WFDB Programmer's Guide

Tenth Edition
(revised and with additions for WFDB library version 10.7.0)
10 June 2022

George B. Moody
Harvard-MIT Division of Health Sciences and Technology

The most recent versions of the software described in this guide may be downloaded from `http://physionet.org/`.

See `http://physionet.org/physiotools/wpg/` for an HTML version of this guide.

Table of Contents

Preface

This guide documents the Waveform Database interface library (the *WFDB library*), a package of C-callable functions that provide clean and uniform access to digitized, annotated signals stored in a variety of formats. These functions were originally designed for use with databases of electrocardiograms, including the MIT-BIH Arrhythmia Database (*MIT DB*) and the AHA Database for the Evaluation of Ventricular Arrhythmia Detectors (*AHA DB*). In February 1990, the predefined annotation set was expanded to accommodate the needs of the European ST-T Database (*ESC DB*). The WFDB library is sufficiently general, however, to be 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 other databases that include signals such as blood pressure, respiration, oxygen saturation, EEG, as well as ECGs. Among these multi-parameter databases are the MIT-BIH Polysomnographic Database, the MGH/Marquette Foundation Waveform Database, and the MIMIC Database. Thus the WFDB library is considerably more than an *ECG* database interface.

This guide describes how to write C-language programs that use databases of ECGs and other signals. A standard set of such programs is included in the WFDB Software Package, and is described in the *WFDB Applications Guide*; other documents describe the databases themselves, and existing programs that use them (see Appendix E [Sources], page 127, for information about obtaining these and related items).

There are a few important concepts that should be well understood before going further. These concepts include *records*; *signals*, *samples*, and *time*; and *annotations*.

Records

The databases for which the WFDB library was designed consist of a small number of *records*, each of which is quite large (typically a megabyte or more). Before 1990, database records usually originated as multi-channel analog tape recordings that had been digitized and stored as disk files. For this historical reason, they are sometimes referred to as *tapes*, although most newly created records are digitally recorded onto disk media. Each record contains a continuous recording from a single subject. A typical application program accesses only a single record, and most (if not all) of the access within the record is sequential. Much less frequently, it may be of interest to compare the contents of several records, or to select sets of records. These databases are therefore qualitatively different from those for which conventional database management software is written.

Records are identified by *record names* of up to 20 characters (the limit is `MAXRNL`, defined in `<wfdb/wfdb.h>`). For example, record names in the MIT DB are three-digit numbers, those in the AHA DB are four-digit numbers, and those in the ESC DB are four-digit numbers prefixed by the letter '`e`'. You may create database records with names containing letters, digits, and underscores. Case is significant in record names that contain letters, even in environments such as MS-Windows for which case translation is normally performed by the operating system on file names; thus '`e0104`' is the name of a record found in the ESC DB, whereas '`E0104`' is not. A record is comprised of several files, which contain signals, annotations, and specifications of signal attributes; each file belonging to a given record normally includes the record name as the first part of its name. A record is an extensible

collection of files, which need not all be located in the same directory, or even on the same physical device. Thus it is possible, for example, to create a local disk file of your own annotations for a record read from a web server or a CDROM, and to treat your file as part of the record.

Signals, Samples, and Time

Signals are commonly understood to be functions of time obtained by observation of physical variables. In this guide, a *signal* is defined more restrictively as a finite sequence of integer *samples*, usually obtained by digitizing a continuous observed function of time at a fixed *sampling frequency* expressed in Hz (samples per second). The time interval between any pair of adjacent samples in a given signal is a *sample interval*; all sample intervals for a given signal are equal. The integer value of each sample is usually interpreted as a voltage, and the units are called analog-to-digital converter units, or *adu*. The *gain* defined for each signal specifies how many adus correspond to one *physical unit* (usually one millivolt, the nominal amplitude of a normal QRS complex on a body-surface ECG lead roughly parallel to the mean cardiac electrical axis). All signals in a given record are usually sampled at the same frequency, but not necessarily at the same gain (see Section 5.4 [Multi-Frequency Records], page 73, for exceptions to this rule). MIT DB records are sampled at 360 Hz; AHA and ESC DB records are sampled at 250 Hz.

The *sample number* is an attribute of a sample, defined as the number of samples of the same signal that precede it; thus the sample number of the first sample in each signal is zero. Within this guide, the units of *time* are sample intervals; hence the "time" of a sample is synonymous with its sample number.

Samples having the same sample number in different signals of the same record are treated as simultaneous. In truth, they are usually not *precisely* simultaneous, since most multi-channel digitizers sample signals in "round-robin" fashion. If this subtlety makes a difference to you, you should be prepared to compensate for inter-signal sampling skew in your programs.

Annotations

MIT DB records are each 30 minutes in duration, and are *annotated* throughout; by this we mean that each beat (QRS complex) is described by a label called an *annotation*. Typically an *annotation file* for an MIT DB record contains about 2000 beat annotations, and smaller numbers of rhythm and signal quality annotations. AHA DB records are either 35 minutes or 3 hours in duration, and only the last 30 minutes of each record are annotated. ESC DB records are each 2 hours long, and are annotated throughout. The "time" of an annotation is simply the sample number of the sample with which the annotation is associated. Annotations may be associated with a single signal, if desired. Like samples in signals, annotations are kept in time and signal order in annotation files (but see Section 5.10 [Annotation Order], page 78, for exceptions to this rule). No more than one annotation in a given annotation file may be associated with any given sample of any given signal. There may be many annotation files associated with the same record, however; they are distinguished by *annotator names*. The annotator name `atr` is reserved to identify *reference annotation files* supplied by the developers of the databases to document correct beat labels. You may use other annotator names (which may contain letters, digits and underscores, as for record

names) to identify annotation files that you create. You may wish to adopt the convention that the annotator name is the name of the file's creator (a program or a person).

Annotations are visible to the WFDB library user as C structures, the fields of which specify time, beat type, and several user-definable variables. The WFDB library performs efficient conversions between these structures and a compact bit-packed representation used for storage of annotations in annotation files.

Applications

Some typical uses of the WFDB library are these:

- A *waveform editor*, such as `wave` (see the *WAVE User's Guide* (`http://physionet.org/physiotools/wug/`)), reads the digitized signals of a database record and displays them with annotations superimposed on the waveforms. Such a program allows the user to select any portion of the signals for display at various scales, and to add, delete, or correct annotations.
- *Signal processing programs* (e.g., see [Example 7], page 89) apply digital filters to the signals of a database record and then record the filtered signals as a new record. Similar programs perform sampling frequency conversion.
- *Analysis programs* (e.g., see [Example 10], page 98) read the digitized signals, analyze them, and then record their own annotations.
- An *annotation comparator*, such as `bxb` (see Appendix C [WFDB Applications], page 117), reads two or more sets of annotations corresponding to a given record, and tabulates discrepancies between them. If the reference annotations supplied with the database are compared in this way with annotations produced using an analysis program, this comparison is a means of establishing the accuracy of the analysis program's output.

The WFDB library provides the means for programs such as those described above to select a database record, read and write signals, read and write annotations, jump to arbitrary points in the record, and determine attributes of the signals such as the sampling frequency. The library also provides a variety of other more specialized services for programs that need them. The library defines an interface between programs and the database that is sufficiently powerful, general, and efficient to eliminate the need for *ad hoc* user-written database I/O.

About this Guide

You should have a good grasp of the C language in order to make the best use of this guide. If ANSI C prototypes, used here to document the WFDB library functions, are unfamiliar to you, see pp. 217–218 in the second edition of *The C Programming Language* by Kernighan and Ritchie, Prentice Hall, 1988. (This is the famous *K&R*; all *K&R* references in this guide include page numbers for the second edition. Newcomers to C should have a copy for ready reference while reading this guide.) It may also be helpful to have a copy of a database directory, such as the *MIT-BIH Arrhythmia Database Directory* (`http://physionet.org/physiobank/database/html/mitdbdir/`). The *WFDB Applications Guide* (`http://physionet.org/physiotools/wag/`) will be useful as a reference for existing WFDB library-based applications (available from PhysioNet, `http://physionet.org/`).

You should have access to a computer that has the WFDB library and at least one or two database records on-line, or access to the World Wide Web, where database records can be obtained from PhysioNet and other sources. (If you are installing the WFDB library on a new computer for the first time, please read the installation notes supplied with the WFDB library first, or see Appendix B [Installing the WFDB Software Package], page 115, then return here.) You should know how to create a C source file using your favorite editor, and you should know how to compile it and how to run the resulting executable program.

Resist all temptation to plunge into the esoteric details of file formats. (Those who find such details irresistible will find them in Section 5 of the *WFDB Applications Guide*; note, however, that support for new file formats is added to the WFDB library from time to time, so that the information you find there may be incomplete.) The WFDB library provides an efficient means of reading and writing files in many formats; it is not a trivial task to duplicate it, and time spent doing so is time that could be spent doing something useful, enjoyable, or possibly both. If you really think you need to understand the file formats in order to translate them into whatever the ECGWhizz Model 666 needs, consider instead writing a format translator using the WFDB library to read the files; then you will at least have a program that requires only recompilation with a new version of the WFDB library when file formats change. *In extremis*, use `rdann` and `rdsamp` — available from PhysioNet in source and ready-to run formats — to translate files into text format.

Chapter 1 of this guide begins with a simple example program that reads a few samples from a database record. The C version of this program is also translated into a variety of other languages supported by the WFDB library itself or by separately available bindings. This example should help you understand the mechanics of compiling and using a program that does something with an ECG database. Chapter 2 introduces the library functions themselves, with a number of brief examples; you may wish to skim through this material on a first reading to get acquainted with what is available, and then refer to it as needed while writing your programs. Data structures for annotations and for signal and annotator attributes are described in chapter 3. Chapter 4 contains a table of annotation types and descriptions of several annotation-mapping macros. Database files and related topics are discussed in chapter 5, which can be skipped on a first reading. Chapter 6 contains additional example programs that illuminate a few of the darker corners of the WFDB library. The glossary defines the ordinary-sounding words such as *signal* that have specialized meanings in this guide; such words are *emphasized* in their first appearances in order to warn you that you should look them up in the glossary on a first reading (see Appendix A [Glossary], page 107).

If the WFDB library has not yet been installed on your system, see Appendix B [Installing the WFDB Software Package], page 115. Another appendix (see Appendix C [WFDB Applications], page 117) includes brief descriptions of the application programs that are distributed with the WFDB library as part of the WFDB software package.

Another appendix discusses porting the WFDB library to new machines or operating systems, and includes notes on adding support for new file formats, annotation codes, and other enhancements (see Appendix D [Extensions], page 123). The WFDB library has been written with portability in mind. It runs on a wide variety of machines and operating systems, including Unix (BSD 4.x, System V, SunOS, Solaris, HP-UX, OSF/1, Version 7, XENIX, VENIX, ULTRIX, GNU/Linux, FreeBSD, OpenBSD, IRIX, AIX, AUX, Darwin, Mac OS X, SCO, Coherent, and more), MS-DOS, MS-Windows, VMS, and classic Mac

OS. This guide was written for Unix users (with notes for MS-Windows and MS-DOS users where differences exist), but others should find only minor differences.

At the end of the guide is a list of sources for databases and other materials that may be useful to readers (see Appendix E [Sources], page 127), and a log of changes made to this library since 1999 (see [Recent Changes], page 137).

Many friends have contributed to the development of the WFDB library. Thanks to Paul Albrecht, Ted Baker, Phil Devlin, Scott Greenwald, Thomas Heldt, Isaac Henry, David Israel, Roger Mark, Joe Mietus, Warren Muldrow, Ikaro Silva, Wei Zong, and especially to Paul Schluter, whose elegant 8080 assembly language functions inspired these (long live `getann`!). Pat Hamilton and Bob Farrell contributed ports, to classic Mac OS and the MS 32-bit Windows environments, respectively. Jose Garcia Moros and Salvador Olmos contributed Matlab/Octave reimplementations of a useful subset of the WFDB library. Jonas Carlson wrote, documented, and contributed a set of Matlab wrappers for the WFDB library, and Michael Craig created and contributed the WFDB Toolbox for Matlab. Isaac Henry contributed SWIG wrappers that provide interfaces for a variety of scripting languages, as well as a set of translations of the C example programs in this guide into the languages supported by the SWIG wrappers. Mike Dakin provided the first implementation of NETFILES (HTTP client support in the WFDB library) based on the W3C's `libwww`. Following the W3C's decision to end development of `libwww`, Benjamin Moody reimplemented NETFILES using `libcurl`, and also implemented variable-layout records. Bug reports (and in some cases fixes) have been contributed by Omar Abdala, Winton Baker, David Brooks, Bob Farrell, Virginia Faro-Maza, Ion Gaztañaga, Fred Geheb, Mathias Gruber, Thomas Heldt, Isaac Henry, Justin Leo Chang Loong, Benjamin Moody, Guido Muesch, Joonas Paalasmaa, Tony Ricke, Dave Schaffer, Dan Scott, Allavatam Venugopal, Mauro Villarroel, Andrew Walsh, Piotr Wlodarek, and Yinqi Zhang. Thanks also to the many readers of earlier versions of this guide; if this edition answers your questions, it is because someone else has already asked them, and hounded the author until he produced comprehensible answers.

Before May, 1999, and the release of version 10.0.0 of the library, the WFDB library was known as the DB library, and this guide was the *ECG Database Programmer's Guide*. The name of the library was changed because of confusion caused by the proliferation of another library with the same name (a reimplementation of the Berkeley Unix DBM library). The names of this guide, and of the *WFDB Applications Guide* (formerly the *ECG Database Applications Guide*), have been changed in view of the increasingly broad range of applications in which the library is being used.

The first edition of this guide was written as a tutorial for MIT students using the ECG databases for a variety of signal-processing and analysis projects. The guide, and the WFDB library itself, have been extensively revised since they first appeared in 1981. Your comments and suggestions are welcome. Please send them to:

PhysioNet `<wfdb@physionet.org>`
MIT Room E25-505A
Cambridge, MA 02139
USA

If you use the GNU `emacs` editor, you can peruse a hypertext version of this guide using `info` if it has been installed on your system; among its many other features, `emacs` makes

it easy to copy code from the examples into your own programs. Installation instructions are included in the WFDB Software Package; type `C-h i` within GNU `emacs` to start up `info` (see Appendix E [Sources], page 127, for information about obtaining GNU `emacs`).

An HTML version of this guide, suitable for viewing using any web browser, is included with the WFDB Software Package. The latest version may always be viewed at `http://physionet.org/physiotools/wpg/` using your web browser.

You can format and print copies of this guide using TeX if you have it (see `makefile` in the `doc` directory of the library distribution for instructions on doing so). You may obtain preformatted versions in PDF and PostScript formats from `http://physionet.org/`.

1 Using the WFDB Library

This chapter gives a brief overview of the steps needed to compile, load, and run a program that uses the WFDB library. It assumes that you are able to log onto a Unix-based computer on which the WFDB Software Package has been installed (see Appendix B [Installing the WFDB Software Package], page 115), and that you know how to create a source file using a text editor such as `emacs` or `vi`. If you are using an MS-DOS system, there are a few differences noted below.

1.1 A Trivial Example Program in C

Suppose we wish to print the first ten samples of record `100s`. (Record `100s` is the first minute of MIT-BIH Arrhythmia Database record `100`, supplied as a sample in the `data` directory of all source distributions of the WFDB Software Package.) We might begin by creating a source file called `psamples.c` that contains:

```
#include <stdio.h>
#include <wfdb/wfdb.h>

main()
{
    int i;
    WFDB_Sample v[2];
    WFDB_Siginfo s[2];

    if (isigopen("100s", s, 2) < 2)
        exit(1);
    for (i = 0; i < 10; i++) {
        if (getvec(v) < 0)
            break;
        printf("%d\t%d\n", v[0], v[1]);
    }
    exit(0);
}
```

(See `http://physionet.org/physiotools/wfdb/examples/psamples.c` for a copy of this program.)

All programs that use the WFDB library *must* include the statement

```
#include <wfdb/wfdb.h>
```

which defines function interfaces, data types (such as the `WFDB_Sample` and `WFDB_Siginfo` types used in this example), and a few useful constants. (Most MS-DOS C compilers accept '`/`' as a directory separator. If you prefer to use the non-portable '`\`' under MS-DOS, remember to quote it: '`#include <wfdb\\wfdb.h>`'.)

The functions used in the example are described in detail in the next chapter, and the data types are described in the following chapter (see Chapter 3 [Data Types], page 57). For now, note that `isigopen` prepares a record to be read by `getvec`, which reads a sample from each of the two signals each time it is called.

Note that in some cases it may be important to insure that all memory allocated by the WFDB library is freed before the program exits; in the example program, this can be done by adding the line

```
wfdbquit();
```

just above `exit(0);` (see [wfdbquit], page 44).

1.2 Compiling a Program with the WFDB Library

The WFDB library is developed and tested using `gcc`, the GNU C/C++ compiler, but careful attention has been given to making it usable with any ANSI/ISO C compiler. Since `gcc` is free, high quality, and supported, it is highly recommended that you use it for compiling your WFDB applications.

To compile the example program using `gcc` on a Unix shell, we can say:

```
gcc -o psamples psamples.c `wfdb-config --cflags --libs`
```

to produce an executable program called `psamples`. (Your C compiler may be named `cc`, `acc`, `CC`, or something else, rather than `gcc`.)

Note that this command contains backticks (`), not apostrophes ('). The program `wfdb-config` is part of the WFDB Software Package, and tells the compiler where to find the header files (such as `wfdb/wfdb.h`) and the library itself (such as `libwfdb.so`). Writing the `wfdb-config` command in backticks means that the output of the command is inserted into the `gcc` command line. So if the WFDB library is installed in `/usr/local/lib`, then the command above is equivalent to:

```
gcc -o psamples psamples.c -I/usr/local/include -L/usr/local/lib -lwfdb
```

In addition to being shorter to type, it's a good idea to use `wfdb-config` so that your program can be compiled on other systems, regardless of where the WFDB library has been installed.

You may use any other compiler options you choose, but "`wfdb-config --cflags --libs`" must appear in the `cc` command line following any and all source (`*.c`) and object (`*.o`) file names, in order to instruct the loader to search the WFDB library for any functions that the program needs (in this case, `isigopen` and `getvec`). Some programs will need additional libraries, and the corresponding '`-l`' options can usually be given before or after the `wfdb-config` command.

Under MS-Windows, `gcc` is included in the freely available Cygwin software development system (`http://www.cygwin.com/`), and also in the freely available MinGW package (`http://www.mingw.org/`). An MS-DOS version of `gcc` is available in the free djgpp package (`http://www.delorie.com/djgpp/`). These are used within a Cygwin terminal emulator window or an MS-DOS box in exactly the same way as described above for C compilers on all other platforms. For most purposes, Cygwin is recommended, since it provides a Unix-compatible standard C library (`cygwin1.dll`), so that applications behave exactly as they do on all other platforms. WAVE can only be built under Windows in this way. When building WFDB-based plugins for use with .NET applications or others such as Matlab that rely on the native Windows C library, however, the WFDB library must be recompiled to use the native library. This can be done using either MinGW gcc, or Cygwin gcc with its `-mno-cygwin` option.

If you choose to use an incompatible proprietary compiler, you are on your own! You may be able to create a linkable version of the WFDB library from the sources in the `lib` directory of the WFDB source tree using a proprietary compiler, but doing so is unsupported (see your compiler's documentation). If you are not able to build the WFDB library using your compiler, you can compile the library sources together with the source file(s) for your application. It may be easiest to copy the library sources (both the `*.c` and the `*.h` files) into the same directory as the application sources. If you follow this approach, find the directory that contains `stdio.h` on your system and make a `wfdb` subdirectory within that directory, then copy the WFDB library's `*.h` files into the `wfdb` subdirectory (this is necessary so that statements of the form '`#include <wfdb/wfdb.h>`' will be handled properly by your compiler). For example, to compile `psamples.c` with Microsoft C/C++, set up the WFDB library source files as just described, then use this command:

```
cl psamples.c wfdbio.c signal.c annot.c calib.c wfdbinit.c
```

With Borland C/C++ or Turbo C or C++, substitute '`bcc`' or '`tcc`', respectively, for '`cl`' in the command above. You will find that some WFDB applications do not need to be compiled with all of the WFDB library sources (for example, `psamples` needs only `wfdbio.c` and `signal.c`); in such cases, you may omit the unneeded sources for faster compilation and smaller executable binaries.

1.3 Using the WFDB library with other languages

Bindings and wrappers are available so that programs written in a number of other programming languages can make use of the WFDB library.

C++ bindings

If you prefer to write your applications in C++, you may do so, but note that the WFDB library is written in C. (Most C++ compilers can be run in ANSI/ISO C compatibility mode in order to compile the WFDB library itself.) Each C++ source file that uses WFDB library functions must include `<wfdb/wfdb.h>`, in order to instruct your compiler to use C conventions for argument passing and to use unmangled names for the WFDB library functions. In order for this to work, your C++ compiler should predefine '`__cplusplus`' or '`c_plusplus`'; if it predefines neither of these symbols, modify `<wfdb/wfdb.h>` so that the symbols '`wfdb_CPP`' and '`wfdb_PROTO`' are defined at the top of the file, or define '`__cplusplus`' in each of your source files before including `<wfdb/wfdb.h>`. Compile and link your program using whatever standard methods are supported by your compiler for linking C++ programs with C libraries. See your compiler manual for further information.

Fortran wrappers

A set of wrapper functions is also available for those who wish to use the WFDB library together with applications written in Fortran. These functions, defined in `wfdbf.c` (`http://physionet.org/physiotools/wfdb/fortran/wfdbf.c`), provide a thin 'wrapper' around the WFDB library functions, by accepting Fortran-compatible arguments (there are no structures, and all arguments are passed by reference rather than by value). For example, here is the Fortran equivalent of the example program in the previous section:

```
      integer i, v(2), g
```

```
      i = isigopen("100s"//CHAR(0), 2)
      do i = 1, 10
       g = getvec(v)
       write (6,3) v(1), v(2)
3     format("v(1) = ", i4, "    v(2) = ", i4)
      end do
      end
```

(See `http://physionet.org/physiotools/wfdb/fortran/fsamples.f` for a copy of this program; an extensively commented version of this program is also available, at `http://physionet.org/physiotools/wfdb/fortran/example.f`.)

