2 September, not February 9.)

Now we're done with providing signal information and it is time to write the actual signal data. This must be stored column-wise in a matrix (one signal per column, one sample per row). If our data is stored in the variable `DATA` we would use:

```
>> WFDB_putvec(DATA)
```

Finally, we need to record the information from `S` into a header (`.hea`) file for later use. We need to choose a name for the new record we are creating (avoiding the names of any existing records that we wish to read in the future); if we choose `test1` as the record name, we can create the header file by:

```
>> WFDB_newheader('test1')
```

These operations will create a header file, `test1.hea`, in the current directory, and a signal file with the name previously specified in the `fname` fields of the signal information structure, `S`. (Notice that `WFDB_newheader` must always be invoked *after* all of the samples have been written using `WFDB_putvec`, because the header file includes the number of samples in the record and checksums for each signal, which are not known by the WFDB library until all of the samples have been written.) As always, we use

```
>> WFDB_wfdbquit
```

to flush all pending output to the files, to close them, and to reset the internal WFDB library variables.

As in the previous example, we can inspect our results using WAVE:

```
>> !wave -r test1 &
```

An elaborated version of this example is provided in the `examples` directory of the WFDB_tools package (look for `example4.m`).

Chapter 3

WFDB_tools library functions

This chapter describes the functions that are provided by the WFDB_tools package. The arrangement of this chapter parallels that of chapter 2 of the *WFDB Programmer's Guide*, which describes the underlying WFDB library functions that the WFDB_tools functions use.

3.1 Selecting Database Records

These functions are used to open input and output signal and annotation files.

WFDB_annopen

Usage: `WFDB_annopen(RECORD, ANNINFO)`
Input: `RECORD`: (string) record name
`ANNINFO`: annotator information structure(s)

This function opens input and output annotation files for a selected record. If RECORD begins with '+', previously opened annotation files are left open, and the record name is taken to be the remainder of RECORD after discarding the '+'. Otherwise, WFDB_annopen closes any previously opened annotation files, and takes all of RECORD as the record name. ANNINFO is a structure array created by WFDB_Anninfo (see section 3.8), with one array element for each annotator to be opened. The caller must fill in the WFDB_Anninfo structure array to specify the names of the annotators, and to indicate which annotators are to be read, and which are to be written. Input and output annotators may be listed in any order in ANNINFO. Annotator numbers (for both input and output annotators) are assigned in the order in which the annotators appear in ANNINFO (the first annotator is number 0). For example, these instructions

```
>> a = WFDB_Anninfo(3);
```

```
>> a(1).name = 'a'; a(1).stat = 'WFDB_READ';
>> a(2).name = 'b'; a(2).stat = 'WFDB_WRITE';
>> a(3).name = 'c'; a(3).stat = 'WFDB_READ';
>> WFDB_annopen('100s', a)
```

attempt to open three annotation files for record '100s'. Annotator 'a' becomes input annotator 0, 'b' becomes output annotator 0, and 'c' becomes input annotator 1. Thus WFDB_getann(1) (see section 3.3) will read all annotations from annotator 'c', and WFDB_putann(0, ANN) (see section 3.3) will write an annotation for annotator 'b'. Input annotation files will be found if they are located in any of the directories in the WFDB path (see section 3.7). Output annotators are created in the current directory (but note that, under Unix at least, it is possible to specify annotator names such as '/here' or 'zzz/there' or even '../somewhere/else')

See also: WFDB_Anninfo (3.8), WFDB_getann (3.3), WFDB_putann (3.3)

WFDB_isigopen

Usage: `S = WFDB_isigopen(RECORD)`
Input: `RECORD`: (string) record name
Output: `S`: Siginfo structure(s)

This function opens input signal files for a selected record. If RECORD begins with '+', previously opened input signal files are left open, and the record name is taken to be the remainder of RECORD after discarding the '+'. Otherwise, WFDB_isigopen closes any previously opened input signal files, and takes all of RECORD as the record name. S is an array of WFDB_Siginfo structures (see section 3.8 for an explanation of the fields), one for each signal that was opened.

Calling WFDB_isigopen also sets internal WFDB library variables that record the base time and date, the length of the record, and the sampling and counter frequencies, so that time conversion functions such as WFDB_strtim (see section 3.5) that depend on these quantities will work properly.

WFDB_isigopen will fill the structure array S with information about the signals in the order in which signals are specified in the 'hea' file for the record. Signal numbers begin with 0, so that fields in S(n) are the attributes of signal (n-1). For example, the gain attributes of each signal in record '100s' can be read like this:

```
>> S = WFDB_isigopen('100s');
>> for ii = 1:length(S)
sprintf('Signal %d, gain: %d', ii-1, S(ii).gain)
end
ans =
Signal 0, gain: 200
ans =
Signal 1, gain: 200
```

An error message is produced if WFDB_isigopen is unable to open any of the signals listed in the header file, or if it cannot read the header file. It is not considered an error if only some of the signals can be opened, however.

See also: WFDB_Siginfo (3.8)

WFDB_osigopen

Usage: `S = WFDB_osigopen(RECORD, NSIG);`
Input: `RECORD`: (string) record name
`NSIG`: number of signals to open
Output: `S`: Siginfo structure(s)

This function opens output signal files. Use it only if signals are to be written using WFDB_putvec. The signal specifications, including the file names, are read from the header file for a specified record and returned in the structure array S. If RECORD begins with '+', previously opened output signal files are left open, and the record name is taken to be the remainder of RECORD after discarding the '+'. Otherwise, osigopen closes any previously opened output signal files, and takes all of RECORD as the record name. S is a WFDB_Siginfo structure array which, on return, will be filled with the signal specifications.

No more than NSIG (additional) output signals will be opened by `WFDB_osigopen`, even if the header file contains specifications for more than NSIG signals.

See also: WFDB_Siginfo (3.8), WFDB_putvec (3.3)

WFDB_osigfopen

Usage: `WFDB_osigfopen(S);`
Input: `S`: Siginfo structure(s)

This function opens output signals, as does WFDB_osigopen, but the signal specifications, including the signal file names, are supplied by the caller to WFDB_osigfopen, rather than read from a header file as in WFDB_osigopen. Any previously open output signals are closed by WFDB_osigfopen. S is a WFDB_Siginfo structure array (see section 3.8), with one element for each signal to be opened.

Before invoking WFDB_osigfopen, the caller must fill in the fields of the WFDB_Siginfo structure S. To make a multiplexed signal file, specify the same fname and group for each signal to be included. For ordinary (non-multiplexed) signal files, specify a unique fname and group for each signal. See section 2.4: Creating a signal file, for an illustration of the use of WFDB_osigfopen.

See also: WFDB_Siginfo (3.8)

WFDB_wfdbinit

[Not yet implemented]

3.2 Special Input Modes

WFDB_setifreq

Usage: `WFDB_setifreq(IFREQ);`
Input: `IFREQ`: desired input sampling frequency

This function sets the current input sampling frequency (in samples per second per signal). It should be invoked after opening the input signals (using `WFDB_isigopen`), and before using any of `WFDB_getvec, WFDB_getann, WFDB_putann, WFDB_isigsettime, WFDB_isgsettime, WFDB_timstr, WFDB_mstimstr`, or `WFDB_strtim`. Note that the operation of `WFDB_getframe` is unaffected by `WFDB_setifreq`.

Use `WFDB_setifreq` when your application requires input samples at a specific frequency. After invoking `WFDB_setifreq`, `WFDB_getvec` resamples the digitized signals from the input signals at the desired frequency (see section 3.3), and all of the WFDB_tools functions that accept or return times in sample intervals automatically convert between the actual sampling intervals and those corresponding to the desired frequency.

See also: WFDB_getvec (3.3)

WFDB_getifreq

Usage: `FREQ = WFDB_getifreq;`
Input: `FREQ`: input sampling frequency

This function returns the current input sampling frequency (in samples per second per signal), which is either the raw sampling frequency for the record (as would be returned by WFDB_sampfreq (see section 3.7), or the frequency chosen using a previous invocation of WFDB_setifreq.

See also: WFDB_sampfreq (3.7), WFDB_setifreq (3.2)

WFDB_setgvmode

Usage: `WFDB_setgvmode(MODE);`
Input: `MODE`: (string) either 'WFDB_LOWRES' (default) or 'WFDB_HIGHRES'.

Set the mode used by WFDB_getvec when reading a multi-frequency record. If MODE is 'WFDB_LOWRES', WFDB_getvec decimates oversampled signals. If MODE is 'WFDB_HIGHRES', WFDB_getvec interpolates signals sampled at a lower frequency (repeating the last sample value).

Example: Signal 0 is sampled using 100 Hz and signal 1 using 200Hz. With WFDB_LOWRES, WFDB_getvec returns samples using 100 Hz and signal 1 is decimated from 200 Hz to 100 Hz. With WFDB_HIGHRES, WFDB_getvec returns samples using 200 Hz and signal 0 is interpolated from 100 Hz to 200 Hz.