To compile this program using `gfortran` (the GNU Fortran compiler), save it as `fsamples.f` in the current directory, copy `wfdbf.c` (which can be found in the same directory as `wfdb.h`, usually `/usr/local/include/wfdb`) to the current directory, then type:

```
gfortran -o fsamples -DFIXSTRINGS fsamples.f wfdbf.c `wfdb-config --libs`
```

The Fortran wrapper functions are not discussed in this guide; for further information, refer to fortran/README (`http://physionet.org/physiotools/wfdb/fortran/README`) in the WFDB Software Package.

SWIG wrappers for Java, Perl, Python, and other languages

Isaac Henry has contributed WFDB wrappers for Java, Perl, and Python, as well as for .NET languages such as C#, created using the Simplified Wrapper Interface Generator (SWIG, `http://www.swig.org/`). Using these wrappers, the example program can be written in any of these languages:

Java:

```
import wfdb.*;

public class psamples {
    static {
        System.loadLibrary("wfdbjava");
    }

    public static void main(String argv[]) {
        WFDB_SiginfoArray siarray = new WFDB_SiginfoArray(2);
        if (wfdb.isigopen ("100s", siarray.cast(), 2) < 2)
            System.exit(1);
        WFDB_SampleArray v = new WFDB_SampleArray(2);
        for (int i = 0; i < 10; i++) {
            if (wfdb.getvec(v.cast()) < 0)
                break;
            System.out.println("\t" + v.getitem(0) +"\t"+ v.getitem(1));
        }
    }
}
```

Perl:

```
package wfdb;
```

```
use wfdb;

$siarray = new wfdb::WFDB_SiginfoArray(2);
if ($nsig = isigopen("100s", $siarray->cast(), 2) < 2) {
    exit(1);
}
$v = new wfdb::WFDB_SampleArray(2);
for ($i=0; $i < 10; $i++) {
    if (getvec($v->cast()) < 0) {
        exit(2);
    }
    print "\t", $v->getitem(0), "\t", $v->getitem(1), "\n";
}
```

Python:

```
import wfdb, sys

def main(argv):
    siarray = wfdb.isigopen("100s")
    if siarray.nsig < 2: sys.exit(1)
    v = wfdb.WFDB_SampleArray(2)
    for i in range(0,10):
        if wfdb.getvec(v.cast()) < 0: sys.exit(2)
        print "\t%d\t%d" % (v[0], v[1])

if __name__ == "__main__":
    main(sys.argv[1:])
```

C#:

```
using System;
using Wfdb;

public class psamples {
    static void Main(string[] argv) {
        WFDB_SiginfoArray siarray = new WFDB_SiginfoArray(2);
        if (wfdb.isigopen("100s", siarray.cast(), 2) < 2)
            Environment.Exit(1);
        WFDB_SampleArray v = new WFDB_SampleArray(2);
        for (int i = 0; i < 10; i++) {
            if (wfdb.getvec(v.cast()) < 0)
                break;
    Console.WriteLine("\t" + v.getitem(0) + "\t" + v.getitem(1));
        }
    }
}
```

All SWIG wrappers for the WFDB library are generated using a single interface file, `wfdb.i`. In principle, this file might be used to generate wrappers for other programming

languages supported by SWIG, including several versions of LISP, Modula-3, PHP, Ruby, and Tcl.

Matlab toolbox

The WFDB Toolbox for Matlab, contributed by Michael Craig and available from `http://physionet.org/physiotools/matlab/wfdb-swig-matlab/`, is a collection of WFDB applications implemented as functions in Matlab, built on the SWIG Java wrappers for the WFDB library. For example, using it in Matlab, one can read and plot the first five seconds of the same signals as in the example program above, by:

```
r = rdsamp('100s', 'maxt', ':5');
plot(r(:,1), r(:,2));
```

The WFDB Toolbox for Matlab replaces the `wfdb_tools` library of wrapper functions contributed by Jonas Carlson, since current versions of Matlab are no longer compatible with the `wfdb_tools` library.

The WFDB Software Package includes `wfdb2mat`, an application that converts all or any desired segment of a signal file into a `.mat` file that can be read directly by Matlab.

Jesus Olivan Palacios has written a tutorial (available at `http://www.neurotraces.com/scilab/sciteam/`) that includes a detailed section on using the WFDB Software Package with Scilab (an open-source scientific software package for numerical computations, with a language similar to that of Matlab, available from `http://www-rocq.inria.fr/scilab/`). The methods described in this tutorial can also be adapted for use with GNU Octave (another free language that is mostly compatible with Matlab, available from `http://www.gnu.org/software/octave/`).

Also available is a reimplementation of a useful subset of the WFDB library in native m-code (contributed by Jose Garcia Moros and Salvador Olmos) at `http://physionet.org/physiotools/matlab/`.

1.4 The Database Path and Other Environment Variables

WFDB applications make use of several *environment variables*, which are named `WFDB`, `WFDBCAL`, `WFDBGVMODE`, and `WFDBANNSORT`. If these variables have not been otherwise defined by the user, their values are those given by `DEFWFDB`, `DEFWFDBCAL`, `DEFWFDBGVMODE`, and `DEFWFDBANNSORT` (defined in `wfdblib.h` at the time the WFDB library was compiled). Unless you have a non-standard setup, you may not need to set these variables, but it will be helpful to read this section to understand how they influence the behavior of WFDB applications.

When WFDB applications *read* database files, they must be able to find them in various locations that may vary from system to system. The WFDB library refers to a character string that consists of an ordered list of locations to be searched *for input files*. This string is called the *database path*, or the *WFDB path*.

On most systems, the environment variable `WFDB`, if set, specifies the value of the WFDB path, and overrides the default value. If you need to use a non-default WFDB path, you must set the `WFDB` environment variable appropriately before running any WFDB applications, so that the WFDB path can be examined by the running program. The WFDB software package includes easily customizable shell scripts (batch files) that illustrate how to do this

for popular shells and command interpreters; see *setwfdb*(1), in the *WFDB Applications Guide*. (Under classic Mac OS, for which the concept of environment variables is foreign, the WFDB path may be set only by using `DEFWFDB`.) For further information, see Section 1.7 [WFDB path syntax], page 14.

The shell scripts that set `WFDB` also set the `WFDBCAL` environment variable, which is important if you make use of records that contain signals other than ECGs. `WFDBCAL` names a *calibration file* located in one of the directories named by `WFDB`. (The symbol `DEFWFDBCAL` is usually defined in `wfdblib.h` to specify the name of a default calibration file, to be used by the WFDB library if `WFDBCAL` has not been set.) Each signal type may be represented by an entry in the calibration file. Entries specify the characteristics of any calibration pulses that may be present, and customary scales for plotting the signals.

The other environment variables are less frequently used than `WFDB` and `WFDBCAL`, and in most cases, the compiled-in defaults will be appropriate (see Section 5.10 [Annotation Order], page 78, and see Section 5.4 [Multi-Frequency Records], page 73, for details).

1.5 Running the Example Program

If `WFDB` is properly set, MIT DB record `100s` is on-line and readable, and the example program was compiled correctly, it can be run by typing

```
psamples
```

(Try '`./psamples`' if '`psamples`' doesn't work.) Its output will appear as:

```
995     1011
995     1011
995     1011
995     1011
995     1011
995     1011
995     1011
995     1011
1000    1008
997     1008
```

The left column contains samples from signal 0, and the right column contains those from signal 1.

1.6 A Note on Identifiers

External identifiers that begin with the underscore ('`_`') character are reserved under the rules of ANSI C to the compiler and libraries. In order to make the WFDB library as portable as possible, its own external identifiers do not begin with underscores (since otherwise they might conflict with external identifiers used by a standard library).

External identifiers beginning with '`wfdb_`' are reserved for the use of the WFDB library. These names are used for functions and global variables that are intended for the private use of the WFDB library; your programs should not need to use them. You should avoid defining functions or global variables with such names in your programs.

External identifiers beginning with '`WFDB_`' are used for constants and data types defined within `<wfdb/wfdb.h>`. Use these identifiers as needed in your programs, but avoid redefining them.

1.7 More About the WFDB Path

When a WFDB file must be opened for input, the WFDB library attempts to locate it by attaching each of the components of the WFDB path (one at a time) as a prefix to the file name. If two or more matching files exist in different locations in the WFDB path, the WFDB library opens only the file that resides in the first of these locations. Any other matching files are effectively invisible to WFDB applications unless the WFDB path is rearranged.

The default WFDB path is specified at the time the WFDB library is compiled, by defining a value for the symbol `DEFWFDB` in `wfdblib.h`. Current versions of the WFDB library are compiled with a three-component default WFDB path; the first component is empty (i.e., it refers to the current directory), the second component names the *system-wide database directory* (which contains the sample WFDB files supplied with the WFDB software package), and the third component is `http://physionet.org/physiobank/database` (referring to the PhysioBank data archives). Note that this default may be changed at the time the WFDB library is compiled. Normally, however, this means that any record available from PhysioBank is readable by any WFDB application provided that PhysioBank is accessible from the user's computer and that the database name is included in the record name (for example, `slpdb/slp60` or `nsrdb/16265`).

Under Unix and VMS, the WFDB path can be given as a colon-separated list of prefixes, in the format used for the Bourne shell's `PATH` variable. Under MS-Windows, MS-DOS, and classic Mac OS, the WFDB path can be given in the format used for the MS-DOS `PATH` variable, with semicolons used to separate prefixes (colons retain their customary meanings, as drive letter suffixes under MS-DOS, or as directory separators on the Macintosh). Alternatively, components of the WFDB path may be separated by whitespace (under any operating system); this also implies that embedded spaces are not permitted within path components. **For this reason, avoid using directories with names such as `My Documents`, or their subdirectories, to store WFDB files.**

When WFDB applications *write* database files, these files are generally written to the current directory. (As an example, an application that analyzes one or more signals in a record may record its findings in an annotation file in the current directory.) If the record name (as provided by the application to the WFDB library) contains path information, however, output files are written to the corresponding subdirectory of the current directory. (For example, if a WFDB application writes an annotation file for record `edb/e0103`, the file will be written in the `edb` subdirectory of the current directory. The `edb` subdirectory will be created by the WFDB library if does not exist already. This feature was introduced in WFDB library version 10.2.0.)

Note particularly that the current directory is *not* necessarily part of the WFDB path. If you modify your WFDB path, you must explicitly include an empty (null) component, which corresponds to the current directory, in order to be sure that your WFDB applications can read any WFDB files that you have previously written. In most cases, this null component should be the first in the WFDB path. Thus, if you write into the current directory a

modified version of an existing WFDB file, any later actions that would read this file will read your modified version rather than the original.

The WFDB path may contain `http://` and `ftp://` URL prefixes (other schema, such as `file://` and `https://`, may also be supported if they are supported by your version of `libcurl`). If NETFILES support is not compiled into the WFDB library, any WFDB path components containing `://` are ignored. (These features were first introduced in WFDB library version 10.1.0.)

If the WFDB library finds that the value assigned to the WFDB path is of the form '`@file`', it replaces that value with the contents of the specified *file*. (This feature was first introduced in WFDB library version 8.0.) Indirect WFDB path files may be nested up to ten levels (this arbitrary limit is imposed to avoid infinite recursion if the contents of the indirect file are incorrect). This method of indirect assignment is useful under classic Mac OS, where recompilation of the WFDB library would otherwise be necessary in order to change the WFDB path. It may also be useful under MS-DOS to reduce the need for environment space, or if the length of the command needed to set the `WFDB` environment variable would otherwise approach or exceed the 128-byte limit for MS-DOS commands.

If a WFDB header file (see Chapter 5 [Database Files], page 71) specifies that a signal file is to be found in a directory that is not already in the WFDB path, that directory is appended to the end of the WFDB path; in this case, if the WFDB path is not set, it is created with an initial null component followed by the directory that contains the signal file. (This feature was first introduced in WFDB library version 6.2.)

The string '`%r`' is replaced by the current record name wherever it appears in the WFDB path; '`%N`*r*' is replaced by the first *N* digits of the record name, if *N* is a non-zero digit. For example, if (under Unix) the WFDB path is '`:/cdrom/mimicdb/%3r:/cdrom/mitdb`', a request to read a file associated with record 055n will cause the WFDB library to look first in the current directory (since the WFDB path begins with an empty component), then in '`/cdrom/mimicdb/055`', and then in '`/cdrom/mitdb`'. If '`%`' is followed by any character other than '`r`' or a non-zero digit followed by '`r`', that character is used as is in the WFDB path; thus a literal '`%`' can be included in the WFDB path by 'escaping' it as '`%%`'. (Substitutions of '`%`'-strings in the WFDB path were first introduced in WFDB library version 9.7.)

1.8 Exercises

These exercises should require only a few minutes. If you work through them, you will have an opportunity to become acquainted with a few of the most common errors in using the WFDB library.

1. Compile the example program in this chapter and run it. If the WFDB Software Package has not already been installed on your system, download and install the most recent version from PhysioNet first (see Appendix B [Installing the WFDB Software Package], page 115).
2. Find out where database records are kept on your system. What records are available locally?
3. Modify the example program so that you can specify the record to be opened, either as a command-line argument or by having the program prompt you to type a record name. If you are unfamiliar with command-line argument processing, see [Example 2], page 82.

4. Use the modified version of the example to read samples from records `mitdb/200`, `edb/e0103`, `slpdb/slp04`, and `mimicdb/237/237`. The last two of these records have 4 and 6 signals respectively, so you will need to make a few additional changes to the program in order to read these records successfully.
5. Once again using the modified version of the example, what happens if you omit the path information from one of the records in the previous exercise (for example, if you try to open `e0103` instead of `edb/e0103`? Figure out how to set the WFDB path so that the program will work properly in this case. (Hint: use the application `wfdbwhich`, included with the WFDB Software Package, to find the header file for record `edb/e0103`; this information will help you to determine how to set the WFDB path.)
6. If you use MS-DOS or MS-Windows, explore and explain what happens in the previous exercise if you type the record name using upper-case letters, or if you type a '\' (backslash) instead of '/' (forward slash). (Hint: record names are *not* filenames!)
7. What happens when you compile the example program as shown, but with the `#include` statement omitted? with the '`-lwfdb`' ('`-link wfdb`', etc.) omitted?
8. What is the type of the argument to `getvec`? Why can't `getvec` simply return the value it reads, as in '`v = getvec()`'?

2 WFDB Library Functions

This chapter describes the functions that are available to programs compiled with the WFDB library. The functions are introduced in several groups, with examples to illustrate their usage.

About these functions

Each function description begins with an ANSI C function prototype, which specifies the types of any arguments as well as the type of the quantity returned by the function (see *K&R*, pp. 217–218). Note that many of these functions take pointer arguments. These can be traps for newcomers to C. Study the examples carefully! Often a function will return information to the caller in a variable or structure to which the pointer argument points. **It is necessary in such cases for the caller to allocate storage for the variables or structures and to initialize the pointers so that they point to the allocated storage. If you fail to do so, the compiler probably will not warn you of your error; instead your program will fail mysteriously, probably with a core dump and an "illegal memory reference" error message.**

With few exceptions, WFDB library functions return integers that indicate success or failure. The tables that follow the function prototypes list the possible returns and their meanings. By convention, a return code of −1 indicates end-of-file on input files, and no error message is printed. Other negative return codes signify other types of errors, and are usually accompanied by descriptive messages on the standard error output (but see [`wfdbquiet` and `wfdbverbose`], page 45). Zero may indicate success or failure, depending on context (see the descriptions of the individual functions below). Positive codes (returned by only a few functions) always indicate success.

A comprehensive discussion of database files appears later in this guide (see Chapter 5 [Database Files], page 71). Most readers should not need to learn about the gruesome details of how the data are actually stored. You should know, however, that there are files that contain digitized signals, other files that contain annotations, and still others (called *header* files) that describe attributes of the signals such as sampling frequency. The database path lists directories in which database files are found; the WFDB library functions can find them given only the record (and annotator) names, provided that `WFDB` has been properly set (see Section 1.4 [WFDB path], page 12). WFDB library functions responsible for opening signal files find them by reading the header file (which contains their names) first.

The first two sections of this chapter describes functions that extract information from header files in order to gain access to signal and annotation files, and functions that control how these files are read and written. The following two sections describe functions that read and write signal and annotation files. Many readers will not need to go any further; the remaining sections deal with special-purpose functions that exist to serve unusual applications.

2.2 Selecting Database Records

annopen

```
int annopen(char *record, const WFDB_Anninfo *aiarray,
            unsigned int nann)
```

Return:

0	Success
-3	Failure: unable to open input annotation file
-4	Failure: unable to open output annotation file
-5	Failure: illegal `stat` (in *aiarray*) specified for annotation file
-6	Failure: unable to sort output annotations (only if *nann* is 0)
-7	Failure: error writing annotation file (only if *nann* is 0)

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, `annopen` closes any previously opened annotation files, and takes all of *record* as the record name. *aiarray* is a pointer to an array of `WFDB_Anninfo` structures (see Section 3.3 [Annotator Information Structures], page 60), one for each annotator to be opened. *nann* is the number of `WFDB_Anninfo` structures in *aiarray*. The caller must fill in the `WFDB_Anninfo` structures 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 *aiarray*. *Annotator numbers* (for both input and output annotators) are assigned in the order in which the annotators appear in *aiarray*. For example, this code fragment

```
...
char *record = "100s";
WFDB_Anninfo a[3];

a[0].name = "a"; a[0].stat = WFDB_READ;
a[1].name = "b"; a[1].stat = WFDB_WRITE;
a[2].name = "c"; a[2].stat = WFDB_READ;
if (annopen(record, a, 3) < 0)
...
```

attempts 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 `getann(1, &annot)` (see [`getann`], page 29) will read an annotation from annotator `c`, and `putann(0, &annot)` will write an annotation for annotator `b`. Input annotation files will be found if they are located in any of the directories specified by `WFDB` (see Section 1.4 [WFDB path], page 12); output annotators are created in the current directory. Several of the example programs in chapter 6 illustrate the use of `annopen`; for example, see [Example 1], page 81.

To close all annotation files and check that any output annotations were written successfully, invoke `annopen("", NULL, 0)`. (Prior to WFDB library version 10.7.0, this would always return 0.) This can also be useful to force open annotation files to be closed without closing open signal files.

isigopen

`int isigopen(char *record, WFDB_Siginfo *siarray, int nsig)`

Return:

`>0`	Success: the returned value is the number of input signals (i.e., the number of valid entries in *siarray*)
`0`	Failure: no input signals available
`-1`	Failure: unable to read header file (probably incorrect record name)
`-2`	Failure: incorrect header file format
`-3`	Failure: internal limits exceeded (all signal files closed)

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, `isigopen` closes any previously opened input signal files, and takes all of *record* as the record name. If the record name is '`-`', `isigopen` reads the standard input rather than a `hea` file. *Siarray* is a pointer to an array of `WFDB_Siginfo` structures (see Section 3.1 [Signal Information Structures], page 58), one for each signal to be opened.

As a special case, if *nsig* is 0, *siarray* can be `NULL`. In this case, `isigopen` closes any open input signals, then returns the number of signals in *record* without opening them. Use this feature to determine the amount of storage needed for signal-related variables, as in the example below, or to force open input signal files to be closed without closing open annotation or output signal files. This action 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 `strtim` that depend on these quantities will work properly.

If `nsig` is greater than 0, `isigopen` normally returns the number of input signals it actually opened, which may be less than *nsig* but is never greater than *nsig*. The caller must allocate storage for the `WFDB_Siginfo` structures; `isigopen` will fill them in with information about the signals. *Signal numbers* are assigned in the order in which signals are specified in the `hea` file for the record; on return from `isigopen`, information for signal *i* will be found in *siarray[i]*. For example, we can read the `gain` attributes of each signal in record `100s` like this:

```
#include <stdio.h>
#include <wfdb/wfdb.h>

main()
{
    int i, nsig;
    WFDB_Siginfo *siarray;

    nsig = isigopen("100s", NULL, 0);
    if (nsig < 1)
        exit(1);
    siarray = (WFDB_Siginfo *)malloc(nsig * sizeof(WFDB_Siginfo));
```

```
    nsig = isigopen("100s", siarray, nsig);
    for (i = 0; i < nsig; i++)
        printf("signal %d gain = %g\n", i, siarray[i].gain);
    exit(0);
}
```

(See `http://physionet.org/physiotools/wfdb/examples/pgain.c` for a copy of this program.)

This program, unlike the example in the previous chapter, does not assume that the number of signals is known. The first invocation of `isigopen` determines this number (and the program quits if there are no signals). Next, the program allocates the array for the signal information, and then it opens the signals using the second invocation of `isigopen`, passing in the pointer `siarray` and the number of signals determined from the first call (`nsig`).

An error message is produced if `isigopen` is unable to open *any* of the signals listed in the header file. It is not considered an error if only some of the signals can be opened, however. A signal will not be opened if its signal file is unreadable, if an input buffer cannot be allocated for it, or if opening all of the signals in its group would exceed the limits defined by *nsig*. (Note, however, that most records have only one signal group; as a consequence, `isigopen` fails if *nsig* is less than the total number of signals in such cases.) If necessary, the caller can inspect the file names and signal descriptions in *siarray* to determine which signals were opened; see Section 3.1 [Signal Information Structures], page 58. Several of the example programs in chapter 6 illustrate the use of `isigopen`; for example, see [Example 5], page 86.

If the overall set of open signals exceeds internal limits (for example, the total number of samples per frame is greater than `INT_MAX`), `isigopen` returns -3 and closes *all* previously-opened input signals.