WFDB_setgvmode also affects how annotations are read and written. If WFDB_setgvmode('WFDB_HIGHRES') is invoked before using any of WFDB_annopen, WFDB_getvec, WFDB_sampfreq, WFDB_strtim, or WFDB_timstr, then all time data (including the time attributes of annotations read by WFDB_-getann or written by WFDB_putann) visible to the application are in units of the high-resolution sampling intervals. (Otherwise, time data are in units of frame intervals.)

WFDB_getspf

Usage: `SPF = WFDB_getspf;`
Output: `SPF`: samples per frame

Unless the application is operating in WFDB_HIGHRES mode (see section 3.2) and has then opened a multi-frequency record, this function returns 1. For the case of a multi-frequency record being read in high resolution mode, however, WFDB_getspf returns the number of samples per signal per frame (hence WFDB_sampfreq/WFDB_getspf is the number of frames per second).

3.3 Reading and Writing Signals and Annotations

WFDB_getvec

Usage: `DATA = WFDB_getvec(NSIG);`
`DATA = WFDB_getvec(NSIG, NSAMP);`
`DATA = WFDB_getvec(NSIG, NSAMP, TSTART);`
Input: `NSIG`: number of signals
`NSAMP`: (optional) number of samples to read
`TSTART`: (optional) sample number of the first sample to read
Output: `DATA`: sample value(s)

This function reads samples from open input signals. Typically, we prepare to use this function by

```
S = WFDB_isigopen(record);
NSIG = length(S);
```

to open the signals for a `record` of choice, and to determine `NSIG`, the number of signals available in the record.

To read `NSAMP` samples of each signal, beginning at sample number `TSTART`:

```
DATA = WFDB_getvec(NSIG, NSAMP, TSTART);
```

The first sample of each signal has sample number 0 (not 1!).

To read the next `NSAMP` samples of each signal:

```
DATA = WFDB_getvec(NSIG, NSAMP);
```

This form returns up to `NSAMP` samples, from sample number `T` to sample number `T+NSAMP-1`, where `T` is the input pointer (initially 0). The input pointer is incremented by the number of samples that have been read, so that a subsequent use of `WFDB_getvec` returns the next `NSAMP` samples, etc. Use `WFDB_isigsettime` (see section 3.4) to set the input pointer directly.

If the record is not too long, read it all at once by:

```
DATA = WFDB_getvec(NSIG);
```

Note that recordings can be arbitrarily long and are often much larger than available memory; also note that there may be a very long delay if an entire record is read from a remote web server over a slow link.

WFDB_getframe

Usage: `DATA = WFDB_getframe(NSPF);`
`DATA = WFDB_getframe(NSPF, NF);`
`DATA = WFDB_getframe(NSPF, NF, TSTART);`

Input: `NSPF`: number of samples per frame
`NF`: (optional) number of frames to read
`TSTART`: (optional) frame number of the first frame to read

Output: `DATA`: sample value(s)

This function reads frames from open input signals. It is very similar to WFDB_getvec, and differs only when reading multifrequency records, which it neither interpolates nor decimates; rather, it returns all sample values for each data frame.

For example, if there are two signals, and signal 0 is sampled twice as fast as signal 1, each frame contains two samples of signal 0 followed by one sample of signal 1, so NSPF is 3 (two plus one). Suppose that we run:

```
{\tt DATA = WFDB_getframe(3, 5, 10)}
```

This loads five frames (frame numbers 10 through 14) into DATA, which will be organized as:

```
[signal 0, sample 20]  [signal 0, sample 21]  [signal 1, sample 10]
[signal 0, sample 22]  [signal 0, sample 23]  [signal 1, sample 11]
[signal 0, sample 24]  [signal 0, sample 25]  [signal 1, sample 12]
[signal 0, sample 26]  [signal 0, sample 27]  [signal 1, sample 13]
[signal 0, sample 28]  [signal 0, sample 29]  [signal 1, sample 14]
```

WFDB_putvec

Usage: `WFDB_putvec(DATA))`

Input: `DATA`: sample(s) to be written

This function writes the specified sample(s) to output signal file(s). Within DATA, signals are stored as columns. The number of columns must equal the number of output signals as specified when calling WFDB_osigfopen or WFDB_osigopen.

WFDB_getann

Usage: `ANNOTATION = WFDB_getann(ANN_NUM)`
`ANNOTATION = WFDB_getann(ANN_NUM, NANN)`
Input: `ANN_NUM`: annotator number (note: the first input annotator is annotator number 0)
`NANN`: (optional) number of annotations wanted
Output: `ANNOTATION`: Annotation structure(s)

This function reads the next NANN annotations from an open input annotation file (designated by its annotator number, ANN_NUM). If NANN is omitted, it reads all annotations at once. Use WFDB_iannsettime to set the file pointer.