If *nsig* is less than 0, `isigopen` fills in up to $-nsig$ members of *siarray*, based on information from the header file for *record*, but *no signals are actually opened*. The value returned in this case is the number of signals named in the `hea` file. Note, however, that there is no guarantee that all (or indeed any) of the signals named in the `hea` file are available to be opened. The features described in this paragraph were first introduced in version 4.4 of the WFDB library.

osigopen

`int osigopen(char *record, WFDB_Siginfo *siarray, unsigned int nsig)`

Return:

`>0`	Success: the returned value is the number of output signals; this number should match *nsig*
`-1`	Failure: unable to read header file
`-2`	Failure: incorrect header file format
`-3`	Failure: unable to open output signal(s)

This function opens output signal files. Use it only if signals are to be *written* using `putvec`. The signal specifications, including the file names, are read from the header file for a specified record. Unmodified MIT or AHA database `hea` files cannot be used, since `osigopen` would

attempt to overwrite the (write-protected) signal files named within. 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. If the record name is '-', `osigopen` reads the standard input rather than a `hea` file. *siarray* is a pointer to an uninitialized array of `WFDB_Siginfo` structures; *siarray* must contain at least *nsig* members. The caller must allocate storage for the `WFDB_Siginfo` structures. On return, `osigopen` will have filled in the `WFDB_Siginfo` structures with the signal specifications.

No more than *nsig* (additional) output signals will be opened by `osigopen`, even if the header file contains specifications for more than *nsig* signals. For example, this code fragment

```
...
WFDB_Siginfo s[2];
int i, nsig;

nsig = osigopen("8l", s, 2);
for (i = 0; i < nsig; i++)
    printf("signal %d will be written into `%s'\n", i, s[i].fname);
...
```

creates 2 output signals named `data0` and `data1` (see Section 5.8 [Piped and Local Records], page 76). See [Example 6], page 88, and see [Example 7], page 89, for illustrations of the use of `osigopen`.

As a special case, if *nsig* is 0, *siarray* can be `NULL`. This can be useful to force open output signal files to be closed without closing open annotation or input signal files.

osigfopen

`int osigfopen(const WFDB_Siginfo *`*siarray*`, unsigned int `*nsig*`)`

Return:

`>0`	Success: the returned value is the number of output signals; this number should match *nsig*
`-2`	Failure: error in signal specification (`fname` or `desc` too long, illegal `fmt` or `bsize`, or incorrect signal group assignment)
`-3`	Failure: unable to open output signal(s)
`-4`	Failure: error writing signal file (only if *nsig* is 0)

This function opens output signals as does `osigopen`, but the signal specifications, including the signal file names, are supplied by the caller to `osigfopen`, rather than read from a header file as in `osigopen`. Any previously open output signals are closed by `osigfopen`. *siarray* is a pointer to an array of `WFDB_Siginfo` structures (see Section 3.1 [Signal Information Structures], page 58), one for each signal to be opened. *nsig* is the number of `WFDB_Siginfo` structures in *siarray*.

Before invoking `osigfopen`, the caller must fill in the fields of the `WFDB_Siginfo` structures in *siarray* (see Chapter 3 [Data Types], page 57; the `initval`, `nsamp`, and `cksum` fields may be left uninitialized, however). To make a multiplexed signal file, specify the

same `fname` and `group` for each signal to be included (see Section 5.3 [Multiplexed Signal Files], page 73). For ordinary (non-multiplexed) signal files, specify a unique `fname` and `group` for each signal. See [Example 8], page 92, for an illustration of the use of `osigfopen`.

To close all output signal and header files and check that they were written successfully, invoke `osigfopen(NULL, 0)`. (Prior to WFDB library version 10.7.0, this would always return 0.) This can also be useful to force open output signal files to be closed without closing open annotation or input signal files.

wfdbinit

```
int wfdbinit(char *record,
             const WFDB_Anninfo *aiarray, unsigned int nann,
             WFDB_Siginfo *siarray, unsigned int nsig)
```

Return:

`>0`	Success: the returned value is the number of input signals (i.e., the number of valid entries in *siarray*)
`0`	Annotation files opened successfully, input signals unavailable (not an error for programs that don't need them; no error message is printed if *nsig* is 0)
`-1`	Failure: unable to read header file (probably incorrect record name)
`-2`	Failure: incorrect header file format
`-3`	Failure: unable to open input annotation file
`-4`	Failure: unable to open output annotation file
`-5`	Failure: illegal `stat` (in *aiarray*) specified for annotation file (see Section 3.3 [Annotator Information Structures], page 60)

This function opens database files other than output signal files for a selected record. The code

```
n = wfdbinit(record, a, na, s, ns);
```

is exactly equivalent to

```
n = annopen(record, a, na);
if (n == 0)
    n = isigopen(record, s, ns);
```

Avoid using `wfdbinit` and `setifreq` in the same program. See [Example 9], page 96, for an illustration of the use of `wfdbinit`. See [`osigopen`], page 20, and see [`osigfopen`], page 21, for methods of opening output signal files.

2.3 Special I/O Modes

setifreq

`void setifreq(WFDB_Frequency` *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 `isigopen` or `wfdbinit`), and before using any of `getvec`, `putann`, `isigsettime`, `isgsettime`, `timstr`, `mstimstr`, or `strtim`. *Note that the operation of* `getframe` *is unaffected by* `setifreq`.

Use `setifreq` when your application requires input samples at a specific frequency. After invoking `setifreq`, `getvec` resamples the digitized signals from the input signals at the desired frequency (see [getvec], page 27), and all of the WFDB library functions that accept or return times in sample intervals automatically convert between the actual sampling intervals and those corresponding to the desired frequency. This slightly elaborated version of the example program from the previous chapter invokes `setifreq`, passing it the desired sampling frequency from the command line, then prints the samples in record 100s, beginning 1 second (`t0`) and ending 2 seconds (`t1`) from the beginning of the record:

```
#include <stdio.h>
#include <wfdb/wfdb.h>

main(int argc, char **argv)
{
    WFDB_Frequency f = (WFDB_Frequency)0;
    WFDB_Sample v[2];
    WFDB_Siginfo s[2];
    WFDB_Time t, t0, t1;

    if (argc > 1) sscanf(argv[1], "%lf", &f);
    if (f <= (WFDB_Frequency)0) f = sampfreq("100s");

    if (isigopen("100s", s, 2) < 1)
        exit(1);
    setifreq(f);
    t0 = strtim("1");
    isigsettime(t0);
    t1 = strtim("2");
    for (t = t0; t <= t1; t++) {
        if (getvec(v) < 0)
            break;
        printf("%d\t%d\n", v[0], v[1]);
    }
    exit(0);
}
```

(See `http://physionet.org/physiotools/wfdb/examples/psamplex.c` for a copy of this program. Compile it as shown in the previous chapter, then run it using a command such

as '`psamplex 100`'.) The QRS detector in chapter 6 also illustrates the use of `setifreq` (see [Example 10], page 98).

Avoid using `wfdbinit` and `setifreq` in the same program.

getifreq

```
WFDB_Frequency getifreq(void)
```

Return:

`(WFDB_Frequency)`
: the 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 `sampfreq`, see [sampfreq], page 46), or the frequency chosen using a previous invocation of `setifreq`.

setgvmode

```
void setgvmode(int *mode)
```

This function sets the mode used by `getvec` when reading a multi-frequency record (see Section 5.4 [Multi-Frequency Records], page 73). If *mode* is `WFDB_LOWRES`, `getvec` decimates any signals sampled at multiples of the frame rate, so that one sample is returned per signal per frame (i.e., the oversampled signals are resampled by simple averaging of the samples for each signal within each frame). If *mode* is `WFDB_HIGHRES`, each sample of any oversampled signal is returned by successive invocations of `getvec`, and each sample of any signal sampled at a lower frequency is returned by two or more successive invocations of `getvec` (i.e., the less frequently sampled signals are resampled using zero-order interpolation). `getvec` operates in `WFDB_LOWRES` mode by default. `WFDB_LOWRES` and `WFDB_HIGHRES` are defined in `<wfdb/wfdb.h>`.

In WFDB library version 9.6 and later versions, `setgvmode` also affects how annotations are read and written. If `setgvmode(WFDB_HIGHRES)` is invoked *before* using `annopen`, `wfdbinit`, `getvec`, `sampfreq`, `strtim`, or `timstr`, then all `WFDB_Time` data (including the `time` attributes of annotations read by `getann` or written by `putann`) visible to the application are in units of the high-resolution sampling intervals. (Otherwise, `WFDB_Time` data are in units of frame intervals.)

Version 10.4 and later versions of the WFDB library support two modes of handling invalid or missing samples. By default, `getframe`, `getvec`, and `sample` return the special value `WFDB_INVALID_SAMPLE` in such cases. If *mode* is `WFDB_GVPAD + WFDB_LOWRES` or `WFDB_GVPAD + WFDB_HIGHRES`, however, these functions replicate the previous valid sample whenever they encounter an invalid or missing sample, which may simplify the design of applications such as digital filters. The constant `WFDB_GVPAD` is defined in `<wfdb/wfdb.h>`.

getgvmode

```
int setgvmode(void)
```

This function returns the operating mode used by `getvec`. If the returned value has the `WFDB_HIGHRES` bit set, `getvec` is operating in high-resolution mode; if the returned value has the `WFDB_GVPAD` bit set, `getvec` replicates the previous valid sample whenever it

encounters an invalid or missing sample, rather than returning the value `WFDB_INVALID_SAMPLE`. The constants `WFDB_HIGHRES`, `WFDB_GVPAD`, and `WFDB_INVALID_SAMPLE` are defined in `<wfdb/wfdb.h>`.

getspf

```
int getspf(void)
```

Return:

`(int)` the number of samples per signal per frame

Unless the application is operating in `WFDB_HIGHRES` mode (see [setgvmode], page 24) 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, `getspf` returns the number of samples per signal per frame (hence `sampfreq(NULL)/getspf()` is the number of frames per second).

setiafreq

```
void setiafreq(WFDB_Annotator an, WFDB_Frequency frequency)
```

This function sets the time resolution (number of ticks per second) used by `getann` and `ungetann` for the given input annotator. By default, the time resolution equals the input sampling frequency (see [getifreq], page 24) at the time `annopen` is called. After calling this function, the `time` fields of subsequent annotations will be scaled according to the new time resolution.

getiafreq

```
WFDB_Frequency getiafreq(WFDB_Annotator an)
```

Return:

`(WFDB_Frequency)>0.`
: Success: the annotation time resolution in Hz

`(WFDB_Frequency)-2.`
: Failure: incorrect annotator number specified

This function returns the current time resolution of the given input annotator. The time resolution equals the input sampling frequency by default, but may be changed (see [setiafreq], page 25).

getiaorigfreq

```
WFDB_Frequency getiaorigfreq(WFDB_Annotator an)
```

Return:

`(WFDB_Frequency)>0.`
: Success: the annotation time resolution in Hz

`(WFDB_Frequency)0.`
: Failure: the annotation time resolution is not defined

`(WFDB_Frequency)-2.`
: Failure: incorrect annotator number specified

This function returns the original time resolution for the given input annotator, if it was specified by the application that created the annotation file (see [setafreq], page 26).

If the application that created the annotation file did not specify a time resolution, `getiaorigfreq` returns zero. (In this case, the time resolution is assumed to equal the record's frame frequency.)

setafreq

`void setafreq(WFDB_Frequency` *`frequency`*`)`

This function sets the time resolution, in ticks per second, for any output annotation files created after it has been invoked. By default, the time resolution is equal to the input sampling frequency (and `setifreq` invokes `setafreq` to maintain this behavior if the input sampling frequency is changed).

This function has no effect on output annotation files that are already open when it is invoked, nor on annotations read from input annotation files (for which the `time` fields always are expressed in units equal to the input sample intervals).

getafreq

`WFDB_Frequency getafreq(void)`

Return:

`>0`	output annotation time resolution, if previously set by `setafreq`
`0`	otherwise

This function returns the current output annotation time resolution in ticks per second if it has been set using `getafreq`.

2.4 Reading and Writing Signals and Annotations

getvec

`int getvec(WFDB_Sample *vector)`

Return:

`>0`	Success; the returned value is the number of input signals (the number of valid entries in *vector*)
`-1`	End of data (contents of *vector* not valid)
`-3`	Failure: unexpected physical end of file
`-4`	Failure: checksum error (detected only at end of file)

This function reads a sample from each input signal. The caller should allocate storage for an array of `WFDB_Sample`s (integers) and pass a pointer to this array to `getvec`. (The length of the array must be no less than the number of input signals, as obtained from `isigopen` or `wfdbinit`.) On return, *vector[i]* contains the next sample from signal *i*. For example, this modified version of the example from chapter 1 reads and prints the first ten samples of each available input signal:

```
#include <stdio.h>
#include <wfdb/wfdb.h>

main()
{
    int i, j, nsig;
    WFDB_Sample *v;
    WFDB_Siginfo *s;

    nsig = isigopen("100s", NULL, 0);
    if (nsig < 1)
        exit(1);
    s = (WFDB_Siginfo *)malloc(nsig * sizeof(WFDB_Siginfo));
    if (isigopen("100s", s, nsig) != nsig)
        exit(1);
    v = (WFDB_Sample *)malloc(nsig * sizeof(WFDB_Sample));
    for (i = 0; i < 10; i++) {
        if (getvec(v) < 0)
            break;
        for (j = 0; j < nsig; j++)
            printf("%8d", v[j]);
        printf("\n");
    }
    exit(0);
}
```

(See `http://physionet.org/physiotools/wfdb/examples/exgetvec.c` for a copy of this program.)

Notice how the value returned by the first invocation of `isigopen` is used to determine how many input signals there are. Several of the example programs in chapter 6 illustrate the use of `getvec`; for example, see [Example 6], page 88.

If `setifreq` has been used to modify the input sampling rate, `getvec` resamples the input signals at the desired rate, using linear interpolation between the pair of samples nearest in time to that of the sample to be returned. The results will generally be satisfactory, provided that the original signals do not contain frequencies near or above the Nyquist limit (half of the desired sampling frequency). If this is a concern, you may wish to low-pass filter the input signals using, for example, '`fir`' (see the *WFDB Applications Guide*) before resampling them. If you use `setifreq` to *increase* the sampling frequency by a large factor, you may wish to filter the resampled signals within your application to remove harmonics of the original sampling frequency introduced by resampling.

getframe

```
int getframe(WFDB_Sample *vector)
```

Return:

`>0`	Success; the returned value is the number of input signals
`-1`	End of data (contents of *vector* not valid)
`-3`	Failure: unexpected physical end of file
`-4`	Failure: checksum error (detected only at end of file)

This function reads a vector of samples, including at least one sample from each open input signal. If all signals are sampled at the same frequency, only one sample is read from each signal. Otherwise, signals sampled at multiples of the frame frequency are represented by two or more consecutive elements of the returned *vector*. For example, if the frame frequency is 125 Hz, signal 0 is sampled at 500 Hz, and the remaining 3 signals are sampled at 125 Hz each, then the returned *vector* has 7 valid components: the first 4 are samples of signal 0, and the remaining 3 are samples of signals 1, 2, and 3. The caller should allocate storage for an array of `WFDB_Sample`s (integers) and pass a pointer to this array to `getframe`. The length of *vector* must be determined by summing the values of the `spf` (samples per frame) fields in the `WFDB_Siginfo` structures associated with the input signals (see [`isigopen`], page 19).

putvec

```
int putvec(const WFDB_Sample *vector)
```

Return:

`>0`	Success: the returned value is the number of output signals (the number of entries in *vector* that were written)
`0`	Slew rate too high for one or more signals (difference format only; the DC level(s) will be corrected as soon as the slew rate permits)
`-1`	Failure: write error

This function writes a sample to each input signal. The caller should fill an array of `WFDB_Sample`s with the samples and pass a pointer to this array to `putvec`. (The length of

the array must be no less than the number of output signals, as given to `osigfopen` or `osigopen`.) On entry, *vector[i]* contains the next sample from signal *i*. For example, this modified version of the previous example (see [`getvec`], page 27) copies the first ten samples of each available input signal:

```
#include <stdio.h>
#include <wfdb/wfdb.h>

main()
{
    int i, j, nsig;
    WFDB_Sample *v;
    WFDB_Siginfo *s;

    nsig = isigopen("100s", NULL, 0);
    if (nsig < 1)
        exit(1);
    s = (WFDB_Siginfo *)malloc(nsig * sizeof(WFDB_Siginfo));
    if (isigopen("100s", s, nsig) != nsig ||
        osigopen("8l", s, nsig) != nsig)
        exit(1);
    v = (WFDB_Sample *)malloc(nsig * sizeof(WFDB_Sample));
    for (i = 0; i < 10; i++)
        if (getvec(v) < 0 || putvec(v) < 0)
            break;
    wfdbquit();
    exit(0);
}
```

(See `http://physionet.org/physiotools/wfdb/examples/exputvec.c` for a copy of this program.)

All programs that write signals or annotations *must* invoke `wfdbquit` to close the output files properly (see [`wfdbquit`], page 44). This example uses record `8l` (see Section 5.8 [Piped and Local Records], page 76) for the output signal specifications; the output signal files will be named `data0` and `data1` in the current directory. Several of the example programs in chapter 6 illustrate the use of `putvec`; for example, see [Example 6], page 88.

Note that prior to WFDB library version 10.7.0, `putvec` would modify the input vector in some cases (the *vector* argument was not declared as `const`.)

getann

`int getann(WFDB_Annotator` *an*`, WFDB_Annotation *`*annot*`)`

Return:

`0`	Success
`-1`	End of file (**annot* is not valid)
`-2`	Failure: incorrect annotator number specified
`-3`	Failure: unexpected physical end of file

This function reads the next annotation from the input annotator specified by *an* into the annotation structure (see Section 3.4 [Annotation Structures], page 61) pointed to by *annot*. The caller must allocate storage for the annotation structure. Input annotators are numbered 0, 1, 2, etc. This short program uses `getann` to read the contents of the reference (`atr`) annotation file for record `100s`:

```
#include <stdio.h>
#include <wfdb/wfdb.h>

main()
{
    WFDB_Anninfo a;
    WFDB_Annotation annot;

    a.name = "atr"; a.stat = WFDB_READ;
    if (annopen("100s", &a, 1) < 0)
        exit(1);
    while (getann(0, &annot) == 0)
        printf("%s %s\n", mstimstr(annot.time), annstr(annot.anntyp));
    exit(0);
}
```

(See `http://physionet.org/physiotools/wfdb/examples/exgetann.c` for a copy of this program.)

See Section 3.3 [Annotator Information Structures], page 60, for information on the contents of the `WFDB_Anninfo` structure, and see [`mstimstr`], page 36, and see [`annstr`], page 34, for details of the functions used to print portions of the annotations read by `getann` in this example.

ungetann

`int ungetann(WFDB_Annotator` *`an`*`, const WFDB_Annotation *`*`annot`*`)`

Return:

`0`	Success
`-1`	Failure: push-back buffer full (`*`*`annot`* was not pushed back)
`-2`	Failure: incorrect annotator number specified

This function arranges for the annotation structure pointed to by *annot* to be the next one read by `getann` from input annotator *an*. The pushed-back annotation need not necessarily be one originally read by `getann`. No more than one annotation may be pushed back at a time for each input annotator. (This function was first introduced in WFDB library version 5.3.)

putann

`int putann(WFDB_Annotator` *`an`*`, const WFDB_Annotation *`*`annot`*`)`

Return:

`0`	Success

`-1` Failure: write error

`-2` Failure: incorrect annotator number specified

This function writes the next annotation for the output annotator specified by *an* from the annotation structure pointed to by *annot*. Output annotators are numbered 0, 1, 2, etc. The caller must fill in all fields of the annotation structure. Using version 9.7 and later versions of the WFDB library, annotations may be written in any order (see Section 5.10 [Annotation Order], page 78). Earlier versions require that annotations be supplied to `putann` in canonical order, and return an error code of `-3` if an out-of-order annotation is supplied. All programs that write signals or annotations *must* invoke `wfdbquit` to close the output files properly (see [`wfdbquit`], page 44). Several of the example programs in chapter 6 illustrate the use of `putann`; for example, see [Example 1], page 81.

2.5 Non-Sequential Access to WFDB Files

The next three functions permit random access to signal and annotation files. It is not possible, however, to skip backwards on piped input.

isigsettime

```
int isigsettime(WFDB_Time t)
```

Return:

`0`	Success
`-1`	Failure: EOF reached or improper seek

This function resets the input signal file pointers so that the next samples returned from `getvec` will be those with sample number `|`t`|`. Only the magnitude of t is significant, not its sign; hence values returned by `strtim` can always be used safely as arguments to `isigsettime` (see [`timstr` and `strtim`], page 36). This function will fail if a pipe is used for input and `|`t`|` is less than the current sample number. See [Example 7], page 89, and see [Example 9], page 96, for illustrations of the use of `isigsettime`.

isgsettime

```
int isgsettime(WFDB_Group sgroup, WFDB_Time t)
```

Return:

`0`	Success
`-1`	Failure: EOF reached or improper seek
`-2`	Failure: incorrect signal group number specified

This function does the job of `isigsettime`, but only for the signal group specified by *sgroup*. This function may be of use if more than one record is open simultaneously (see Section 5.6 [Multiple Record Access], page 75).

tnextvec

```
WFDB_Time tnextvec(WFDB_Signal s, WFDB_Time t)
```

Return:

`>= 0`	Time of the next valid sample of signal s at or after t
`-1`	Failure: EOF reached or improper seek

This function resets the input signal file pointers so that the next samples read by `getvec` will include the next valid sample of the specified signal s occurring at or after t. Use `tnextvec` to skip lengthy gaps in a signal of interest efficiently.

iannsettime

```
int iannsettime(WFDB_Time t)
```

Return:

`0`	Success
`-1`	Failure: EOF reached or improper seek

`-3` Failure: unexpected physical end of file

This function resets the input annotation file pointers so that the next annotation read by `getann` from each input annotation file will be the first occurring on or after sample number `|`t`|` in that file. Only the magnitude of t is significant, not its sign; hence values returned by `strtim` can always be used safely as arguments to `iannsettime` (see [`timstr` and `strtim`], page 36). This function will fail if a pipe is used for input and `|`t`|` is less than the time of the most recent annotation read from the pipe. See [Example 9], page 96, for an illustration of the use of `iannsettime`.

sample and sample_valid

```
WFDB_Sample sample(WFDB_Signal s, WFDB_Time t)
int sample_valid(void)
```

Return:

n	(from `sample`): The value (in raw adus) of sample number t in open signal s,if successful, or the value of the previous successfully read sample.
`1`	(from `sample_valid`): The most recent value returned by `sample` was valid
`0`	(from `sample_valid`): The most recent t given to `sample` follows the end of the record
`-1`	(from `sample_valid`): The most recent value returned by `sample` was invalid (because signal s is not available at time t)

The `sample` function allows the caller to read samples of the currently open input signals in any order. The first argument is a signal number (a non-negative integer between 0 and *nsig*-1, where *nsig* is the number of open input signals), and the second is a time, expressed as a non-negative sample number. If `sample` is invoked with valid input arguments, the companion function `sample_valid` returns 1.