See WFDB_Annotation (section 3.8) for information on the contents of the annotation structures.

WFDB_ungetann

[Not yet implemented]

WFDB_putann

Usage: `WFDB_putann(ANN_NUM, ANNOTATION)`
Input: `ANN_NUM`: annotator number (note: the first output annotator is annotator number 0)
`ANNOTATION`: Annotation structure(s)

This function writes annotation(s) to an open output annotation file (designated by its annotator number, ANN_NUM, and created using WFDB_Anninfo and WFDB_annopen). Use WFDB_Annotation to create the annotation structures, and fill in their data fields before invoking WFDB_putann. If possible, annotations should be written in canonical (time/chan) order (otherwise, the annotations will be rewritten in order when WFDB_wfdbquit is invoked).

3.4 Non-Sequential Access to WFDB Files

The next several functions permit random access to signal and annotation files.

WFDB_isigsettime

Usage: `WFDB_isigsettime(TIME)`
Input: `TIME`: sample number of the next sample to be read

Set input signal file pointer to sample number TIME. (The first sample is sample 0.)

WFDB_isgsettime

Usage: `WFDB_isgsettime(SGROUP, TIME)`
Input: `SGROUP`: signal group number (note: the first signal group is group 0, etc.)
`TIME`: sample number of the next sample to be read

This function does the job of WFDB_isigsettime, but only for the signal group SGROUP. This function may be of use if more than one record is open simultaneously.

WFDB_iannsettime

Usage: `WFDB_iannsettime(TIME)`
Input: `TIME`: minimum value for the `time` of the next annotation to be read

This function can be used to skip to a desired TIME in a set of annotations, so that the next annotation to be read will be the first one occurring at or after the specified TIME.

WFDB_sample

[Not yet implemented]

WFDB_sample_valid

[Not yet implemented]

3.5 Conversion Functions

Functions in this section perform various useful conversions: between annotation codes and printable strings, between times in sample intervals and printable strings, between Julian dates and printable strings, and between ADC units and physical units.

WFDB_annstr

Usage: `MNEMONIC = WFDB_annstr(CODE);`
Input: `CODE`: (scalar or vector) annotation code(s)
Output: `MNEMONIC`: (string, if `CODE` is scalar; otherwise, a cell of strings) annotation mnemonic(s)

`WFDB_annstr` translates WFDB annotation code(s) to mnemonic(s).

WFDB_anndesc

Usage: `DESC = WFDB_anndesc(CODE);`
Input: `CODE`: (scalar or vector) annotation code(s)
Output: `DESC`: (string, if `CODE` is scalar; otherwise, a cell of strings) annotation description string(s)

`WFDB_anndesc` translates WFDB annotation code(s) to description(s).

WFDB_ecgstr

Usage: `MNEMONIC = WFDB_ecgstr(CODE);`
Input: `CODE`: (scalar or vector) annotation code(s)
Output: `MNEMONIC`: (string, if `CODE` is scalar; otherwise, a cell of strings) annotation mnemonic(s)

`WFDB_ecgstr` translates WFDB annotation code(s) to mnemonic(s). The mnemonics shared by `WFDB_ecgstr` and `WFDB_strecg` can be modified independently of those shared by `WFDB_annstr` and `WFDB_strann`, although both sets of mnemonics are initially identical. For further details, see the description of `setannstr` in the *WFDB Programmer's Guide*.

WFDB_strann

Usage: `MNEMONIC = WFDB_strann(CODE);`
Input: `MNEMONIC`: (string, or cell of strings) annotation mnemonic(s)
Output: `CODE`: annotation code(s)

This function translates one or more ASCII strings (annotation mnemonics) to numeric annotation codes.

WFDB_strecg

Usage: `MNEMONIC = WFDB_strecg(CODE);`
Input: `MNEMONIC`: (string, or cell of strings) annotation mnemonic(s)
Output: `CODE`: annotation code(s)

This function translates one or more ASCII strings (annotation mnemonics) to numeric annotation codes.

WFDB_setannstr

[Not yet implemented]

WFDB_setanndesc

[Not yet implemented]

WFDB_setecgstr

[Not yet implemented]

WFDB_timstr

Usage: `STR = WFDB_timstr(T);`
Input: `T`: (integer) time, in sample intervals
Output: `STR`: (string) time, in HH:MM:SS format

This function translates a time expressed in sample intervals into a string indicating time in hours, minutes, and seconds. If T is positive, the string indicates the time interval between sample number T and sample number 0; otherwise, the time interval represented by -T is added to the base time (as indicated in the '.hea' file or previously set using `WFDB_setbasetime`) and the string indicates the time of day (and the date, if the base date has been set) at the time of sample number (-T).