The `sample_valid` function can be used to check the results of the most recent invocation of `sample`. If `sample_valid` returns 1, the latest value returned by `sample` is valid. `sample_valid` returns 0 if `sample` has most recently returned a padded value following the end of the record. This allows code that uses `sample` to use the condition '`sample_valid() != 0`' (or simply '`sample_valid()`' to determine if more samples are available. If `sample_valid` returns -1, the most recent value returned by `sample` was `WFDB_INVALID_SAMPLE` (because the requested signal s was unavailable at the requested time t). Use the condition '`sample_valid() > 0`' to check if the most recent value returned by `sample` is a valid value (e.g., suitable for inclusion in a running average or similar calculation).

For an example of the use of `sample` and `sample_valid`, see [Example 7], page 89.

Be sure to call `wfdbquit` before exiting from any program that uses `sample`, to be certain that dynamically allocated memory used by `sample` is freed.

2.6 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.

annstr, anndesc, and ecgstr

```
char *annstr(int code)
char *anndesc(int code)
char *ecgstr(int code)
```

Return:

`(char *)` pointer to a printable string that describes the code, or `NULL`

These functions translate the annotation code specified by their argument into a string (see Chapter 4 [Annotation Codes], page 67). Illegal or undefined codes are translated by `annstr` and `ecgstr` into decimal numerals surrounded by brackets (e.g., '`[55]`'); `anndesc` returns `NULL` in such cases. The strings returned by `annstr` are mnemonics (usually only one character), which may be modified either by `setannstr` or by the presence of *modification labels* in an input annotation file (see [`setannstr`], page 34). The strings returned by `anndesc` are brief descriptive strings, usually those given in the table of annotation codes (see Chapter 4 [Annotation Codes], page 67). The strings returned by `ecgstr` are usually the same as those returned by `annstr`, but they can be modified only by `setecgstr`, and not by the presence of modification labels as for `annstr`. The intent is that `ecgstr` should be used rather than `annstr` only when it is necessary that a fixed set of mnemonics be used, independent of any modification labels.

Here is a little program that prints a table of the codes, mnemonic strings, and descriptions:

```
#include <stdio.h>
#include <wfdb/wfdb.h>
#include <wfdb/ecgcodes.h>

main()
{
    int i;

    printf("Code\tMnemonic\tDescription\n");
    for (i = 1; i <= ACMAX; i++) {
        printf("%3d\t%s", i, annstr(i));
        if (anndesc(i) != NULL)
            printf("\t\t%s", anndesc(i));
        printf("\n");
    }
}
```

(See `http://physionet.org/physiotools/wfdb/examples/exannstr.c` for a copy of this program.)

`ACMAX` is defined in `<wfdb/ecgcodes.h>`. The range from 1 through `ACMAX` includes all legal annotation codes; if you run this program, you will find some undefined but legal

annotation codes in this range. See [Example 3], page 84, for another illustration of the use of `annstr`. (`annstr` and `anndesc` were first introduced in WFDB library version 5.3.)

strann and strecg

```
int strann(const char *string)
int strecg(const char *string)
```

Return:

`(int)` annotation code

These functions translate the null-terminated ASCII character strings to which their arguments point into annotation codes. Illegal strings are translated into `NOTQRS`. Input strings for `strann` and `strecg` should match those returned by `annstr` and `ecgstr` respectively. See [Example 9], page 96, for an illustration of the use of `strann`. (`strann` was first introduced in WFDB library version 5.3.)

setannstr, setanndesc, and setecgstr

```
int setannstr(int code, const char *string)
int setanndesc(int code, const char *string)
int setecgstr(int code, const char *string)
```

Return:

`0` Success

`-1` Failure: illegal `code`

These functions modify translation tables used by functions that convert between annotation codes and strings. `setannstr` modifies the table shared by `annstr` and `strann`; `setanndesc` modifies the table used by `anndesc`; and `setecgstr` modifies the table shared by `ecgstr` and `strecg`. They may be used to redefine strings for defined annotation codes as well as to define strings for undefined annotation codes. For example, `setannstr(NORMAL, "\\267")` redefines the string for normal beats as a PostScript bullet, '•' (`NORMAL` is defined in `<wfdb/ecgcodes.h>`).

An important difference between `setannstr` (or `setanndesc`) and `setecgstr` is that `annopen` and `wfdbinit` insert modification labels in any output annotation files that are created *after* invoking `setannstr` or `setanndesc`; `setecgstr` does not have this side effect. By using `setannstr` before `annopen`, a WFDB application may create annotation files with self-contained code tables, which can be read properly by other WFDB applications without the need to inform them explicitly about non-standard codes. For this scheme to work as intended, all custom code mnemonics and descriptions must be defined before the output annotation files are opened.

By passing a negative value as *code* to `setannstr` or `setanndesc`, the translation for $-$*code* can be modified without triggering the generation of a modification label. This feature can be useful for programs that use alternate sets of mnemonics or descriptions for speakers of different languages.

Note that it is possible, though not desirable, to define identical strings for two or more codes; the behavior of `strann` and `strecg` in such cases is implementation-dependent. (`setannstr` and `setanndesc` were first introduced in WFDB library version 5.3.)

The next three functions convert between "standard time format" strings and times in units of sample intervals. Normally they should be invoked after `isigopen`, `wfdbinit`, or `sampfreq`, any of which will determine the duration of a sample interval and the base time from a header file, or after defining these quantities using `setsampfreq` and `setbasetime`. If this is not done, or if these time-conversion functions are used after `wfdbquit`, they will perform conversions in units of seconds (i.e., the sample interval is taken to be one second in such cases).

[ms]timstr

```
char *timstr(WFDB_Time t)
char *mstimstr(WFDB_Time t)
```

Return:

`(char *)` pointer to a string that represents the time

These functions convert times or time intervals into null-terminated ASCII strings. If the argument, *t*, is greater than zero, it is treated as a time interval, and converted directly into *HH:MM:SS* format by `timstr`, or to *HH:MM:SS.SSS* format by `mstimstr`, with leading zero digits and colons suppressed. If *t* is zero or negative, it is taken to represent negated elapsed time from the beginning of the record, and it is converted to a time of day using the base time for the record as indicated by the `hea` file or the caller (see [`setbasetime`], page 46); in this case, if the base time is defined, the string will contain all digits even if there are leading zeroes, it will include the date if a base date is defined, and it will be marked as a time of day by being bracketed (e.g., '`[08:45:00 23/04/1989]`'). The result of the conversion is truncated to a multiple of a second by `timstr`, or to a multiple of a millisecond by `mstimstr`. Note in each case that the returned pointer addresses static data (shared by `timstr` and `mstimstr`), the contents of which are overwritten by subsequent calls. See [Example 3], page 84, for an illustration of the use of `mstimstr`; also see [Example 5], page 86, for an example of the use of `timstr`.

strtim

```
WFDB_Time strtim(const char *string)
```

Return:

`(WFDB_Time) >0`
: number of sample intervals corresponding to the argument interpreted as a time interval

`(WFDB_Time) <0`
: (negated) elapsed time in sample intervals from the beginning of the record, corresponding to the argument interpreted as a time of day

`(WFDB_Time) 0`
: a legal return if the argument matches the base time; otherwise an error return indicating an incorrectly formatted argument

This function converts an ASCII string in *standard time format* to a time in units of sample intervals. Examples of standard time format:

`2:14.875` 2 minutes + 14.875 seconds

`[13:6:0]` 13:06 (1:06 PM)

`[8:0:0 1]` 8 AM on the day following the base date

`[12:0:0 1/3/1992]`
noon on 1 March 1992

`143` 143 seconds (2 minutes + 23 seconds)

`4:02:01` 4 hours + 2 minutes + 1 second

`s12345` 12345 sample intervals

`c350.5` counter value 350.5

`e` time of the end of the record (if defined)

`i` time of the next sample in input signal 0

`o` (the letter 'o') time of the next sample in output signal 0

If the argument is bracketed (as in the second, third, and fourth examples), it is taken as a time of day, and `strtim` uses the base time defined by the header file or by the caller (see [`setbasetime`], page 46); in this case, the value returned is zero or negative (and can be converted into elapsed time from the beginning of the record by simply negating it). If the argument is not bracketed, it is taken as a time interval, and converted directly into a positive number of sample intervals. These notations match those used by `timstr` and `mstimstr`, which are (approximately) inverse functions of `strtim`; in fact, for MIT DB and AHA DB records (and any others with sampling frequencies below 1 KHz), `strtim(mstimstr(t))` $= t$, for any t. The '`s`'-format (as in the seventh example above) is provided to allow "conversion" of time intervals already expressed in sample intervals. The similar '`c`'-format converts counter values (see [`getcfreq`], page 47) into sample intervals. The length of the record in sample intervals can be obtained using `strtim("e")`, which evaluates to zero if this quantity is undefined. The sample number of the next sample to be read or written can be determined using `strtim("i")` or `strtim("o")`. If the argument string is incorrectly formatted, `strtim` returns zero (indistinguishable from a correct input that evokes a zero output); this may be considered a feature. Several of the programs in chapter 6 illustrate the use of `strtim` (for example, see [Example 7], page 89).

The next two functions convert between Julian dates and ASCII strings. Julian dates as defined by astronomers begin at noon GMT; these begin at midnight local time.

datstr

```
char *datstr(WFDB_Date date)
```

Return:

`(char *)` pointer to a string that represents the date

This function converts the Julian date represented by *date* into an ASCII string in the form *DD/MM/YYYY*.

strdat

```
WFDB_Date strdat(const char *string)
```

Return:

`(WFDB_Date)`
: Julian date corresponding to the argument

This function converts *string* into a Julian date. The argument should be in the format used by `datstr`; if *string* is improperly formatted, `strdat` returns zero. Note that dates such as '`15/3/89`' refer to the first century A.D., not the twentieth. For example, the interval in days between the events commemorated by the French and American national holidays is `strdat("14/7/1789")` – `strdat("4/7/1776")`.

The next four functions convert between analog-to-digital converter (ADC) units and physical units, using as a conversion factor the gain for the specified input signal. The first two (`aduphys` and `physadu`) are general-purpose functions that convert absolute levels (i.e., they account for non-zero `baseline` values); the last two (`adumuv` and `muvadu`) are for use with millivolt-dimensioned signals only, and convert potential differences (i.e., `adumuv(`*s*`, 0)` = `muvadu(`*s*`, 0)` = 0 for all *s*, irrespective of the `baseline` values specified in the header file). Normally, these functions should be invoked after `isigopen` or `wfdbinit`, either of which will determine the gain from the `hea` file. If this is not done, or if the header file indicates that the gain is uncalibrated, or if the specified input signal is not currently open, a gain of `WFDB_DEFGAIN` (defined in `<wfdb/wfdb.h>`) ADC units per millivolt, and a baseline of zero, are assumed. If the physical units (see Section 3.1 [Signal Information Structures], page 58) are not millivolts, `adumuv` and `muvadu` convert to and from thousandths of the defined physical units. Note that `adumuv` and `muvadu` deal exclusively with integers, but `aduphys` returns and `physadu` accepts double-precision floating point physical values.

aduphys

```
double aduphys(WFDB_Signal s, WFDB_Sample a)
```

Return:

`(double)` physical value corresponding to a sample value of *a* ADC units

This function converts the sample value *a* from ADC units to physical units, based on the `gain` and `baseline` for input signal *s*. (`aduphys` was first introduced in WFDB library version 6.0.)

physadu

```
WFDB_Sample physadu(WFDB_Signal s, double v)
```

Return:

`(WFDB_Sample)`
: sample value, in ADC units, corresponding to *v*, in physical units

This function converts the value *v* from physical units to ADC units, based on the `gain` and `baseline` for input signal *s*. (`physadu` was first introduced in WFDB library version 6.0.)

adumuv

```
int adumuv(WFDB_Signal s, WFDB_Sample a)
```

Return:

`(int)` number of microvolts corresponding to *a* ADC units

This function converts the potential difference *a* from ADC units to microvolts, based on the `gain` for input signal *s*.

muvadu

```
WFDB_Sample muvadu(WFDB_Signal s, int v)
```

Return:

`(int)` number of ADC units corresponding to *v* microvolts

This function converts the potential difference *v* from microvolts to ADC units, based on the `gain` for input signal *s*.

2.7 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 is maintained in memory and which contains entries for various types of signals.

calopen

```
int calopen(const char *file)
```

Return:

`0`	Success
`-1`	Failure: insufficient memory for calibration list
`-2`	Failure: unable to open calibration file

This function reads the specified calibration *file* (which must be located in one of the directories specified by `WFDB`, see Section 1.4 [WFDB path], page 12) into the calibration list. If *file* is `NULL`, the file named by `WFDBCAL` is read. Normally, the current contents of the calibration list are discarded before reading the calibration file; if *file* begins with '`+`', however, the '`+`' is stripped from the file name and the contents of the file are appended to the current calibration list. If *file* is '`-`', `calopen` reads the standard input rather than a calibration file. (This function was first introduced in WFDB library version 6.0.)

getcal

```
int getcal(const char *desc, const char *units, WFDB_Calinfo *cal)
```

Return:

`0`	Success; `*cal` contains the requested data
`-1`	Failure: no match found

This function attempts to find calibration data for signals of type *desc*, having physical units as given by *units*. If successful, it fills in the contents of the `WFDB_Calinfo` structure (see Section 3.2 [Calibration Information Structures], page 60) pointed to by *cal*. The caller must allocate storage for the `WFDB_Calinfo` structure, and must not modify the contents of the strings addressed by the `sigtype` and `units` fields of the `WFDB_Calinfo` structure after `getcal` returns. `getcal` returns data from the first entry in the calibration list that contains a `sigtype` field that is either an exact match or a prefix of *desc*, and a `units` field that is an exact match of *units*; if either *desc* or *units* is `NULL`, however, it is ignored for the purpose of finding a match. `getcal` cannot succeed unless the calibration list has been initialized by a previous invocation of `calopen` or `putcal`. (This function was first introduced in WFDB library version 6.0.)

putcal

```
int putcal(const WFDB_Calinfo *cal)
```

Return:

`0`	Success
`-1`	Failure: insufficient memory

This function adds the `WFDB_Calinfo` structure pointed to by *cal* to the end of the calibration list. (This function was first introduced in WFDB library version 6.0.)

newcal

```
int newcal(const char *file)
```

Return:

`0`	Success
`-1`	Failure: unable to open *file*

This function creates a new calibration *file* (in the current directory) containing the contents of the calibration list (which is not modified). *file* must satisfy the standard conditions for a WFDB file name, i.e., it may contain letters, digits, or underscores. (This function was first introduced in WFDB library version 6.0.)

flushcal

```
void flushcal()
```

This function discards the current calibration list and returns the memory that it occupied to the heap. Note that `wfdbquit` does *not* perform the function of `flushcal`. (This function was first introduced in WFDB library version 6.0.)

2.8 Miscellaneous WFDB Functions

newheader

```
int newheader(char *record)
```

Return:

0	Success
-1	Failure: unable to create header file

This function creates a `hea` file (in the current directory, unless *record* includes path information). Use `newheader` just after you have finished writing the signal files, but before calling `wfdbquit`. If *record* begins with '+', the '+' is discarded and the remainder of *record* is taken as the record name. Otherwise, all of *record* (excluding any path information) is taken to be the record name. If the record name is '-', the header file is written to the standard output. Record names may include letters in lower or upper case, digits, and underscores ('_'); they may not include any other characters. If *record* does not conform to these requirements, `newheader` will return −1; see [Example 8], page 92, for an illustration of the use of `newheader` to check the validity of a record name. For compatibility with the widest range of operating systems, keep record names short (6 characters or less) and avoid those that are distinguished by case alone. To avoid confusion with MIT DB and AHA DB records, do not use three- or four-digit record names.

After calling `newheader`, you can call `putinfo` to add info strings to the header file. To close the header file, along with output signal files, and check that they were written successfully, invoke `osigfopen(NULL, 0)` (see [`osigfopen`], page 21).

setheader

```
int setheader(char *record, const WFDB_Siginfo *siarray,
              unsigned int nsig)
```

Return:

0	Success
-1	Failure: unable to create header file

This function creates or recreates a header file (in the current directory) for the specified *record*, based on the contents of the first *nsig* members of *siarray*. The preferred way to create a header file for a new record is using `newheader`, which records signal checksum and length variables maintained by `putvec`. The intended use of `setheader` is for editing header files, e.g., to change recorded signal gains from a calibration program, or to add signal descriptions or "info" strings. In the following code fragment, the header file for record `old` is used to create a header file for record `new`:

```
...
int nsig, status;
WFDB_Siginfo *s;

nsig = isigopen("old", NULL, 0);
s = (WFDB_Siginfo *)malloc(nsig * sizeof(WFDB_Siginfo));
nsig = isigopen("old", s, -nsig);
```

```
if (nsig > 0) {
    s[0].gain = 100.0;
    status = setheader("new", s, (unsigned int)nsig);
}
...
```

The header file for record `new` will contain the same signal information as that for record `old`, except that the `gain` for signal 0 will have been changed as shown. Any "info" strings in the `hea` file for record `old` must be copied explicitly; see [`getinfo`], page 50, and see [`putinfo`], page 50. (This function was first introduced in WFDB library version 5.0.)

To close the header file, along with output signal files, and check that they were written successfully, invoke `osigfopen(NULL, 0)` (see [`osigfopen`], page 21).

setmsheader

`int setmsheader(char *`*`record`*`, char *`*`snarray`*`[], unsigned int `*`nsegments`*`)`

Return:

`0`	Success
`-1`	Failure: illegal record name, or no segments specified, or header not writable
`-2`	Failure: segment name too long, or insufficient memory
`-3`	Failure: attempt to nest multi-segment records, or unreadable segment header
`-4`	Failure: segment length unspecified, or numbers of signals or sampling frequencies don't match between segments

This function creates a header file (in the current directory) for a multi-segment *record* (see Section 5.5 [Multi-Segment Records], page 74. *snarray* contains the names of the segments, each of which must be an existing (single-segment) record; *nsegments* specifies the number of segments in *snarray*. Once a header has been created by `setmsheader`, any WFDB application can read the concatenated signal files of the constituent segment simply by opening the multi-segment record (using `isigopen` or `wfdbinit`). Note that the signal files themselves are not modified in any way, nor are they copied; rather, the other WFDB library functions that read signals (`getvec`, `getframe`, `isigsettime`, and `isgsettime`) automatically switch among the signal files of the segments as required. For an example of the use of `setmsheader`, see `app/wfdbcollate.c` in the WFDB Software Package. (This function was first introduced in WFDB library version 9.1.)

To close the header file, along with output signal files, and check that they were written successfully, invoke `osigfopen(NULL, 0)` (see [`osigfopen`], page 21).

getseginfo

`int getseginfo(WFDB_Segment **`*`psegarray`*`)`

Return:

`(int)`	number of segments belonging to the current input record

Invoking `getseginfo(`*`psegarray`*`)`, where *psegarray* is declared to be of type `**WFDB_Seginfo`, sets **psegarray* so that it points to an array of `WFDB_Seginfo` structures that describe the segments of the currently open (multi-segment) record. The return value

indicates the number of segments (i.e., the number of valid `WFDB_Seginfo` structures in **psegarray*. If there is no current input record, or if the current input record is not a multi-segment record, this function returns 0 and does not modify **psegarray*.

contain the segments' names, lengths, and starting sample numbers.

wfdbquit

```
void wfdbquit(void)
```

This function closes all open WFDB files and frees any memory allocated by other WFDB library functions. It also resets the following:

- the WFDB path (in versions 10.5.7 and later)
- 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 [`timstr` and `strtim`], page 36,
- the parameters used for converting between adus and physical units (reset to `WFDB_DEFGAIN` adu/mV, a quantity defined in `<wfdb/wfdb.h>`); see [`aduphys` and `physadu`], page 38,
- internal variables used to determine output signal specifications; see [`newheader`], page 42.

If any annotations have been written out-of-order (see Section 5.10 [Annotation Order], page 78), 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.

Programs that do not write annotations or signals need not use `wfdbquit`. Note, however, that several WFDB library functions allocate memory that is maintained for later use by the library. This is not generally a problem, since these functions also free such memory if it is no longer needed on a subsequent call; thus these 'memory leaks' do not grow over time. Virtually all operating systems reclaim memory allocated by user-level applications on exit, so that a small and self-limiting leak is not a problem. Nevertheless, there are embedded systems and other environments in which memory is not reclaimed when a user application exits, and in these cases it is best to invoke `wfdbquit()` on exit from any WFDB application, even those that do not write output using the library. In an ANSI/ISO C environment, this can be ensured by including the line

```
atexit(wfdbquit);
```

early in the code, before the first exit.

iannclose

```
void iannclose(WFDB_Annotator an)
```

This function closes the annotation file associated with input annotator *an*. It was first introduced in WFDB library version 9.1.

oannclose

```
void oannclose(WFDB_Annotator an)
```

This function closes the annotation file associated with output annotator *an*. It was first introduced in WFDB library version 9.1.