WFDB_mstimstr

Usage: `STR = WFDB_mstimstr(T);`
Input: `T`: (integer) time, in sample intervals
Output: `STR`: (string) time, in HH:MM:SS.sss format

This function is similar to `WFDB_timstr`, except that the output string specifies the time with a resolution of 1 millisecond.

WFDB_strtim

Usage: `T = WFDB_strtim(STR);`
Input: `STR`: (string) time, in HH:MM:SS format
Output: `T`: (integer) time, in sample intervals

`WFDB_strtim` converts an ASCII string to a time in units of sample intervals. The string should be supplied in *standard time format* (see http://www.physionet.org/physiotools/wpg/strtim.htm).

WFDB_datstr

Usage: `STR = WFDB_datstr(DATE);`
Input: `DATE`: (integer) Julian date
Output: `STR`: (string) date, in DD/MM/YYYY format

This function converts the Julian date represented by DATE into an ASCII string.

WFDB_strdat

Usage: `STR = WFDB_strdat(DATE);`
Input: `STR`: (string) date, in DD/MM/YYYY format
Output: `DATE`: (integer) Julian date

This function converts an ASCII string in DD/MM/YYYY format into a Julian date. If the string is improperly formatted, `WFDB_strdat` returns zero. Note that dates such as '15/3/89' refer to the first century A.D., not the twentieth or twenty-first.

WFDB_aduphys

Usage: `PHYS = WFDB_aduphys(SIG_NUM, ADU);`
Input: `SIG_NUM`: signal number (note: the first signal is signal number 0, etc.)
ADU: signal values in A/D units (scalar or vector)
Output: `PHYS`: signal value(s) in physical units

`WFDB_aduphys` converts signal value(s) in ADU from A/D units to physical units, based on the values of gain and baseline for input signal number SIG_NUM.

WFDB_physadu

Usage: `ADU = WFDB_physadu(SIG_NUM, PHYS);`
Input: `SIG_NUM`: signal number (note: the first signal is signal number 0, etc.)
`PHYS`: signal value(s) in physical units (scalar or vector)
Output: ADU: signal values in A/D units

`WFDB_physadu` converts signal value(s) in PHYS from physical units to ADU units, based on the values of gain and baseline for input signal number SIG_NUM.

WFDB_adumuv

Usage: `MUV = WFDB_adumuv(SIG_NUM, ADU);`
Input: `SIG_NUM`: signal number (note: the first signal is signal number 0, etc.)
ADU: signal values in A/D units (scalar or vector)
Output: `MUV`: signal value(s) in microvolts

`WFDB_adumuv` converts signal value(s) in ADU from A/D units to microvolts, based on the values of gain and baseline for input signal number SIG_NUM.

WFDB_muvadu

Usage: `ADU = WFDB_muvadu(SIG_NUM, MUV);`
Input: `SIG_NUM`: signal number (note: the first signal is signal number 0, etc.)
`MUV`: signal value(s) in microvolts (scalar or vector)
Output: `ADU`: signal values in A/D units

`WFDB_muvadu` converts signal value(s) in MUV from microvolts to ADU units, based on the values of gain and baseline for input signal number SIG_NUM.

3.6 Calibration Functions

Functions in this section are used to determine specifications for calibration pulses and customary scales for plotting signals. All of them make use of the *calibration list*, which contains entries for various types of signals.

WFDB_calopen

[Not yet implemented]

WFDB_getcal

[Not yet implemented]

WFDB_putcal

[Not yet implemented]

WFDB_newcal

[Not yet implemented]

WFDB_flushcal

[Not yet implemented]

3.7 Miscellaneous Functions

WFDB_newheader

Usage: `WFDB_newheader(RECORD);`
Input: `RECORD`: (string) record name

This function creates a '.hea' file. Use WFDB_newheader just after you have finished writing the signal files, but before calling WFDB_wfdbquit. If RECORD begins with '+', the '+' is discarded and the remainder of RECORD is taken as the record name. If the record name is '-', the header file is written to the standard output.

WFDB_setheader

Usage: `WFDB_setheader(RECORD, S, NSIG)`
Input: `RECORD`: (string) record name
`S`: Siginfo structure(s)
`NSIG`: number of signals

This function creates or recreates a header file for the specified RECORD, based on the contents of the first NSIG members of S.