If any annotations have been written out-of-order (see Section 5.10 [Annotation Order], page 78), 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.

wfdbquiet

```
void wfdbquiet(void)
```

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

wfdbverbose

```
void wfdbverbose(void)
```

This function can be used to restore normal error reporting after using `wfdbquiet`. (This function was first introduced in WFDB library version 4.0.)

wfdberror

```
char *wfdberror(void)
```

Return:

`(char *)` pointer to error string

This function returns a pointer to a string containing the text of the most recent WFDB library error message (or to a string containing the WFDB library version number, if there have been no errors). Function `wfdberror` is primarily intended for use in applications for which the standard error output is unavailable or inadequate, such as in X Window System applications. (Note that this function may be unnecessary for MS-Windows applications, since the MS-Windows version of the WFDB library generates a message box for error messages, unless `wfdbquiet` has been used to silence them.) This function was first introduced in WFDB library version 4.5. Versions earlier than 9.4 return an empty string rather than the library version number if there have been no errors.

wfdbmemerr

```
void wfdbmemerr(int exit_on_error)
```

This function sets how the WFDB library behaves in the event of a memory allocation error. If *exit_on_error* is true (any non-zero value), then such an event causes the WFDB library to emit an appropriate error message and then terminate the running program. (This behavior is the default.)

If *exit_on_error* is false (zero), a subsequent memory allocation error will cause the WFDB library function in which it occurs to continue running if possible (after emitting an error message as above).

This function was first introduced in WFDB library version 10.4.6.

sampfreq

```
WFDB_Frequency sampfreq(char *record)
```

Return:

`(WFDB_Frequency)>0.`
: Success: the returned value is the sampling frequency in Hz

`(WFDB_Frequency)-1.`
: Failure: unable to read header file

`(WFDB_Frequency)-2.`
: Failure: incorrect header file format

This function determines the sampling frequency (in Hz) for the record specified by its argument. If its argument is NULL, `sampfreq` returns the currently defined sampling frequency, if any. It also sets the internal variables used by the time-conversion functions (see [`timstr` and `strtim`], page 36) for converting between sample intervals and seconds. See [Example 3], page 84, for an illustration of the use of `sampfreq`. Note that the value returned by `sampfreq` for a multifrequency record depends on the current `getvec` mode (see [`setgvmode`], page 24).

setsampfreq

```
int setsampfreq(WFDB_Frequency freq)
```

Return:

`0`
: Success

`-1`
: Failure: illegal frame frequency specified (*freq* must not be negative)

This function sets the frame frequency used by the time-conversion functions (see [`timstr` and `strtim`], page 36). Use `setsampfreq` before creating a new `hea` file (see [`newheader`], page 42).

Note that despite the name of this function, the argument specifies the record *frame* frequency, not the *sampling* frequency; the two are not equivalent when reading or writing a multi-frequency record (see Section 5.4 [Multi-Frequency Records], page 73).

See [Example 8], page 92, for an illustration of the use of `setsampfreq`.

setbasetime

```
int setbasetime(char *string)
```

Return:

`0`
: Success

`-1`
: Failure: incorrect string format

This function sets the base time used by the time-conversion functions `timstr` and `strtim`. Its argument is a null-terminated ASCII string in *HH:MM:SS* format. An optional base date in *dd/mm/yyyy* format can follow the time in *string*; if present, the date should be separated from the time by a space or tab character. If *string* is empty or `NULL`, the current date and time are read from the system clock. Use `setbasetime` after defining the sampling frequency and before creating a header file. (In versions 10.5.8 and later, it is not

necessary to define the sampling frequency first.) (see [newheader], page 42). See [Example 8], page 92, for an illustration of the use of `setbasetime`.

There is no `getbasetime` function; use `mstimstr(0)` at any time after opening a record to convert the base date and time into a string.

findsig

```
int findsig(const char *string)
```

Return:

`WFDB_Signal`
: Success

`-1`
: Failure: signal not found

This function converts its argument to an input signal number. If *string* is numeric and can be interpreted as a valid input signal number, it is taken as such; otherwise, it is assumed to be a signal name, and if it is an exact match to the `desc` field of a currently open input signal's `siginfo` structure, `findsig` returns the corresponding signal number. If two or more signals have identical matching names, `findsig` returns the lowest matching signal number.

Database records are sometimes obtained from analog tapes for which a tape counter is available. Since many analog tape recorders lack elapsed time indicators, it is often useful to identify events in the analog tape using counter values. A similar situation may arise if a chart recording or other hard copy with numbered pages is to be compared with a database record. To simplify cross-referencing between the analog tape or chart and the digital database record, the WFDB library supports conversion of counter values (or page numbers) to time. For this to be possible, the counter must be linear (i.e., it must change at the same rate throughout the tape; this is not true of those that count the number of revolutions of the supply or take-up reel), and the base counter value (the counter value or page number corresponding to sample 0) and the counter frequency (the difference between counter values separated by a one-second interval, or the reciprocal of the number of seconds per page) must be defined. The following four functions, first introduced in WFDB library version 5.2, are used to obtain or set the values of these parameters.

getcfreq

```
WFDB_Frequency getcfreq(void)
```

Return:

`(WFDB_Frequency)`
: the counter frequency in Hz

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

setcfreq

```
void setcfreq(WFDB_Frequency freq)
```

This function sets the counter frequency. Use `setcfreq` before creating a `hea` file (see [`newheader`], page 42). The effect of `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.

getbasecount

```
double getbasecount(void)
```

Return:

`(double)` base counter value

This function returns the base counter value, which is set by the functions that read header files, or by `setbasecount`. If the base counter value has not been set explicitly, `getbasecount` returns zero.

setbasecount

```
void setbasecount(double count)
```

This function sets the base counter value. Use `setbasecount` before creating a header file (see [`newheader`], page 42). The effect of `setbasecount` is nullified by later invoking any of the functions that read `hea` files.

setwfdb

```
void setwfdb(const char *string)
```

This function may be used to set or change the database path (see Section 1.4 [WFDB path], page 12) within a running program. The argument points to a null-terminated string that specifies the desired database path (but see the next paragraph for an exception). The string contains a list of locations where input files may be found. These locations may be absolute directory names (such as '`/usr/local/database`' under Unix, or '`d:/database`' under MS-DOS), relative directory names (e.g., `../mydata`), or URL prefixes (e.g., '`http://physionet.org/physiobank/database`'). If NETFILES support is unavailable, any URL prefixes in the string are ignored. The special form '`.`' refers to the current directory. Entries in the list may be separated by whitespace or by semicolons; under Unix, colons may also be used as separators. An empty component, indicated by an initial or terminal separator, or by two consecutive separators, will be understood to specify the current directory (which may also be indicated by a component consisting of a single '`.`'). If the string is empty or `NULL`, the database path is limited to the current directory.

If *string* begins with '`@`', the remaining characters of *string* are taken as the name of a file from which the WFDB path is to be read. This file may contain either the WFDB path, as described in the previous paragraph, or another indirect WFDB path specification. Indirect WFDB path specifications may be nested no more than ten levels deep (an arbitrary limit imposed to avoid infinite recursion). Evaluation of indirect WFDB paths is deferred until `getwfdb` is invoked, either explicitly or by the WFDB library while attempting to open an input file (e.g., using `annopen` or `isigopen`). (The features described in this paragraph were first introduced in WFDB library version 8.0.) See [`getwfdb`], page 49, for an example of the use of `setwfdb`.

getwfdb

```
char *getwfdb(void)
```

Return:

`(char *)` pointer to the database path string

This function returns the current database path. For example, this code fragment

```
...
char *oldp, *newp;

oldp = getwfdb();
if (newp = malloc(strlen("/usr/mydb;") + strlen(oldp) + 1)) {
    sprintf(newp, "/usr/mydb;%s", oldp);
    setwfdb(newp);
}
...
```

adds the directory '`/usr/mydb`' to the beginning of the database path. (The standard '`/`' directory separator can be used, even under MS-DOS; if you elect to use the alternate '\', remember to quote it within a C string as '\\'.)

resetwfdb

```
void resetwfdb(void)
```

This function restores the WFDB path to its initial value (either the first value returned by `getwfdb` in the current process, or `NULL`).

wfdbfile

```
char *wfdbfile(const char *type, char *record)
```

Return:

`(char *)` pointer to a filename, or `NULL`

This function attempts to locate an existing WFDB file by searching the database path (see Section 1.4 [WFDB path], page 12). Normally, the file is specified by its *type* (e.g., `hea`, or an annotator name such as `atr`) and by the *record* to which it belongs. A file that does not include a record name as part of its name can be found by `wfdbfile` if the name is passed in the *type* variable and *record* is `NULL`. The string returned by `wfdbfile` 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. If the WFDB library has been compiled with NETFILES support, and the WFDB path includes one or more URL prefixes, the string returned may be a URL rather than a pathname. If the file cannot be found, `wfdbfile` returns `NULL`. (This function was first introduced in WFDB library version 4.3.)

wfdbflush

```
void wfdbflush(void)
```

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.

getinfo

```
char *getinfo(char *record)
```

Return:

`(char *)` pointer to an "info" string, or `NULL`

If *record* is not `NULL`, `getinfo` reads the first "info" string for *record*. If *record* is `NULL`, then `getinfo` reads the next available info string for the currently open record. Info strings are null-terminated and do not contain newline characters. Some records may contain no info strings; others may contain more than one info string. They may be stored within `hea` or `info` files; if any are contained in a record's `hea` file, they are returned first.

For example, the following code fragment may be used to read and print all of the info for record `100s`:

```
...
char *info;

if (info = getinfo("100s"))
    do {
        puts(info);
    } while (info = getinfo(NULL));
...
```

If the `hea` file was opened by another WFDB function, such as `isigopen`, `annopen`, `sampfreq`, or `wfdbinit`, the *record* argument can be `NULL` even for the first `getinfo`, like this:

```
...
char *info;
WFDB_Frequency sps;

if (sps = sampfreq("100s")
    while (info = getinfo(NULL))
        puts(info);
...
```

(This function was first introduced in WFDB library version 4.0.)

putinfo

```
int putinfo(const char *s)
```

Return:

`0`	Success
`-1`	Failure: header not initialized

This function writes *s* as an "info" string associated with the current output record. If `setinfo` has been called more recently than `newheader` or `setheader`, the info string is written to the `info` file; otherwise, it is written to the `hea` file, unless none of these three functions has been called (which results in an error). The string argument, *s*, must be null-terminated and should not contain newline characters. No more than 254 characters may be written in a single invocation of `putinfo`. Two or more info strings may be written

to the same header or info file by successive invocations of `putinfo`. (This function was first introduced in WFDB library version 4.0.)

setinfo

```
int setinfo(char *record)
```

Return:

`0`	Success
`-1`	Failure: illegal record name, or info file could not be opened
`-2`	Failure: error writing info file (only if *record* is `NULL`)

This function opens the `info` file for the specified *record* for writing using `putinfo`. If *record*`.info` does not exist in the current directory, it is created; otherwise, it is opened for appending (i.e., so that anything `putinfo` writes is added at the end of the file rather than overwriting any existing contents).

The file opened by `setinfo` can be used to store arbitrary information associated with *record*, without altering *record*'s header file. (`setinfo` was first introduced in WFDB library version 10.5.11.)

To close the `info` file and check that it was written successfully, invoke `setinfo(NULL)`. (Prior to WFDB library version 10.7.0, this would always return 0.)

wfdb_freeinfo

```
void wfdb_freeinfo()
```

This function releases memory allocated by `getinfo`, and closes the file opened by `setinfo`, if any. After calling it, `getinfo` behaves as it does on its initial call, (re)reading the info for the specified record (or the currently open record, if no record is specified). `wfdb_freeinfo` is invoked by `wfdbquit`. (This function was first introduced in WFDB library version 10.5.11.)

setibsize

```
int setibsize(int size)
```

Return:

`>0`	Success: the returned value is the new input buffer size in bytes
`-1`	Failure: buffer size could not be changed
`-2`	Failure: illegal value for *size*

This function can be used to change the default size of the input buffers allocated by `getvec`. It cannot be used while input signals are open (i.e., after invoking `isigopen` or `wfdbinit` and before invoking `wfdbquit`). If *size* is positive, the default input buffers will be *size* bytes; if *size* is zero, the system default buffer size (`BUFSIZ`) is used. Note that the default buffer size has no effect on reading signals for which an explicit buffer size is given in the header file, i.e., those for which the `bsize` field of the `WFDB_Siginfo` structure (see Section 3.1 [Signal Information Structures], page 58) is non-zero. (This function was first introduced in WFDB library version 5.0.)

setobsize

```
int setobsize(int size)
```

Return:

`>0`	Success: the returned value is the new output buffer size in bytes
`-1`	Failure: buffer size could not be changed
`-2`	Failure: illegal value for *size*

This function can be used to change the default size of the output buffers allocated by `putvec`. It cannot be used while output signals are open (i.e., after invoking `osigopen` or `osigfopen` and before invoking `wfdbquit`). If *size* is positive, the default output buffers will be *size* bytes; if *size* is zero, the system default buffer size (`BUFSIZ`) is used. Note that the default buffer size has no effect on writing signals for which an explicit buffer size is given in the `hea` file read by `osigopen`, or in the `bsize` field of the `WFDB_Siginfo` structure (see Section 3.1 [Signal Information Structures], page 58) passed to `osigfopen`. (This function was first introduced in WFDB library version 5.0.)

wfdbgetskew

```
int wfdbgetskew(WFDB_Signal s)
```

Return:

`(int)`	the skew (in frames) for input signal *s*

This function returns 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 *s* is not a valid input signal number. Since sample vectors returned by `getvec` or `getframe` are already corrected for skew, `wfdbgetskew` is useful primarily for programs that need to rewrite existing `hea` files, where it is necessary to preserve the previously recorded skews. The following code fragment demonstrates how this can be done:

```
char *record;
int nsig;
WFDB_Signal s;
static WFDB_Siginfo *si;

...

if ((nsig = isigopen(record, NULL, 0)) < 1)
    exit(1);
si = (WFDB_Siginfo *)malloc(nsig * sizeof(WFDB_Siginfo));
if (si == NULL || isigopen(record, siarray, nsig)!= nsig)
    exit(1);
for (s = 0; s < nsig; s++) {
    wfdbsetskew(s, wfdbgetskew(s));
    wfdbsetstart(s, wfdbgetstart(s));
}
setheader(record, siarray, (unsigned)nsig);
```

Note that this function does not *determine* the skew between signals; the problem of doing so is not possible to solve in the general case. `wfdbgetskew` merely reports what has previously been determined by other means and recorded in the header file for the input record. (This function was first introduced in WFDB library version 9.4.)

wfdbsetskew

```
void wfdbsetskew(WFDB_Signal s, int skew)
```

This function sets the specified *skew* (in frames) to be recorded by `newheader` or `setheader` for signal *s*. For an example of the use of `wfdbsetskew`, see [wfdbgetskew], page 52. Note that `wfdbsetskew` has no effect on the skew correction performed by `getframe` (or `getvec`), which is determined solely by the skews that were recorded in the header file at the time the input signals were opened. (This function was first introduced in WFDB library version 9.4.)

wfdbgetstart

```
long wfdbgetstart(WFDB_Signal s)
```

Return:

`(long)` the length of the prolog of the file that contains input signal *s*

This function returns 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 `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 `isigopen` and `isigsettime` take the prolog into account when calculating byte offsets for `getframe` and `getvec`, `wfdbgetstart` is useful primarily for programs that need to rewrite existing `hea` files, where it is necessary to preserve the previously recorded byte offsets. For an example of how this can be done, see [wfdbgetskew], page 52. (This function was first introduced in WFDB library version 9.4.)

wfdbsetstart

```
void wfdbsetstart(WFDB_Signal s, long bytes)
```

This function sets the specified prolog length (*bytes*) to be recorded by `newheader` or `setheader` for signal *s*. For an example of the use of `wfdbsetstart`, see [wfdbgetskew], page 52. Note that `wfdbsetstart` has no effect on the calculations of byte offsets within signal files as performed by `isigsettime`, which are determined solely by the contents of the `hea` file at the time the signals were opened. (This function was first introduced in WFDB library version 9.4.)

wfdbputprolog

```
void wfdbputprolog(const char *prolog, long bytes,
                   WFDB_Signal s)
```

This function writes the specified *prolog* of length *bytes* to the signal file for the specified output signal *s*, and invokes `wfdbsetstart(`*s*, *bytes*`)`. (This function was first introduced in WFDB library version 10.4.15.)

2.9 memory allocation macros

These macros use the standard ANSI/ISO C functions `calloc()`, `realloc()`, `free()`, `strlen()`, and `strcpy()` to handle dynamic memory allocation tasks for the WFDB library. They can also be used by applications that include `wfdb/wfdb.h`, where they are defined.

These macros provide safe handling of insufficient memory and double free errors (either condition results in a descriptive error message, which by default is followed by an `exit(1)` to end the process with a signal to the parent shell or other process).

MEMERR

`MEMERR(`*object_name*`, size_t` *n_elements*`, size_t` *element_size*`)`

This macro uses `wfdb_error` to send a short error message of the form `WFDB: can't allocate (`*n_elements*`*`*element_size* `) bytes for` *object_name*`)`. Unless `wfdbmemerr(1)` has been invoked previously, the process that invoked `MEMERR` exits immediately.

SFREE

`SFREE(`*object* `*`*pointer*`)`

This macro releases memory previously allocated to the *object* addressed by the specified *pointer*, somewhat more safely than by invoking the standard `free()` function. On completion, *pointer* is set to NULL.

`SFREE` does nothing if *pointer* is initially NULL (unlike `free()`, which may cause the process to crash). If `SFREE` receives a non-NULL pointer, it passes that pointer to `free()`, which may cause a crash if the pointer does not point to a block of memory that was previously allocated using one of the macros below, or directly using `malloc()`, `calloc()`, or `realloc()`.

SUALLOC

`SUALLOC(`*object_name*`, size_t` *n_elements*`, size_t` *element_size*`)`

This macro allocates memory sufficient for *n_elements* items of *element_size* bytes each, and sets the pointer given by *object_name* to point to the allocated memory. If there is not enough available memory, `SUALLOC` invokes `MEMERR` (above).

The newly allocated memory block is filled with zeroes.

`SUALLOC` does not check to see if *object_name* already points to allocated memory, which will lead to memory leaks if so.

SALLOC

`SALLOC(`*object_name*`, size_t` *n_elements*`, size_t` *element_size*`)`

This macro allocates memory sufficient for *n_elements* items of *element_size* bytes each, and sets the pointer given by *object_name* to point to the allocated memory. If there is not enough available memory, `SALLOC` invokes `MEMERR` (above).

The newly allocated memory block is filled with zeroes.

Unless *object_name* is initially NULL, `SALLOC` frees it using `SFREE` before allocating the requested memory.

SREALLOC

SREALLOC(*object_name*, size_t *n_elements*, size_t *element_size*)

This macro allocates memory sufficient for *n_elements* items of *element_size* bytes each, and sets the pointer given by *object_name* to point to the allocated memory. If there is not enough available memory, SREALLOC invokes MEMERR (above).

Use SREALLOC to expand a previously allocated block of memory, preserving its contents. SREALLOC usually allocates a new block of the desired size, moving the contents of the previously allocated block into the beginning of the new block and then freeing the original block. Pointers to locations in the original block will no longer be valid in this case.

The portion of the newly allocated block that extends beyond the previous contents is uninitialized.

If *object_name* is initially NULL, SUALLOC, SALLOC, and SREALLOC are functionally equivalent, except that SREALLOC does not fill the allocated block with zeroes.

SSTRCPY

SSTRCPY(char *destination, char *source)

This macro copies the *source* string (including a trailing null character) into newly-allocated memory, and it sets *destination* to point to the copy. If *destination* is not NULL on entry, SSTRCPY uses SFREE to release the previously allocated memory.

3 Data Types

Simple data types used by the WFDB library are defined in `<wfdb/wfdb.h>`. These include:

`WFDB_Sample`
: a signed integer type (at least 32 bits) used to represent sample values, in units of adus.

`WFDB_Time`
: a signed integer type (at least 32 bits) used to represent times and time intervals, in units of sample intervals. Only the magnitude is significant; the sign of a `WFDB_Time` variable indicates how it is to be printed by `timstr` or `mstimstr`.

  The definition of `WFDB_Time` depends on whether the `WFDB_LARGETIME` macro is defined (see Section 3.9 [Large time values], page 66).

`WFDB_Date`
: a signed integer type (at least 32 bits) used to represent Julian dates, in units of days.

`WFDB_Frequency`
: a floating point type used to represent sampling and counter frequencies, in units of Hz.

`WFDB_Gain`
: a floating point type used to represent signal gains, in units of adus per physical unit.

`WFDB_Group`
: an unsigned integer type used to represent signal group numbers.

`WFDB_Signal`
: an unsigned integer type used to represent signal numbers.

`WFDB_Annotator`
: an unsigned integer type used to represent annotator numbers.

Composite data types used by the WFDB library are also defined in `<wfdb/wfdb.h>`. These types, described in detail in the following sections, include:

`WFDB_Siginfo`
: an object containing the name and global attributes of a given signal.

`WFDB_Calinfo`
: an object containing calibration specifications for signals of a given type.

`WFDB_Anninfo`
: an object containing the name and attributes of a given annotator.

`WFDB_Annotation`
: an object describing one or more attributes of one or more signals at a given time.

3.1 Signal Information Structures

The *siarray* argument for `isigopen`, `osigopen`, `wfdbinit`, and `osigfopen` is a pointer to an array of objects of type `WFDB_Siginfo`. The first three of these functions fill in the `WFDB_Siginfo` objects to which *siarray* points, but the caller must supply initialized `WFDB_Siginfo` objects to `osigfopen`. Each object specifies the attributes of a signal:

`char *fname`
: a pointer to a null-terminated string that names the file in which samples of the associated signal are stored. Input signal files are found by prefixing `fname` with each of the components of the database path in turn (see Section 1.4 [WFDB path], page 12). `fname` may include relative or absolute path specifications if necessary; the use of an absolute pathname, combined with an initial null component in `WFDB`, reduces the time needed to find the signal file to a minimum. If `fname` is '`-`', it refers to the standard input or output.