WFDB_setmsheader

[Not yet implemented]

WFDB_wfdbquit

Usage: `WFDB_wfdbquit;`

This function closes all open database files and resets the following:

- the factors used for converting between samples, seconds, and counter values (reset to 1), the base time (reset to 0, i.e., midnight), and the base counter value (reset to 0); see WFDB_mstimstr and WFDB_timstr
- the parameters used for converting between adus and physical units (reset to WFDB_DEFGAIN adu/mV, a quantity defined in `<wfdb/wfdb.h>`); see WFDB_aduphys
- internal variables used to determine output signal specifications; see `WFDB_newheader`

If any annotations have been written out-of-order, this function attempts to run `sortann` (see the *WFDB Applications Guide*) as a subprocess to restore the annotations to canonical order. If this cannot be done, it prints a warning message indicating that the annotations are not in order, and providing instructions for putting them in order.

WFDB_iannclose

Usage: `WFDB_iannclose(ANN);`
Input: `ANN`: annotator number (note: the first annotator is annotator number 0, etc.)

This function closes the annotation file associated with input annotator ANN.

WFDB_oannclose

Usage: `WFDB_oannclose(ANN);`
Input: `ANN`: annotator number (note: the first annotator is annotator number 0, etc.)

This function closes the annotation file associated with output annotator ANN.

WFDB_wfdbquiet

Usage: `WFDB_wfdbquiet;`

This function suppresses error reporting on the standard error output from the WFDB library functions.

WFDB_wfdbverbose

Usage: `WFDB_wfdbverbose;`

This function restores normal error reporting after using `WFDB_wfdbquiet`.

WFDB_wfdberror

Usage: `WFDB_wfdberror;`

This function returns a string containing the text of the most recent WFDB library error message (or a string containing the WFDB library version number, if there have been no errors).

WFDB_sampfreq

Usage: `FREQ = WFDB_sampfreq;`
`FREQ = WFDB_sampfreq(RECORD);`
Input: `RECORD`: (string) record name
Output: `FREQ`: (real) sampling frequency

This function determines the sampling frequency (in Hz) for the record specified by its argument, if any. If RECORD is omitted, `WFDB_sampfreq` returns the currently defined sampling frequency, if any.

WFDB_setsampfreq

Usage: `WFDB_setsampfreq(FREQ);`
Input: `FREQ`: (real) sampling frequency

This function sets the sampling frequency, in sample intervals per second, used by the time-conversion functions. Use setsampfreq before creating a new 'hea' file.

WFDB_setbasetime

Usage: `WFDB_setbasetime;`
`WFDB_setbasetime(BASETIME);`
Input: `BASETIME`: (string) time of sample 0, in HH:MM:SS format; an optional base date in dd/mm/yyyy format can follow the time, separated from it by a space or tab character.

This function sets the base time (the time of day corresponding to sample number 0), used by the time-conversion functions `WFDB_timstr` and `WFDB_strtim`. Use `WFDB_setbasetime` after defining the sampling frequency and before creating a header file. If called without an argument, the current date and time are read from the system clock.

WFDB_getcfreq

Usage: `FREQ = WFDB_getcfreq;`
Output: `FREQ`: (real) counter frequency

This function returns the currently-defined counter frequency. The counter frequency is set by the functions that read header files, or by `WFDB_setcfreq`. If the counter frequency has not been defined explicitly, `WFDB_getcfreq` returns the sampling frequency.

WFDB_setcfreq

Usage: `WFDB_setcfreq(FREQ);`
Input: `FREQ`: (real) counter frequency

This function sets the counter frequency, in Hz. Use `WFDB_setcfreq` before creating a '.hea' file. The effect of `WFDB_setcfreq` is nullified by later invoking any of the functions that read header files. If FREQ is zero or negative, the counter frequency is treated as equivalent to the sampling frequency.

WFDB_getbasecount

Usage: `BASECOUNT = WFDB_getbasecount;`
Output: `BASECOUNT`: (real) base counter value

This function gets the base counter value (the counter value corresponding to sample 0), which is set by the functions that read header files, or by `WFDB_setbasecount`. If the base counter value has not been set explicitly, `WFDB_getbasecount` returns zero.

WFDB_setbasecount

Usage: `WFDB_setbasecount(BASECOUNT);`
Input: `BASECOUNT`: (real) base counter value

This function sets the base counter value (the counter value corresponding to sample number 0). Use `WFDB_setbasecount` before creating a header file. The effect of `WFDB_setbasecount` is nullified by later invoking any of the functions that read '.hea' files.

WFDB_setwfdb

[Not yet implemented]

WFDB_getwfdb

Usage: `WFDB = WFDB_getwfdb`
Output: `WFDB`: (string) WFDB path

This function gets the current database path (the list of locations that the WFDB library searches to find its input files).