`char *desc`
: a pointer to a null-terminated string without embedded newlines (e.g., '`ECG lead V1`' or '`trans-thoracic impedance`'). The length of the `desc` string is restricted to a maximum of `WFDB_MAXDSL` (defined in `<wfdb/wfdb.h>`) characters, not including the null.

`char *units`
: a pointer to a null-terminated string without embedded whitespace. The string specifies the physical units of the signal; if NULL, the units are assumed to be millivolts. The length of the `units` string is restricted to a maximum of `WFDB_MAXUSL` (defined in `<wfdb/wfdb.h>`) characters (not including the null).

`WFDB_Gain gain`
: the number of analog-to-digital converter units (adus) per physical unit (see previous item) relative to the original analog signal; for an ECG, this is roughly equal to the amplitude of a normal QRS complex. If `gain` is zero, no amplitude calibration is available; in this case, a `gain` of `WFDB_DEFGAIN` (defined in `<wfdb/wfdb.h>`) may be assumed.

`WFDB_Sample initval`
: the initial value of the associated signal (i.e., the value of sample number 0).

`WFDB_Group group`
: the signal group number. All signals in a given group are stored in the same file. If there are two or more signals in a group, the file is called a *multiplexed signal file*. Group numbers begin at 0; arrays of `WFDB_Siginfo` structures are always kept ordered with respect to the group number, so that signals belonging to the same group are described by consecutive entries in *siarray*.

`int fmt`
: the signal storage format. The most commonly-used formats are format 8 (8-bit first differences), format 16 (16-bit amplitudes), and format 212 (pairs of 12-bit amplitudes bit-packed into byte triplets). See `<wfdb/wfdb.h>` for a complete list of supported formats. All signals belonging to the same group must be stored in the same format.

`int spf`
: the number of samples per frame. This is 1, for all except oversampled signals in multi-frequency records, for which `spf` may be any positive integer. Note

that non-integer values are not permitted (thus the frame rate must be chosen such that all sampling frequencies used in the record are integer multiples of the frame rate).

`int bsize` the block size, in bytes. For signal files that reside on Unix character device special files (or their equivalents), the `bsize` field indicates how many bytes must be read or written at a time (see Section 5.7 [Special Files], page 76). For ordinary disk files, `bsize` is zero. All signals belonging to a given group have the same `bsize`.

`int adcres`
: the ADC resolution in bits. Typical ADCs have resolutions between 8 and 16 bits inclusive.

`int adczero`
: the ADC output given an input that falls exactly at the center of the ADC range (normally 0 VDC). Bipolar ADCs produce two's complement output; for these, `adczero` is usually zero. For the MIT DB, however, an offset binary ADC was used, and `adczero` was 1024.

`int baseline`
: the value of ADC output that would map to 0 physical units input. The value of `adczero` is not synonymous with that of `baseline` (the isoelectric or physical zero level of the signal); the `baseline` is a characteristic of the *signal*, while `adczero` is a characteristic of the *digitizer*. The value of `baseline` need not necessarily lie within the output range of the ADC; for example, if the `units` are '`degrees_Kelvin`', and the ADC range is 200–300 degrees Kelvin, `baseline` corresponds to absolute zero, and lies well outside the range of values actually produced by the ADC.

`long nsamp`
: the number of samples in the signal. (Exception: in multi-frequency records, `nsamp` is the number of samples divided by `spf`, see above, i.e., the number of frames.) All signals in a given record must have the same `nsamp`. If `nsamp` is zero, the number of samples is unspecified, and the `cksum` (see the next item) is not used; this is useful for specifying signals that are obtained from pipes, for which the length may not be known.

`int cksum` a 16-bit checksum of all samples. This field is not usually accessed by application programs; `newheader` records checksums calculated by `putvec` when it creates a new `hea` file, and `getvec` compares checksums that it calculates against `cksum` at the end of the record, provided that the entire record was read through without skipping samples.

The number of `WFDB_Siginfo` structures in *siarray* is given by the *nsig* argument of the functions that open signal files. Input and output signal numbers are assigned beginning with 0 in the order in which the signals are given in *siarray*. Note that input signal 0 and output signal 0 are distinct. Input signal numbers are supplied to `aduphys`, `physadu`, `adumuv`, and `muvadu` in their first arguments. See [Example 5], page 86, for an illustration of how to read signal specifications from `WFDB_Siginfo` structures.

3.2 Calibration Information Structures

The *cal* argument for `getcal` and `putcal` is a pointer to an object of type `WFDB_Calinfo`. A `WFDB_Calinfo` object contains information about signals of a specified type:

`char *sigtype`
: a pointer to a null-terminated string without embedded tabs or newlines. This field describes the type(s) of signals to which the calibration specifications apply. Usually, `sigtype` is an exact match to (or a prefix of) the `desc` field of the `WFDB_Siginfo` object that describes a matching signal.

`char *units`
: a pointer to a null-terminated string without embedded whitespace. This field specifies the physical units of signals to which the calibration specifications apply. Usually, the `units` field of a `WFDB_Calinfo` structure must exactly match the `units` field of the `WFDB_Siginfo` structure that describes a matching signal.

`double scale`
: the customary plotting scale, in physical units per centimeter. WFDB applications that produce graphical output may use `scale` as a default. Except in unusual circumstances, signals of different types should be plotted at equal multiples of their respective `scale`s.

`double low`
`double high`
: values (in physical units) corresponding to the low and high levels of a calibration pulse. If the signal is AC-coupled (see below), `low` is zero, and `high` is the pulse amplitude.

`int caltype`
: a small integer that specifies the shape of the calibration pulse (see `<wfdb/wfdb.h>` for definitions). `caltype` is even if signals of the corresponding `sigtype` are AC-coupled, and odd if they are DC-coupled.

The calibration list is a memory-resident linked list of `WFDB_Calinfo` structures. It is accessible only via `calopen`, `getcal`, `putcal`, `newcal`, and `flushcal`.

3.3 Annotator Information Structures

The *aiarray* argument for `annopen` and `wfdbinit` is a pointer to an array of objects of type `WFDB_Anninfo`. Each member of the array contains information provided to `annopen` and `wfdbinit` about an annotation file associated with the record:

`char *name`
: the annotator name. The name `atr` is reserved for a *reference annotation file* supplied by the creator of the database record to document its contents as accurately and thoroughly as possible. You may use other annotator names to identify annotation files that you create; unless there are compelling reasons not to do so, follow the convention that the annotator name is the name of the file's creator (a program or a person). To avoid confusion, do not use '`dat`', '`datan`', '`dn`', or '`hea`' (all of which are commonly used as parts of WFDB file names) as annotator names. The special name '`-`' refers to the standard input or output.

Other annotator names may contain upper- or lower-case letters, digits, and underscores. Annotation files are normally created in the current directory and found in any of the directories in the database path (see Section 1.4 [WFDB path], page 12).

`int stat` the file type/access code. Usually, `stat` is either `WFDB_READ` or `WFDB_WRITE`, to specify standard ("WFDB format") annotation files to be read by `getann` or to be written by `putann`. Both MIT DB and AHA DB annotation files can be (and generally are) stored in WFDB format. The symbols `WFDB_READ` and `WFDB_WRITE` are defined in `<wfdb/wfdb.h>`. An AHA-format annotation file can be read by `getann` or written by `putann` if the `stat` field is set to `WFDB_AHA_READ` or `WFDB_AHA_WRITE` before calling `annopen` or `wfdbinit` (see [Example 2], page 82). Other formats may be supported via a similar mechanism; consult `<wfdb/wfdb.h>` for more information.

The number of `WFDB_Anninfo` objects in *aiarray* is given by the *nann* argument of `annopen` and `wfdbinit`. The annotation-reading function, `getann`, knows the annotators by number only; `annopen` and `wfdbinit` assign input annotator numbers beginning with 0 in the order in which they are given in the array of `WFDB_Anninfo` objects. Output annotator numbers used by `putann` also start at 0; note that input annotator 0 and output annotator 0 are distinct. Annotator numbers are supplied to `getann` and `putann` in their first arguments. See [`annopen`], page 18, for an example of how to set the contents of an array of `WFDB_Anninfo` objects.

3.4 Annotation Structures

The *annot* argument of `getann` and `putann` is an object of type `WFDB_Annotation` containing these fields:

`long time` time of the annotation, in samples from the beginning of the record. The times of beat annotations in the `atr` files for the MIT DB generally coincide with the R-wave peak in signal 0; for the AHA DB, they generally coincide with the PQ-junction.

`char anntyp`
annotation code; an integer between 1 and `ACMAX`. See Chapter 4 [Annotation Codes], page 67, for a list of legal annotation codes. `ACMAX` is defined in `<wfdb/ecgcodes.h>`.

`signed char subtyp`
`unsigned char chan`
`signed char num`
numbers between −128 and 127. In MIT DB `atr` files, the `subtyp` field is used with noise and artifact annotations to indicate which signals are affected (see Chapter 4 [Annotation Codes], page 67). The `chan` field is intended to indicate the signal to which the annotation is attached. More than one annotation may be written with the same `time` if the `num` or `chan` fields are distinct and in ascending order. The semantics of the `chan` field are unspecified, however; users may assign any desired meaning, which need not have anything to do with signal numbers. In user-created annotation files, these fields can be used to

store arbitrary small integers. The `subtyp` field requires no space in a standard annotation file unless it is non-zero; the `chan` and `num` fields require no space unless they have changed since the previous annotation.

`char *aux` a free text string. The first byte is interpreted as an `unsigned char` that specifies the number of bytes that follow (up to 255). In MIT DB `atr` files, the `aux` field is used with rhythm change annotations to specify the new rhythm, and with comment annotations to store the text of the comment (see Chapter 4 [Annotation Codes], page 67). The string can contain arbitrary binary data, including embedded nulls. It is unwise to store anything but ASCII strings, however, if the annotation file may be transported to a system with a different architecture (e.g., on which multiple-byte quantities may have different sizes or byte layouts). The `aux` field requires no space in a standard annotation file if it is `NULL`. Note that conversion of annotation files to other formats may entail truncation or loss of the `aux` string. Note also that the `aux` pointer returned by `getann` points to a small static buffer (separately allocated for each input annotator beginning with WFDB library version 9.4) that may be overwritten by subsequent calls.

See [Example 3], page 84, for a short program that examines the contents of a `WFDB_Annotation`.

3.5 Segment Information Structures

Objects of type `WFDB_Seginfo` contain these fields:

`char recname[WFDB_MAXRNL+1]`
Segment name (the name of the simple record corresponding to a segment of a multi-segment record), unless `recname` has the special value '`~`'. In the latter case, the segment is a gap (i.e., it corresponds to an interval during which no signals are available).

`WFDB_Time nsamp`
Segment length in samples.

`WFDB_Time samp0`
Number of samples that precede the segment in the multi-segment record to which it belongs. If the segment is opened as an individual record, its `n`th sample has sample number `n-1`, just as for any record. If the record to which the segment belongs is opened, the `n`th sample in the segment has sample number `n-1+samp0`.

3.6 Limits of Numeric Types

It is sometimes useful to refer to the minimum or maximum possible value for a particular data type. Although the WFDB library has certain minimum requirements for its numeric types, the actual range of *possible* values depends on your C compiler, and the CPU and operating system where your program is running.

Note that the range of values for a particular *signal* is usually smaller than the possible range of values that can be stored in a `WFDB_Sample` variable. If you want to know the

maximum and minimum values for a particular signal, refer to the `adcres` and `adczero` fields of the of the `WFDB_Siginfo` structure (see Section 3.1 [Signal Information Structures], page 58).

The following macros (defined in `<wfdb/wfdb.h>`) can be used to determine the limits of *integer* types. In order to use these macros, your program must also include the statement `#include <limits.h>`.

`WFDB_SAMPLE_MIN`
: Smallest value that can be stored as a `WFDB_Sample`.

`WFDB_SAMPLE_MAX`
: Largest value that can be stored as a `WFDB_Sample`.

`WFDB_DATE_MIN`
: Smallest value that can be stored as a `WFDB_Date`.

`WFDB_DATE_MAX`
: Largest value that can be stored as a `WFDB_Date`.

`WFDB_TIME_MIN`
: Smallest value that can be stored as a `WFDB_Time`.

`WFDB_TIME_MAX`
: Largest value that can be stored as a `WFDB_Time`.

`WFDB_GROUP_MAX`
: Largest value that can be stored as a `WFDB_Group`.

`WFDB_SIGNAL_MAX`
: Largest value that can be stored as a `WFDB_Signal`.

`WFDB_ANNOTATOR_MAX`
: Largest value that can be stored as a `WFDB_Annotator`.

The following macros can be used to determine the limits and precision of *floating-point* types. In order to use these macros, your program must also include the statement `#include <float.h>`.

`WFDB_FREQUENCY_MAX`
: Largest finite value that can be represented as a `WFDB_Frequency`.

`WFDB_FREQUENCY_DIG`
: Number of decimal digits of precision of a `WFDB_Frequency`.

`WFDB_FREQUENCY_MAX_10_EXP`
: Largest finite power of 10 that can be represented as a `WFDB_Frequency`.

`WFDB_FREQUENCY_EPSILON`
: The difference between 1.0 and the smallest value greater than 1.0 that can be represented as a `WFDB_Frequency`.

`WFDB_GAIN_MAX`
: Largest finite value that can be represented as a `WFDB_Gain`.

`WFDB_GAIN_DIG`
: Number of decimal digits of precision of a `WFDB_Gain`.

`WFDB_GAIN_MAX_10_EXP`
: Largest finite power of 10 that can be represented as a `WFDB_Gain`.

`WFDB_GAIN_EPSILON`
: The difference between 1.0 and the smallest value greater than 1.0 that can be represented as a `WFDB_Gain`.

3.7 Displaying Numeric Values

To display numeric values on the screen, or convert them to strings, it is often convenient to use the standard `printf`, `fprintf`, or `sprintf` functions. Each of these functions requires you to specify the data type as part of the "format" string (for example, to display an `int`, you might write `printf("%d", x)`, but to display a `long int`, you might write `printf("%ld", x)`).

The macros listed below can be used to display `WFDB_Sample`, `WFDB_Time`, `WFDB_Frequency`, and `WFDB_Gain` values, regardless of which standard C data types these represent. Using these macros can help to ensure your program is portable to other operating systems and C compilers.

Each macro expands to a string constant (such as `"d"` or `"ld"`), which does not include the leading '`%`' character. For example, to display a table of time and sample values, we might write:

```
WFDB_Time t = 0;
WFDB_Sample v[2];
while (getvec(v) > 0) {
    printf("%10"WFDB_Pd_TIME" %6"WFDB_Pd_SAMP" %6"WFDB_Pd_SAMP"\n",
           t, v[0], v[1]);
    t++;
}
```

The following macros are defined in `<wfdb/wfdb.h>`:

Macro	Argument type	Format	Example output
`WFDB_Pd_SAMP`	`WFDB_Sample`	Base 10	'995'
`WFDB_Pi_SAMP`	`WFDB_Sample`	Base 10	'995'
`WFDB_Po_SAMP`	`WFDB_Sample`	Unsigned base 8	'1743'
`WFDB_Pu_SAMP`	`WFDB_Sample`	Unsigned base 10	'995'
`WFDB_Px_SAMP`	`WFDB_Sample`	Unsigned base 16	'3e3'
`WFDB_PX_SAMP`	`WFDB_Sample`	Unsigned base 16	'3E3'
`WFDB_Pd_TIME`	`WFDB_Time`	Base 10	'650000'
`WFDB_Pi_TIME`	`WFDB_Time`	Base 10	'650000'
`WFDB_Po_TIME`	`WFDB_Time`	Unsigned base 8	'2365420'
`WFDB_Pu_TIME`	`WFDB_Time`	Unsigned base 10	'650000'
`WFDB_Px_TIME`	`WFDB_Time`	Unsigned base 16	'9eb10'
`WFDB_PX_TIME`	`WFDB_Time`	Unsigned base 16	'9EB10'
`WFDB_Pe_FREQ`	`WFDB_Frequency`	Exponential	'3.600000e+02'
`WFDB_PE_FREQ`	`WFDB_Frequency`	Exponential	'3.600000E+02'
`WFDB_Pf_FREQ`	`WFDB_Frequency`	Fixed-point	'360.000000'
`WFDB_Pg_FREQ`	`WFDB_Frequency`	Automatic	'360'

WFDB_PG_FREQ	WFDB_Frequency	Automatic	'360'
WFDB_Pe_GAIN	WFDB_Gain	Exponential	'2.000000e+02'
WFDB_PE_GAIN	WFDB_Gain	Exponential	'2.000000E+02'
WFDB_Pf_GAIN	WFDB_Gain	Fixed-point	'200.000000'
WFDB_Pg_GAIN	WFDB_Gain	Automatic	'200'
WFDB_PG_GAIN	WFDB_Gain	Automatic	'200'

(The 'd' and 'i' formats are equivalent, and are provided for symmetry with `scanf`. For more information, see the documentation of your C compiler.)

3.8 Parsing Numeric Values

To convert text input to a numeric value, it is often convenient to use the standard `scanf`, `fscanf`, or `sscanf` functions. Each of these functions requires you to specify the data type as part of the "format" string (for example, to read an integer and store it as an `int`, you might write `scanf("%d", &x)`, but to store it as a `long int`, you might write `scanf("%ld", &x)`).

As with the `printf`-style macros described in the previous section, the macros listed below can be used to convert a string to a `WFDB_Sample`, `WFDB_Time`, `WFDB_Frequency`, or `WFDB_Gain` value, regardless of which standard C data types these represent. Each macro expands to a string constant (such as `"d"` or `"ld"`), which does not include the leading '%' character. For example, to read a table of time and sample values, we might write:

```
WFDB_Time t;
WFDB_Sample v[2];
char buf[1000];
while (fgets(buf, sizeof(buf), stdin)) {
    if (sscanf(buf, "%"WFDB_Sd_TIME"%"WFDB_Sd_SAMP"%"WFDB_Sd_SAMP",
               &t, &v[0], &v[1]) == 3) {
        putvec(v);
    }
}
```

The following macros are defined in `<wfdb/wfdb.h>`:

Macro	Argument type	Format
WFDB_Sd_SAMP	WFDB_Sample *	Base 10
WFDB_Si_SAMP	WFDB_Sample *	Base 8, 10, or 16
WFDB_So_SAMP	WFDB_Sample *	Unsigned base 8
WFDB_Su_SAMP	WFDB_Sample *	Unsigned base 10
WFDB_Sx_SAMP	WFDB_Sample *	Unsigned base 16
WFDB_SX_SAMP	WFDB_Sample *	Unsigned base 16
WFDB_Sd_TIME	WFDB_Time *	Base 10
WFDB_Si_TIME	WFDB_Time *	Base 8, 10, or 16
WFDB_So_TIME	WFDB_Time *	Unsigned base 8
WFDB_Su_TIME	WFDB_Time *	Unsigned base 10
WFDB_Sx_TIME	WFDB_Time *	Unsigned base 16
WFDB_SX_TIME	WFDB_Time *	Unsigned base 16
WFDB_Se_FREQ	WFDB_Frequency *	Decimal

WFDB_SE_FREQ	WFDB_Frequency *	Decimal
WFDB_Sf_FREQ	WFDB_Frequency *	Decimal
WFDB_Sg_FREQ	WFDB_Frequency *	Decimal
WFDB_SG_FREQ	WFDB_Frequency *	Decimal
WFDB_Se_GAIN	WFDB_Gain *	Decimal
WFDB_SE_GAIN	WFDB_Gain *	Decimal
WFDB_Sf_GAIN	WFDB_Gain *	Decimal
WFDB_Sg_GAIN	WFDB_Gain *	Decimal
WFDB_SG_GAIN	WFDB_Gain *	Decimal

(The 'x' and 'X' formats are equivalent, as are the 'e', 'E', 'f', 'g', and 'G' formats, and are provided for symmetry with printf. For more information, see the documentation of your C compiler.)

3.9 Large Time Values

The WFDB_Time type is defined as a signed integer type of at least 32 bits, which means that it can represent sample numbers up to 2,147,483,647. By default, WFDB_Time is defined as an alias for the standard C long int, and many existing applications have been written with the assumption that WFDB_Time and long int are interchangeable.

However, it is quite possible for a record to be longer than 2,147,483,647 samples (about 25 days of recording at 1 kHz) and it is useful to be able to process such records on machines where a long int is only 32 bits.

If you are using a modern C compiler, with WFDB library version 10.7.0 or later, it is possible to define WFDB_Time as long long int instead of long int. This lets your program work with sample numbers as large as 9,223,372,036,854,775,807, even on a 32-bit machine. To do this, add the following line at the very beginning of your source file, *before* including <wfdb/wfdb.h>:

```
#define WFDB_LARGETIME
```

If your program consists of multiple *.c files, be sure to do the same for each file. Alternatively, you can define this macro on the C compiler command line (e.g., -DWFDB_LARGETIME if you are using gcc).

When doing this, you will also need to ensure that your program handles large time values consistently, by using the WFDB_Time data type rather than long or long long, using WFDB_TIME_MAX rather than LONG_MAX or LLONG_MAX, using WFDB_Pd_TIME rather than "ld" or "lld", and so forth.

4 Annotation Codes

Application programs that deal with annotations should include the line

```
#include <wfdb/ecgcodes.h>
```

which provides the symbolic definitions of annotation codes given in the first column of the table below. (The second column of the table shows the strings returned by `annstr` and `ecgstr`.)

Beat annotation codes:

NORMAL	N	Normal beat
LBBB	L	Left bundle branch block beat
RBBB	R	Right bundle branch block beat
BBB	B	Bundle branch block beat (unspecified)
APC	A	Atrial premature beat
ABERR	a	Aberrated atrial premature beat
NPC	J	Nodal (junctional) premature beat
SVPB	S	Supraventricular premature or ectopic beat (atrial or nodal)
PVC	V	Premature ventricular contraction
RONT	r	R-on-T premature ventricular contraction
FUSION	F	Fusion of ventricular and normal beat
AESC	e	Atrial escape beat
NESC	j	Nodal (junctional) escape beat
SVESC	n	Supraventricular escape beat (atrial or nodal) [1]
VESC	E	Ventricular escape beat
PACE	/	Paced beat
PFUS	f	Fusion of paced and normal beat
UNKNOWN	Q	Unclassifiable beat
LEARN	?	Beat not classified during learning

Non-beat annotation codes:

VFON	[	Start of ventricular flutter/fibrillation
FLWAV	!	Ventricular flutter wave
VFOFF	]	End of ventricular flutter/fibrillation
NAPC	x	Non-conducted P-wave (blocked APC) [4]
WFON	(	Waveform onset [4]
WFOFF	)	Waveform end [4]
PWAVE	p	Peak of P-wave [4]
TWAVE	t	Peak of T-wave [4]
UWAVE	u	Peak of U-wave [4]
PQ	‘	PQ junction
JPT	’	J-point
PACESP	^	(Non-captured) pacemaker artifact
ARFCT	|	Isolated QRS-like artifact [2]
NOISE	~	Change in signal quality [2]
RHYTHM	+	Rhythm change [3]
STCH	s	ST segment change [1,3]
TCH	T	T-wave change [1,3,4]

SYSTOLE	*	Systole [1]
DIASTOLE	D	Diastole [1]
MEASURE	=	Measurement annotation [1,3]
NOTE	"	Comment annotation [3]
LINK	@	Link to external data [5]

Notes:

1. Codes SVESC, STCH, and TCH were first introduced in WFDB library version 4.0. Codes SYSTOLE, DIASTOLE, and MEASURE were first introduced in WFDB library version 7.0.
2. In MIT and ESC DB atr files, each non-zero bit in the subtyp field indicates that the corresponding signal contains noise (the least significant bit corresponds to signal 0).
3. The aux field contains an ASCII string (with prefixed byte count) describing the rhythm, ST segment, T-wave change, measurement, or the nature of the comment. By convention, the character that follows the byte count in the aux field of a RHYTHM annotation is '('. See the *MIT-BIH Arrhythmia Database Directory* for a list of rhythm annotation strings.
4. Codes WFON, WFOFF, PWAVE, TWAVE, and UWAVE were first introduced in DB library version 8.3. The 'p' mnemonic now assigned to PWAVE was formerly assigned to NAPC, and the 't' mnemonic now assigned to TWAVE was formerly assigned to TCH. The obsolete codes PQ (designating the PQ junction) and JPT (designating the J-point) are still defined in <wfdb/ecgcodes.h>, but are identical to WFON and WFOFF respectively.
5. The LINK code was first introduced in WFDB library version 9.6. The aux field of a LINK annotation contains a URL (a uniform resource locator, in the form http://machine.name/some/data, suitable for passing to a Web browser such as Netscape or Mosaic). LINK annotations may be used to associate extended text, images, or other data with an annotation file. If the aux field contains any whitespace, text following the first whitespace is taken as descriptive text to be displayed by a WFDB browser such as WAVE.

The annotation codes in the table above are the predefined values of the anntyp field in a WFDB_Annotation. Other values in the range of 1 to ACMAX (defined in <wfdb/ecgcodes.h>) are legal but do not have preassigned meanings. The constant NOTQRS, also defined in <wfdb/ecgcodes.h>, is not a legal value for anntyp, but is a possible output of the macros discussed below.

4.1 Macros for Mapping Annotation Codes

Application programs that use the macros described in this section should include the line

```
#include <wfdb/ecgmap.h>
```

which will make their definitions, and those in <wfdb/ecgcodes.h>, available.

`isann(c)` true (1) if *c* is a legal annotation code, false (0) otherwise

`isqrs(c)` true (1) if *c* denotes a QRS complex, false (0) otherwise

`map1(c)` maps *c* into one of the set {NOTQRS, NORMAL, PVC, FUSION, LEARN}

`map2(c)` maps *c* into one of the set {NOTQRS, NORMAL, SVPB, PVC, FUSION, LEARN}

`annpos(c)`
: maps c into one of the set {`APUNDEF`, `APSTD`, `APHIGH`, `APLOW`, `APATT`, `APAHIGH`, `APALOW`} (see `<wfdb/ecgmap.h>` for definitions of these symbols; this macro was first introduced in WFDB library version 6.0)

If you define your own annotation codes, you may wish to modify the tables used by the macros above. The file `<wfdb/ecgmap.h>` also defines `setisqrs(`*c*`, `*x*`)`, `setmap1(`*c*`, `*x*`)`, `setmap2(`*c*`, `*x*`)`, and `setannpos(`*c*`, `*x*`)` for this purpose. In each case, x is the value to be returned when the corresponding mapping macro is invoked with an argument of c. (These macros were first introduced in WFDB library version 6.0.)

The macros below convert between AHA and MIT annotation codes; they are also defined in `<wfdb/ecgmap.h>`.

`ammap(a)`
: maps a (an AHA annotation code) into an MIT annotation code (one of the set {`NORMAL`, `PVC`, `FUSION`, `RONT`, `VESC`, `PACE`, `UNKNOWN`, `VFON`, `VFOFF`, `NOISE`, `NOTE`}), or `NOTQRS`

`mamap(c, s)`
: maps c (an MIT annotation code) into an AHA annotation code (one of the set {'`N`', '`V`', '`F`', '`R`', '`E`', '`P`', '`Q`', '`[`', '`]`', '`U`', '`O`'}); s is the MIT annotation `subtyp` (significant only if c is `NOISE`)

5 Database Files

The WFDB library has been constructed to provide a standard interface between the database files and application programs. Alternate means of access to database files is strongly discouraged, since file formats may change. Database files are located in the directories specified by `WFDB` (see Section 1.4 [WFDB path], page 12).

Recall that a WFDB record is not a file; rather, it is an extensible *collection* of database files (see [Records], page 1). Thus, for example, record 100 of the MIT-BIH Arrhythmia Database consists of the files named `100.hea`, `100.dat`, and `100.atr` in the `mitdb` directory of the MIT-BIH Arrhythmia Database CDROM (or in PhysioBank, within `http://physionet.org/physiobank/database/mitdb/`), together with any additional files in other directories that you may have associated with record 100 (such as your own annotation file). All files associated with a given record include the record name as the first part of the file name. No explicit action (other than choosing the file name, and locating the file in the WFDB path) is needed in order to associate a new file with an existing WFDB record.

To find the location of a database file easily, you can use `wfdbwhich`, an application included with the WFDB Software Package. Type `wfdbwhich` for brief instructions on its use, or see the *WFDB Applications Guide*.

5.1 File Types

There are four types of files supported by the WFDB library:

Header Files

Header files have names of the form *`record`*`.hea`, where *record* is the record name. (MIT DB records are named 100–124 and 200–234 with some numbers missing. AHA DB records are named 1001–1010, 2001–2010, 3001–3010, 4001–4010, 5001–5010, 6001–6010, 7001–7010, and 8001–8010. ESC DB records are named e0103–e1304, with many numbers missing.) Header files are text files, with lines terminated by ASCII carriage-return/line-feed pairs, created by `newheader`, `setheader`, or `setmsheader`, from which `isigopen`, `osigopen`, and `wfdbinit` read the names of the signal files and their attributes as given in the array of `WFDB_Siginfo` objects; `sampfreq` also reads a header file to determine the sampling frequency used for a record.

Signal Files

Signal files usually have names of the form *`record`*`.dat`. (The `.dat` suffix is conventional, but not required; any file name acceptable to the operating system is permissible.) Signal files are binary, and usually contain either 16-bit amplitudes (format 16), pairs of 12-bit amplitudes bit-packed into byte triplets (format 212), or 8-bit first differences (format 8). (See `<wfdb/wfdb.h>` for information about other formats that are supported.) The functions that read and write signal files perform appropriate transformations so that the samples visible to the application program are always amplitudes of type `int` (at least 16 bits), regardless of the signal file format.

Annotation Files

Annotation files have names of the form *record*`.`*annotator*. Those named *record*`.atr` are reference annotation files (assumed to be correct). Annotation files are binary, and contain records of variable length that average slightly over 16 bits per annotation.

Calibration Files

Unlike header, signal, and annotation files, *calibration files* are not associated with individual records. A calibration file is needed only if you have records containing signals other than ECGs; in this case, it is likely that a single calibration file will be adequate for use with all of your records. Calibration files are text files, with lines terminated by ASCII carriage-return/line-feed pairs, created by `newcal`, from which `calopen` reads the calibration list (see Section 3.2 [Calibration Information Structures], page 60). The WFDB Software Package includes a standard calibration file, `wfdbcal`, in the `data` directory.

EDF Files

European Data Format (EDF) was defined in 1990, and it has become a very widely supported open format for exchange of recorded physiologic signals, especially polysomnograms. EDF files encapsulate functional equivalents of header and signal files, and EDF+ files can also include annotation streams (stored as signals).

EDF files begin with an embedded (text) header containing specifications of the signals and a limited amount of demographic information, followed by the binary samples of the signals. Within each block of samples, typically one second to one minute in length, all samples of the first signal are stored consecutively, followed by all samples of the second signal, etc.

EDF files can be read directly using WFDB library version 10.4.5 and later. The name of the file, which must include a '.' and cannot end in '.hea', can be passed as a record name to any WFDB library function that accepts record names. Although the WFDB library does not support EDF as an output format, the `mit2edf` application included in the WFDB Software Package can convert anything readable by the WFDB library into EDF.

EDF+, defined in 2003, is backwards-compatible with EDF (any EDF reader, including the WFDB library, can read EDF+), but the additional features of EDF+, including methods for reading annotations and recognizing signal discontinuities (which are marked by annotations), are available only from EDF+-specific readers. The WFDB library does not currently include built-in support for the additional features of EDF+, but annotation streams are available as signals, so it is possible for an application using the WFDB library to provide its own means of decoding annotation streams as they are read.

Further information about EDF and EDF+ is available at `http://www.edfplus.info/`.

AHA Format Files

The "AHA Format" was defined in 1980 for storage of database records on 9-track digital tape. Signal files in AHA format are in format 16, with two signals multiplexed into one file (see Section 5.3 [Multiplexed Signal Files], page 73), and may be read and written using `getvec` and `putvec`. AHA-format annotation files are binary, and contain fixed-length (16-byte) annotation records. An annotation file in AHA format may be read or written using `getann` or `putann`, if the `stat` field of the `WFDB_Anninfo` object is set to `WFDB_AHA_READ` or

`WFDB_AHA_WRITE` before opening the file. `annopen` recognizes the format of input annotation files automatically and prints a warning if the format does not match what was expected on the basis of `stat`. AHA format annotation files may be converted to standard format without loss of information, and doing so reduces the storage requirement by a factor of eight.

Yet another format has been used more recently for distribution of AHA DB files on floppy diskettes and CDROMs. This format is compatible with neither the original AHA format nor with any of the formats supported directly by the WFDB library. Programs `a2m` and `ad2m`, supplied with the WFDB Software Package, can convert files in this format (as well as those in the original AHA format) to the standard formats.

5.2 Using Standard I/O for Database Files

If '`-`' is supplied as a record name to any of the functions that read or write header files, the `hea` file is taken to be the standard input or output, as appropriate. If the name of a signal file is specified in the `hea` file (or in the array of `WFDB_Siginfo` objects passed to `osigfopen`) as '`-`', the standard input (output) is used by `getvec` (`putvec`). If the name of an annotator is given in the array of `WFDB_Anninfo` objects as '`-`', the standard input (output) is used by `getann` (`putann`). If the name of a calibration file is given as '`-`', the standard input (output) is used by `calopen` (`newcal`).

Under MS-DOS, these features may not always be usable, since the standard input and output are usually opened in "text" mode (which is unsuitable for binary database files).

Although the WFDB library does not forbid the use of the standard input or output for more than one function (e.g., as both a signal file and an annotation file), such use is in general a gross error that is likely to lead to unintended results.

5.3 Multiplexed Signal Files

Multiplexed signal files may be identified by examining the `group` fields of the array of `WFDB_Siginfo` objects returned by `isigopen` or `wfdbinit`. Signals belonging to the same group are multiplexed together in the same file. If all signals in a given signal file have been sampled at the same frequency, and there are n signals in the file, then each group of n successive samples in that file contains a sample from each signal, always in the same order (but see Section 5.4 [Multi-Frequency Records], page 73).

Multiplexed signal files can be useful if the storage device is sequential-access only (e.g., 9-track tape), if the storage device has lengthy seek times (e.g., optical disk), if many signals must be recorded and Unix's per-process limit on open files would otherwise be exceeded, or if very high speed is required while the file is being created (because of sampling constraints). CDROM signal files, and those available from PhysioNet, are multiplexed unless the record contains only one signal.

5.4 Multi-Frequency Records

When signals of different types are recorded simultaneously and for lengthy periods, it may be appropriate to choose different sampling frequencies in order to reduce the storage requirements for signals of limited bandwidth. The support for multi-frequency records provided in WFDB library version 9.0 (and later versions) allows application programs to

read and write records containing signals digitized at multiple sampling frequencies. In a multi-frequency record, a *frame* of samples contains one *or more* samples from each signal. The *frame rate* (base sampling frequency) of the record, as recorded in the header file and as normally returned by `sampfreq`, is defined as the number of frames per second. Signals sampled at multiples of the frame rate are referred to as *oversampled signals*. For each signal, a frequency multiplier specifies how many samples are included in each frame. The frequency multiplier (1 by default) is an integer, encoded within the format field in the header file, and specified in the `spf` field of the `WFDB_Siginfo` structure for the signal.

A frame can be read as it was written (see [`getframe`], page 28) by an application that has been written to make use of multi-frequency records. Applications that are not "multi-frequency aware" can still read signals using the standard `getvec` interface, which returns (as always) one sample per signal on each invocation. By default, `getvec` reads multi-frequency records in *low-resolution* mode. In this mode, each oversampled signal is resampled at the frame rate by averaging all of its samples in each frame.

The function `setgvmode` can be used to select *high-resolution* mode, in which `getvec` replicates samples of signals digitized at less than the maximum sampling frequency (i.e., using zero-order interpolation) so that each sample of an oversampled signal appears in at least one sample vector returned by `getvec`. In this mode, `sampfreq` returns the number of samples per signal returned by `getvec` per second of the record. Furthermore (when using WFDB library version 9.6 and later versions), all time quantities passed to and from the WFDB library functions are understood to be in units of these shorter sampling intervals; thus, for example, `getann` converts times in frame numbers (as recorded in annotation files) into times in sample numbers before filling in the caller's annotation structure, and `putann` converts times in sample numbers into times in frame numbers before writing annotations into annotation files. This permits applications that are not "multi-frequency aware" to read multi-frequency records with the highest possible resolution.

The operating mode used by `getvec`, if not specified by an explicit call to `setgvmode`, is determined by the value of the environment variable `WFDBGVMODE` if it is set, and otherwise by the value of `DEFWFDBGVMODE` in `wfdblib.h` at the time the library was compiled.. In either case, a value of 0 selects low-resolution mode, and any other value selects high-resolution mode.

5.5 Multi-Segment Records

A multi-segment record consists of two or more concatenated segments. Each segment is an ordinary WFDB record, with its own header file and signal file(s). There are two types of multi-segment records: *fixed-layout* records, in which all signals must appear in the same order within each segment (signals may not be omitted, added, or swapped), and the gain and baseline of any given signal may not change from segment to segment; and *variable-layout* records, which are not bound by these constraints. If the first segment of a multi-segment record has a length of zero samples, that segment is a *layout segment*, and the record is a variable-layout record. Version 9.1 of the WFDB library is the first to support reading and writing fixed-layout multi-segment records, and version 10.3.17 is the first to support reading variable-layout multi-segment records.

In both types of multi-segment records, the sampling frequency of any given signal must be the same in each segment. Segments of multi-segment records must be ordinary

records (it is not permitted to nest one multi-segment record within another, for example), and the length of each segment must be specified (the WFDB library does not impose this requirement on ordinary records that are not part of a multi-segment record). There are no other restrictions on segments; specifically, it is permitted to mix segments with different storage formats, and for any segment to appear more than once. A special header file (created either manually or by using `setmsheader`) specifies the record name for each segment in a multi-segment record. Once this special header exists, the multi-segment record can be read by any WFDB application. Note that only the signal files of the segments are "linked" by the multi-segment record's header; annotation files associated with the individual segments are *not* readable as part of the multi-segment record (although an annotation file associated directly with the multi-segment record can be created and read just as for an ordinary record). From the point of view of a WFDB application, reading a multi-segment record is exactly like reading an ordinary record; specifically, `isigsettime` works as expected, permitting jumps forward and backward between as well as within segments.

Unlike ordinary segments, no signal file is associated with a layout segment; only the header file is needed. In the header file of a layout segment, all of the signals present in any of the other segments of the record are listed, in the desired order and with the desired gains and baselines. When the WFDB library reads the record, `getframe` assembles the frame, scaling and shifting each component as needed. If any signals are missing during a segment, the resulting gaps are filled with the sample value `WFDB_INVALID_SAMPLE`. In this way, WFDB applications do not need to be aware of signal changes; rather, they may read variable-layout records as if they were ordinary (fixed-layout) records.

WFDB applications generally assume fixed-layout records, i.e., that the number and types of available signals (and their sampling frequencies, gains, and baselines) are constant throughout the record. These conditions do not always apply in clinical settings, in which signals may be added, removed, or recalibrated to meet clinical needs, resulting in variable-layout recordings. A variable-layout recording may be divided into segments such that each segment is a fixed-layout WFDB record. The segments can then be reassembled into a multi-segment WFDB record. Version 10.3.17, and later versions, of the WFDB library contain code in `signal.c` to create a virtual fixed-layout record on the fly when reading a variable-layout record. This feature can also be used to transform ordinary records on the fly, if it is desirable to rearrange, delete, duplicate, scale, or shift signals.

5.6 Simultaneous Access to Multiple Records

Selection functions that accept *record* arguments (`annopen`, `isigopen`, `osigopen`, and `wfdbinit`) normally close any active database files of the types with which each deals before proceeding. The argument +*record* is synonymous with *record*, but has the effect of causing these functions to leave any active files open. (For convenience, the other functions that accept *record* arguments — `sampfreq`, `newheader`, and `setheader` — also treat *record* and +*record* as synonymous, but without any noticeable effect.) The restrictions on the total numbers of signals and annotation files still apply.

If the sampling frequencies or lengths of the records do not match, a warning message will be produced (unless `wfdbquiet` was invoked). The time-conversion functions (see [`timstr` and `strtim`], page 36) will continue to use the sampling frequency and base time

defined for the first record that was opened, unless these attributes are reset by `sampfreq`, `setsampfreq`, or `setbasetime`.

Function `calopen` uses the '`+`' convention for calibration file names. Although it normally creates the calibration list from scratch each time it is called, it retains the current calibration list if the calibration file name is prefixed by '`+`'.

5.7 Signals That Are Not Stored in Disk Files

The `fname` component of a `WFDB_Siginfo` object can be any string acceptable as a file name to your operating system. Under Unix, for example, signals can be read from (or written to) 'special' files such as `/dev/rmt0` (the raw tape drive). If I/O must be performed in fixed-size blocks (such as for Unix character devices), the `bsize` component of the `WFDB_Siginfo` object must contain the appropriate block size in bytes. In such cases, the WFDB library must obtain (using `malloc` (see *K&R*, page 167) an amount of memory equal to the size of one block when the signal file is first opened. For large programs running on 16-bit machines, this can cause problems if signal files with large block sizes are read. (In such cases, `isigopen` or `osigopen` will not open the signal file if there is not enough memory to allocate a buffer.) Under Unix, if this problem occurs, use the "piped records" (see Section 5.8 [Piped and Local Records], page 76) instead. The usual method is to read or write the signal file using a utility such as Unix's `dd` and to pipe the data to or from the application program. Although this approach is flexible, there are a few drawbacks:

1. While reading piped input, the standard input cannot be used for other purposes by the application program. Interactive programs can avoid problems by opening `/dev/tty` for I/O, however.
2. Programs that use `isigsettime` or `isgsettime` cannot perform backward skips on piped input, and forward skips can be quite slow.
3. Additional system resources (computation time, process slots, and memory) are needed when using pipes, in comparison with the usual method of operation.

Several special-purpose header files allow application programs to read data directly from 9-track tape. When the WFDB Software Package is installed, these files are copied into the `tape` subdirectory of the system-wide database directory. The record names associated with most of these header files (`tape/512`, `tape/1024`, `tape/4096`, `tape/10240`) specify the block size in bytes. These use 16-bit format, 250 Hz samples, 12 bit ADC with zero ADC offset, two signals multiplexed into one, and data to be read from `/dev/rmt0`. Record `tape/6144d` uses 8-bit difference format, 6144 bytes/block, and is otherwise similar to the others. Records `tape/ahatape` and `tape/mittape` can be used to read or write an AHA-format signal file on a 9-track tape that has been positioned to the beginning of the correct file; the signal file for these is `/dev/nrmt0` (the non-rewinding raw tape drive). If the tape density is encoded into the tape drive name on your system, additional header files may be needed.

5.8 Piped and Local Records

Piped record header files allow application programs to read signals from the standard input, or write them to the standard output. Record `8` specifies 8-bit format, a 10-bit ADC, zero ADC offset, and two signals sampled at 250 Hz, both of which are to be acquired from the

standard input, or written to the standard output. Record `16` specifies 16-bit format and a 12-bit ADC, and is otherwise identical to record `8`. ADCs from several manufacturers can produce output in the format specified by record `16`; thus such output can be piped directly into an application program using record `16`. Signal files in AHA format also match these specifications. Piped records for reading or writing other numbers of signals are provided in the `pipe` subdirectory of the system-wide database directory; they are named `pipe/8x`*n* and `pipe/16x`*n*, where *n* is the number of signals ($n = 1, 2, \ldots, 16$; piped record header files can be created with larger numbers of signals (use the existing files as a model).