WFDB_wfdbfile

Usage: `PATH = WFDB_wfdbfile(NAME);`
`PATH = WFDB_wfdbfile(TYPE, RECORD);`
Input: `NAME`: (string) file name, e.g. '100s.hea'
`TYPE`: (string) type of file, e.g. 'hea' or 'atr'
`RECORD`: (string) record name, e.g. '100s'
Output: `PATH`: (string) pathname of the file or URL

This function locates an existing WFDB file by searching the database path. The file is specified by TYPE and RECORD, or by its NAME. On return, PATH includes the appropriate component of the database path; since the database path may include empty or non-absolute components, the string is not necessarily an absolute pathname. PATH may also be a URL rather than a pathname. If the file cannot be found, WFDB_wfdbfile returns an empty string, ".

WFDB_wfdbflush

Usage: `WFDB_wfdbflush;`

This function brings database output files up-to-date by forcing any output annotations or samples that are buffered to be written to the output files.

WFDB_getinfo

Usage: `INFO = WFDB_getinfo(RECORD);`
Input: `RECORD`: (string) record name
Output: `INFO`: information string(s). If only one info string is found, INFO is a string; if two or more are found, INFO is a cell of strings. If there are no info strings, INFO is " (the empty string).

This function reads information string(s) from a .hea file.

WFDB_putinfo

Usage: `WFDB_putinfo(INFO);`
Input: `INFO`: information string(s) (a string or a cell of strings; no string may be longer than 254 characters)

`WFDB_putinfo` writes “info” string(s) into the '.hea' file that was created by the most recent invocation of `WFDB_newheader`. Two or more info strings may be written to the same header by successive invocations of `WFDB_putinfo`, or by passing INFO as a cell of strings.

Note that `WFDB_newheader` or `WFDB_setheader` must be used before `WFDB_putinfo`.

WFDB_setibsize

[Not yet implemented]

WFDB_setobsize

[Not yet implemented]

WFDB_wfdbgetskew

Usage: `SKEW = WFDB_wfdbgetskew(SIG_NUM);`
Input: `SIG_NUM`: signal number (note: the first signal is signal number 0, etc.)
Output: `SKEW`: skew, in frame intervals

This function gets the skew (as recorded in the '.hea' file, but in frame intervals rather than in sample intervals) of the specified input signal, or 0 if SIG_NUM is not a valid input signal number. Since sample vectors returned by `WFDB_getvec` or `WFDB_getframe` are already corrected for skew, `WFDB_wfdbgetskew` is useful primarily for programs that need to rewrite existing '.hea' files, where it is necessary to preserve the previously recorded skews.

WFDB_wfdbsetskew

Usage: `WFDB_wfdbsetskew(SIG_NUM, SKEW);`
Input: `SIG_NUM`: signal number (note: the first signal is signal number 0, etc.)
`SKEW`: skew, in frame intervals

This function sets the specified skew (in frame intervals) to be recorded by `WFDB_newheader` or `WFDB_setheader` for signal SIG_NUM. `WFDB_wfdbsetskew` has no effect on the skew correction performed by `WFDB_getframe` (or `WFDB_getvec`), which is determined solely by the skews that were recorded in the header file at the time the input signals were opened.

WFDB_wfdbgetstart

Usage: `PLENGTH = WFDB_wfdbgetstart(SIG_NUM);`
Input: `SIG_NUM`: signal number (note: the first signal is signal number 0, etc.)
Output: `PLENGTH`: prolog length, in bytes

This function gets the number of bytes in the prolog of the signal file that contains the specified input signal, as recorded in the header file. Note that `WFDB_wfdbgetstart` does not determine the length of the prolog by inspection of the signal file; it merely reports what has been determined by other means and recorded in the '.hea' file. Since the prolog is not readable using the WFDB library, and since functions such as `WFDB_isigopen` and `WFDB_isigsettime` take the prolog into account when calculating byte offsets for `WFDB_getframe` and `WFDB_getvec`, `WFDB_wfdbgetstart` is useful primarily for programs that need to rewrite existing '.hea' files, where it is necessary to preserve the previously recorded byte offsets.

WFDB_wfdbsetstart

Usage:	`PLENGTH = WFDB_wfdbgetstart(SIG_NUM);`
Input:	`SIG_NUM`: signal number (note: the first signal is signal number 0, etc.)
Output:	`PLENGTH`: prolog length, in bytes

This function sets the specified prolog length (bytes) to be recorded by `WFDB_newheader` or `WFDB_setheader` for signal `SIG_NUM`. `WFDB_wfdbsetstart` has no effect on the calculations of byte offsets within signal files as performed by `WFDB_isigsettime`, which are determined solely by the contents of the '.hea' file at the time the signals were opened.

3.8 Creating structures

The WFDB C functions work with different structures of information about signals, annotators, and annotations. Some of the functions in the previous sections demand input of such structures. The following structure-creating functions are not wrappers to any C functions; rather, they are MATLAB m-files that create structure arrays containing the required fields, with working (although not correct in all cases) default values to avoid hard-to-find errors.

WFDB_Anninfo

Usage:	`A = WFDB_Anninfo(N);`
Input:	`N`: number of Anninfo structures to be created (number of annotation files to be opened)
Output:	`S`: Anninfo structure(s)

Use this function to create Anninfo structures to be passed as input to `WFDB_annopen`. Anninfo structures have two fields:

`name`	string	annotator name; defaults set by WFDB_Anninfo are `a0`, `a1`, ...
`stat`	string	input/output indicator (either '`WFDB_READ`' or '`WFDB_WRITE`'); default is '`WFDB_READ`'

The annotator name identifies a set of annotations; usually the name is that of the creator of the annotations, either a program or a person. The annotator name is the suffix of the annotation file's name; for example, the annotation file `100s.atr` is a set of annotations for record 100s, with the annotator name '`atr`'. The application may change the `name` to any string containing letters, numerals, and the '_' (underscore) character. Avoid names that are used as a suffix for another file type. The application should change the `stat` string for any output annotators to '`WFDB_WRITE`'. Any changes must be made before invoking `WFDB_annopen`; later changes have no effect.

WFDB_Siginfo

Usage: `S = WFDB_Siginfo(N);`
Input: `N`: number of Siginfo structures to be created (number of signals to be written)
Output: `S`: Siginfo structure(s)

Use this function to create Siginfo structures to be passed as input to `WFDB_osigfopen`. Siginfo structures have twelve fields:

`fname`	string	the name of the signal file (may be shared among consecutively numbered signals); default: `record.dat`
`desc`	string	description of the signal (e.g., '`ABP`', '`Resp`'); default: '`Signal 0`', '`Signal 1`', ...
`units`	string	physical units of the signal (e.g., '`mmHg`', '`%`'); default: '`mV`'
`gain`	real	number of A/D units per physical unit; default: 200
`initval`	integer	value of sample 0, in ADC units; default: 0
`group`	integer	group number (all signals sharing a signal file belong to the same group); default: 0
`fmt`	integer	storage format (one of those defined in WFDB_FMT_LIST, in `<wfdb/wfdb.h>`); default: 16
`spf`	integer	samples per frame; default: 1
`bsize`	integer	bytes per block (for use with tape and other block-structured storage media; 0 for ordinary files); default: 0
`adcres`	integer	ADC resolution in bits; default: 12
`adczero`	integer	sample value corresponding to the middle of the ADC range; default: 0
`baseline`	real	(possibly fictitious) sample value corresponding to an input of 0 in physical units; default: 0

The default values indicated above are filled in by `WFDB_Siginfo`. Any changes to these values must be made before invoking `WFDB_osigfopen`; later changes have no effect.

WFDB_Annotation

Usage: `ANN = WFDB_Annotation(N);`
Input: `N`: number of Annotation structures to be created
Output: `S`: Anninfo structure(s)

Use this function to create Annotation structures to be passed as input to `WFDB_putann`. Annotation structures have six fields:

`time`	integer	the sample number to which the annotation 'points'; defaults set by WFDB_Annotation are 1, 2, 3, ...
`anntyp`	integer	the annotation type, an integer between 1 and `ACMAX` (defined in '`<wfdb/ecgcodes.h>`'); default is 1
`subtyp`	integer	the annotation subtype, an integer between -128 and 127; default is 0
`chan`	integer	the annotation chan field, an integer between -128 and 127; default is 0
`num`	integer	the annotation num field, an integer between -128 and 127; default is 0
`aux`	string	the annotation aux string; default is an empty string

The `subtyp`, `chan`, and `num` fields do not have any preassigned meanings; they may be used to record any small integers, or left at their default (0) values, in which case they will not occupy space in the annotation file. Note that the length of the `aux` string may not exceed 254 characters.

Support

If you believe you have found a bug, please send a report including:

- the name of the wrapper
- what input you used
- what result you got
- what result you expected
- what platform you used (CPU type, operating system name and version number, MATLAB version number, WFDB software package version number, WFDB_tools version number)

Bug reports, questions, comments, and suggestions should be addressed to:

Email: `Jonas dot Carlson at kard dot lu dot se`

Fax: +46 46 157857

Address: (postal)
Jonas Carlson
Department of Cardiology
University Hospital
SE - 221 85 LUND
Sweden