Application programs may also read or write signal files in the current directory using *local record* header files. Record `16l` ("one-six-ell") specifies up to sixteen format 16 files, and record `8l` ("eight-ell") specifies up to sixteen format 8 files, named `data0`, `data1`, `data2`, ..., `data`*n* in the current directory. When opened using `isigopen` or `wfdbinit`, these signal files will be readable by `getvec` as signals 0, 1, 2, ... 16 respectively. These files should be created by the user, with the use of `putvec`. It is necessary to create only as many signal files as will be used; if, for example, only one signal is needed, only `data0` need be created.

5.9 NETFILES

If the symbol `WFDB_NETFILES` is defined at the time the WFDB library is compiled, then input files located on remote web (HTTP) and FTP servers can be read directly. This capability is implemented using the `libcurl` library (which is available on many of the platforms supported by the WFDB library). NETFILES support, if available, is transparent to WFDB applications. To make use of this feature, simply link to the NETFILES-enabled WFDB library (the necessary `libcurl` functions will be loaded automatically), and incorporate one or more URL prefixes in the WFDB path.

In current versions of the WFDB library, the default WFDB path (defined in the WFDB library source file `wfdblib.h`, and used as the WFDB path if the WFDB environment variable is undefined) is '`. /usr/database http://physionet.org/physiobank/database`'. (The second component, after the '`.`' that specifies the current directory, may vary, depending on your platform and the choices made during installation.) The third component is a URL prefix pointing to PhysioBank, an on-line archive for a wide variety of standard databases of physiologic signals. For example, the MIT-BIH Polysomnographic Database is kept in `http://physionet.org/physiobank/database/slpdb`, so it is possible to read record `slp37` of that database directly from PhysioBank by passing `slpdb/slp37` as the *record* argument to `wfdbinit` (or `isigopen`, `annopen`, etc.).

The current implementation of `libcurl` permits input from `http://` URLs in much the same way that local files are read, provided that the remote web server supports HTTP 1.1 range requests (most, including PhysioNet's, do). This means that it is not necessary to download an entire file in order to examine part of it, and you may notice little or no speed difference between local file and network file input for many applications. If the remote server does not support range requests, however, or if input is from an `ftp://` URL, the current implementations download the entire file to memory, so you may notice a significant startup delay if the file is long and your network connection is slow, or if the file does not fit into physical memory.

Currently, NETFILES support is limited to input files; as always, any output files created by the WFDB library are written into the current directory, unless the record name contains local path information.

NETFILES support was introduced in WFDB library version 10.0.1.

5.10 Annotation Order

WFDB applications may generally assume (and most of them do assume) that all annotations in any given annotation file are in *canonical order*. Successful use of `iannsettime` requires that this assumption be correct. The earliest versions of the WFDB library (before version 6.1) defined canonical order as time order. Versions 6.1 through 10.4.11 define canonical order as `time` and `chan` order, and versions 10.4.12 and later also use `num` for ordering (thus annotations are arranged first in `time` order, and any simultaneous annotations are arranged according to the value of their `num` fields, from smallest to largest, and those with identical `num` fields are similarly arranged in `chan` order).

The combination of the `time`, `num`, and `chan` fields of an annotation defines a unique *location* in a virtual array of annotations which an annotation file represents. No two annotations may occupy the same location in this virtual array. This restriction was enforced by versions of the WFDB library earlier than version 9.7. In these versions of the WFDB library, `putann` required that annotations be written in canonical order, and refused to write any out-of-order annotations supplied to it.

Current versions of the WFDB library do not impose this requirement. In version 9.7 and later versions, `putann` accepts and records out-of-order annotations and multiple annotations that occupy the same location. If any such annotations have been written, the completed annotation file is rewritten in canonical order by `wfdbquit` or `oannclose`. This is accomplished by running `sortann` (see the *WFDB Applications Guide*) as a separate process using the ANSI C `system` function. If this function is not available, or if `sortann` cannot be run, `wfdbquit` (or `oannclose`) emits a warning message describing how to post-process the annotations to put them into canonical order.

Although it is possible using current versions of the WFDB library to write two or more annotations to the same location, *only the last annotation written to any given location is retained* in the canonically-ordered annotation file. Thus that an application that generates an annotation file can change the `anntyp`, `subtyp`, or `aux` fields of a previously-written annotation simply by writing another annotation to the same location (i.e, with the same `time`, `num`, and `chan` fields). As a special case, an application may *delete* a previously-written annotation by writing a `NOTQRS` annotation to the same location. To move an annotation to a different location (i.e., to change its `time`, `num`, or `chan` fields), it is necessary to delete it from the original location, and then to insert it at the desired location, using two separate invocations of `putann`.

In unusual circumstances, an unsorted annotation file may be useful (for example, as an aid for debugging the application that produced it; `rdann` can be used to list all of the annotations in such a file, in the order in which they were written). In some environments, the use of the ANSI C `system` function may be a security problem, and you may wish to avoid automatic sorting of annotations for this reason. Set the environment variable `WFDBANNSORT` to 0 at run time, or define the symbol `DEFWFDBANNSORT` as 0 when compiling

the WFDB library, if you wish to suppress automatic annotation sorting by `wfdbquit` and `oannclose`.

6 Programming Examples

The programs in this chapter are useful as models for a variety of applications that use the WFDB library. The line numbers are for reference only; they are not part of the programs. Any of these examples can be compiled (under Unix) using a command of the form

```
cc file.c -lwfdb
```

or, if the WFDB library or its `*.h` files are not in the standard locations:

```
cc `wfdb-config --cflags` file.c `wfdb-config --libs`
```

where *file.c* is the name of the file containing the source; see Chapter 1 [Using the WFDB Library], page 7, for further information. The sources for these examples are included in the WFDB Software Package, within the `examples` directory.

Example 1: An Annotation Filter

The following program copies an annotation file, changing all QRS annotations to `NORMAL` and deleting all non-QRS annotations.

```
 1  #include <stdio.h>
 2  #include <wfdb/wfdb.h>
 3  #include <wfdb/ecgmap.h>
 4
 5  main()
 6  {
 7      WFDB_Anninfo an[2];
 8      char record[8], iann[10], oann[10];
 9      WFDB_Annotation annot;
10
11      printf("Type record name: ");
12      fgets(record, 8, stdin); record[strlen(record)-1] = '\0';
13      printf("Type input annotator name: ");
14      fgets(iann, 10, stdin); iann[strlen(iann)-1] = '\0';
15      printf("Type output annotator name: ");
16      fgets(oann, 10, stdin); oann[strlen(oann)-1] = '\0';
17      an[0].name = iann; an[0].stat = WFDB_READ;
18      an[1].name = oann; an[1].stat = WFDB_WRITE;
19      if (annopen(record, an, 2) < 0) exit(1);
20      while (getann(0, &annot) == 0)
21          if (isqrs(annot.anntyp)) {
22              annot.anntyp = NORMAL;
23              if (putann(0, &annot) < 0) break;
24          }
25      wfdbquit();
26  }
```

(See `http://physionet.org/physiotools/wfdb/examples/example1.c` for a copy of this program.)

Notes:

Line 2: All programs that use the WFDB library must include `<wfdb/wfdb.h>`.

Line 3: The `#include` statement makes available not only the mapping macros, one of which will be used in line 21, but also the annotation code symbols in `<wfdb/ecgcodes.h>`, one of which will be needed in line 22.

Line 7: Since there will be two annotators (one each for input and output), the array of `WFDB_Anninfo` objects has two members.

Line 9: This structure will be filled in by `getann`, modified, and passed to `putann` for output.

Lines 11–16:
The record name and the annotator names are filled into the character arrays. The code in lines 12, 14, and 16 illustrates a C idiom for reading a string of limited length; the second statement in each of these lines replaces the trailing newline character (which `fgets` copies into the string) with a null. String arguments to WFDB library functions should not include newline characters.

Lines 17–18:
Pointers to the character arrays (strings) containing the annotator names are filled into the `name` fields of the array of `WFDB_Anninfo` objects. Note that the `name` fields are only pointers and do not contain storage for the strings themselves. If this is not clear to you, review the discussion of pointers and arrays in *K&R*, pp. 97–100. The input annotator is to be read, the output annotator is to be written. `WFDB_READ` and `WFDB_WRITE` are defined in `<wfdb/wfdb.h>`.

Line 19: Note that the first and second arguments of `annopen` are the names of the respective arrays; thus `annopen` receives pointers rather than values in its argument list.

Line 20: An annotation is read from annotator 0 into `annot`. The '`&`' is necessary since `getann` requires a pointer to the structure in order to be able to modify its contents. When `getann` returns a negative value, no more annotations remain to be read and the loop ends.

Line 21: The macro `isqrs` is defined in `<wfdb/ecgmap.h>`; `isqrs(x)` is true if x is an annotation code that denotes a QRS complex, false if x is not a QRS annotation code.

Line 22: `NORMAL` is defined in `<wfdb/ecgcodes.h>`.

Line 23: The call to `putann` now writes the modified annotation in the output annotator 0 file. As for `getann`, a pointer to `annot` must be passed using the '`&`' operator.

Line 25: All files are closed prior to exiting. This is mandatory since the program creates an output file with `putann`.

Example 2: An Annotation Translator

This program translates the `atr` annotations for the record named in its argument into an AHA-format annotation file with the annotator name `aha`.

```
1  #include <stdio.h>
2  #include <wfdb/wfdb.h>
3
```

```
 4  main(argc, argv)
 5  int argc;
 6  char *argv[];
 7  {
 8      WFDB_Anninfo an[2];
 9      WFDB_Annotation annot;
10
11      if (argc < 2) {
12          fprintf(stderr, "usage: %s record\n", argv[0]);
13          exit(1);
14      }
15      an[0].name = "atr"; an[0].stat = WFDB_READ;
16      an[1].name = "aha"; an[1].stat = WFDB_AHA_WRITE;
17      if (annopen(argv[1], an, 2) < 0) exit(2);
18      while (getann(0, &annot) == 0 && putann(0, &annot) == 0)
19          ;
20      wfdbquit();
21      exit(0);
22  }
```

(See `http://physionet.org/physiotools/wfdb/examples/example2.c` for a copy of this program.)

Notes:

Lines 4–6:
If this doesn't look familiar, see *K&R*, pp. 114–115.

Lines 11–14:
This is the standard idiom for producing those cryptic error messages for which Unix programs are notorious; `argv[0]` is the name by which the program was invoked.

Lines 15–16:
These lines set up the annotator information. Input annotator 0 is the `atr` annotation file, and output annotator 0 will be written in AHA format.

Line 17: If we can't read the input or write the output, quit with an error message from `annopen`.

Line 18: Here's where the work is done. The format translation is handled entirely by `getann` and `putann`. The loop ends normally when `getann` reaches the end of the input file, or prematurely if there is a read or write error.

Line 21: Since we have carefully defined non-zero exit codes for the various errors that this program might encounter, we also define this successful exit here. If this program is run as part of a Unix shell script, the exit codes are accessible to the shell, which can determine what to do next as a result. If this line were omitted (as in example 1), the exit code would be undefined.

Example 3: An Annotation Printer

This program prints annotations in readable form. Its first argument is an annotator name, and its second argument is a record name.

```
 1  #include <stdio.h>
 2  #include <wfdb/wfdb.h>
 3
 4  main(argc, argv)
 5  int argc;
 6  char *argv[];
 7  {
 8      WFDB_Anninfo a;
 9      WFDB_Annotation annot;
10
11      if (argc < 3) {
12          fprintf(stderr, "usage: %s annotator record\n", argv[0]);
13          exit(1);
14      }
15      a.name = argv[1]; a.stat = WFDB_READ;
16      (void)sampfreq(argv[2]);
17      if (annopen(argv[2], &a, 1) < 0) exit(2);
18      while (getann(0, &annot) == 0)
19          printf("%s (%"WFDB_Pd_TIME") %s %d %d %d %s\n",
20                 timstr(-(annot.time)),
21                 annot.time,
22                 annstr(annot.anntyp),
23                 annot.subtyp, annot.chan, annot.num,
24                 (annot.aux != NULL && *annot.aux > 0) ?
25                 (char *) annot.aux+1 : "");
26      exit(0);
27  }
```

(See `http://physionet.org/physiotools/wfdb/examples/example3.c` for a copy of this program.)

Notes:

Line 16: The invocation of `sampfreq` here sets the internal variables needed by `timstr` below.

Line 20: This line gives the annotation time as a time of day. If the base time is omitted in the header file, or if we used `timstr(annot.time)` instead, we would obtain the elapsed time from the beginning of the record.

Lines 24–25: This expression evaluates to an empty string unless the `aux` string is non-empty. It makes the assumption that `aux` is a printable ASCII string; the printable part follows the length byte.

Example 4: Generating an R-R Interval Histogram

This program reads an annotation file, determines the intervals between beat annotations (assumed to be the R-R intervals), and accumulates a histogram of them.

```
 1  #include <stdio.h>
 2  #include <wfdb/wfdb.h>
 3  #include <wfdb/ecgmap.h>
 4
 5  main(argc, argv)
 6  int argc;
 7  char *argv[];
 8  {
 9      int rr, *rrhist, rrmax;
10      WFDB_Time t;
11      WFDB_Anninfo a;
12      WFDB_Annotation annot;
13      void *calloc();
14
15      if (argc < 3) {
16          fprintf(stderr, "usage: %s annotator record\n", argv[0]);
17          exit(1);
18      }
19      a.name = argv[1]; a.stat = WFDB_READ;
20      if (annopen(argv[2], &a, 1) < 0) exit(2);
21      if ((rrmax = (int)(3*sampfreq(argv[2]))) <= 0) exit(3);
22      if ((rrhist = (int *)calloc(rrmax+1, sizeof(int))) == NULL) {
23          fprintf(stderr, "%s: insufficient memory\n", argv[0]);
24          exit(4);
25      }
26      while (getann(0, &annot) == 0 && !isqrs(annot.anntyp))
27          ;
28      t = annot.time;
29      while (getann(0, &annot) == 0)
30          if (isqrs(annot.anntyp)) {
31              if ((rr = annot.time - t) > rrmax) rr = rrmax;
32              rrhist[rr]++;
33              t = annot.time;
34          }
35      for (rr = 1; rr < rrmax; rr++)
36          printf("%4d %s\n", rrhist[rr], mstimstr((WFDB_Time)rr));
37      printf("%4d %s (or longer)\n", rrhist[rr], mstimstr((WFDB_Time)rr));
38      exit(0);
39  }
```

(See `http://physionet.org/physiotools/wfdb/examples/example4.c` for a copy of this program.)

Notes:

Lines 21–25:

Here we allocate storage for the histogram. The value returned by `sampfreq`, if positive, specifies the number of sample intervals per second; we will allocate 3 seconds' worth of bins, initialized to zero. See *K&R*, page 167, for a description of `calloc`.

Lines 26–28:

This code sets `t` to the time of the first annotated beat in the record.

Lines 29–34:

Here we read the remainder of the annotations, skipping any non-beat annotations. The difference between the values of `annot.time` for consecutive beat annotations defines an R-R interval (`rr`). Each possible value of `rr` up to `rrmax` is assigned a bin in `rrhist`. Intervals longer than 3 seconds (`rrmax`) are counted in the bin corresponding to `rr` = `rrmax`.

Lines 35–37:

The histogram is printed as a two-column table, with the number of intervals in the first column and the length of the interval (with millisecond resolution) in the second column. (What happens if `rr` starts at 0 rather than 1 in line 35?)

Example 5: Reading Signal Specifications

This program reads the signal specifications of the record named as its argument:

```
 1  #include <stdio.h>
 2  #include <wfdb/wfdb.h>
 3
 4  main(argc, argv)
 5  int argc;
 6  char *argv[];
 7  {
 8      WFDB_Siginfo *s;
 9      int i, nsig;
10
11      if (argc < 2) {
12          fprintf(stderr, "usage: %s record\n", argv[0]);
13          exit(1);
14      }
15      nsig = isigopen(argv[1], NULL, 0);
16      if (nsig < 1) exit(2);
17      s = (WFDB_Siginfo *)malloc(nsig * sizeof(WFDB_Siginfo));
18      if (s == NULL) {
19          fprintf(stderr, "insufficient memory\n");
20          exit(3);
21      }
22      if (isigopen(argv[1], s, nsig) != nsig) exit(2);
23      printf("Record %s\n", argv[1]);
24      printf("Starting time: %s\n", timstr(0L));
```

```
25      printf("Sampling frequency: %g Hz\n", sampfreq(argv[1]));
26      printf("%d signals\n", nsig);
27      for (i = 0; i < nsig; i++) {
28          printf("Group %d, Signal %d:\n", s[i].group, i);
29          printf(" File: %s\n", s[i].fname);
30          printf(" Description: %s\n", s[i].desc);
31          printf(" Gain: ");
32          if (s[i].gain == 0.)
33              printf("uncalibrated; assume %g", WFDB_DEFGAIN);
34          else printf("%g", s[i].gain);
35          printf(" adu/%s\n", s[i].units ? s[i].units : "mV");
36          printf(" Initial value: %d\n", s[i].initval);
37          printf(" Storage format: %d\n", s[i].fmt);
38          printf(" I/O: ");
39          if (s[i].bsize == 0) printf("can be unbuffered\n");
40          else printf("%d-byte blocks\n", s[i].bsize);
41          printf(" ADC resolution: %d bits\n", s[i].adcres);
42          printf(" ADC zero: %d\n", s[i].adczero);
43          if (s[i].nsamp > 0L) {
44              printf(" Length: %s (%ld sample intervals)\n",
45                     timstr(s[i].nsamp), s[i].nsamp);
46              printf(" Checksum: %d\n", s[i].cksum);
47          }
48          else printf(" Length undefined\n");
49      }
50      exit(0);
51  }
```

(See `http://physionet.org/physiotools/wfdb/examples/example5.c` for a copy of this program.)

Notes:

Line 15: The command-line argument, `argv[1]`, is the record name. The number of signals listed in the header file for the record is returned by `isigopen` as `nsig`. If `nsig` < 1, `isigopen` will print an error message; in this case the program can't do anything useful, so it exits.

Line 17: We allocate `nsig` signal information (`WFDB_Siginfo`) objects.

Line 22: On the second invocation of `isigopen`, we pass the pointer to the signal information objects and the number of signals we expect to open. `isigopen` returns the number of signals it is able to open; if any of those named in the header file are unreadable, the return value will not match `nsig`, and the program exits.

Line 24: Invoking `timstr` with an argument of zero (here written '`0L`' to emphasize to the compiler that the argument is a `long` integer) will obtain the starting time of the record. If no starting time is defined, `timstr` will return "`0:00:00`".

Lines 31–34:
Notice how a zero value for `gain` is interpreted.

Line 35: If the `units` field is NULL, the physical units are assumed to be millivolts ("mV").

Lines 38–40:
If `bsize` is zero, I/O can be performed in blocks of any reasonable size; otherwise it must be performed in blocks of exactly the specified `bsize`.

Lines 43–48:
If the length of the record is defined, it is printed in both hours, minutes, and seconds, and in sample intervals. Since the argument of `timstr` in line 39 is positive, it is interpreted as a time interval. The checksum is defined only if the record length is defined.

Example 6: A Differentiator

The program below inverts and differentiates the signals read by `getvec` and writes the results with `putvec`. The output is readable as record `dif`. A wide variety of simple digital filters can be modelled on this example; see [Example 7], page 89, for a more general approach.

```
 1  #include <stdio.h>
 2  #include <wfdb/wfdb.h>
 3
 4  main(argc, argv)
 5  int argc;
 6  char *argv[];
 7  {
 8      WFDB_Siginfo *s;
 9      int i, nsig, nsamp=1000;
10      WFDB_Sample *vin, *vout;
11
12      if (argc < 2) {
13          fprintf(stderr, "usage: %s record\n", argv[0]); exit(1);
14      }
15      if ((nsig = isigopen(argv[1], NULL, 0)) <= 0) exit(2);
16      s = (WFDB_Siginfo *)malloc(nsig * sizeof(WFDB_Siginfo));
17      vin = (WFDB_Sample *)malloc(nsig * sizeof(WFDB_Sample));
18      vout = (WFDB_Sample *)malloc(nsig * sizeof(WFDB_Sample));
19      if (s == NULL || vin == NULL || vout == NULL) {
20          fprintf(stderr, "insufficient memory\n");
21          exit(3);
22      }
23      if (isigopen(argv[1], s, nsig) != nsig) exit(2);
24      if (osigopen("8l", s, nsig) <= 0) exit(3);
25      while (nsamp-- > 0 && getvec(vin) > 0) {
26          for (i = 0; i < nsig; i++)
27              vout[i] -= vin[i];
28          if (putvec(vout) < 0) break;
29          for (i = 0; i < nsig; i++)
```

```
30              vout[i] = vin[i];
31      }
32      (void)newheader("dif");
33      wfdbquit();
34      exit(0);
35  }
```

(See `http://physionet.org/physiotools/wfdb/examples/example6.c` for a copy of this program.)

Notes:

Line 24: Here we attempt to open as many output signals as there are input signals; if we cannot do so, the program exits after `osigopen` prints an error message.

Line 25: The main loop of the program begins here. If 1000 samples can be read from each signal, the loop will end normally; if `getvec` fails before 1000 samples have been read, the loop ends prematurely.

Lines 26–27: For each signal, we compute the negated first difference by subtracting the new sample from the previous sample.

Line 28: One sample of each output signal is written here.

Lines 29–30: The new input samples are copied into the output sample vector in preparation for the next iteration.

Line 32: This step is optional. It creates a header file for a new record to be called `dif`, which we can then open with another program if we want to read the signals that this program has written. Since the *record* argument for `osigopen` was `81`, we can also read these files using record `81`; one reason for making a new `hea` file here is that the `hea` file for `81` may not necessarily indicate the proper sampling frequency for these signals.

Line 33: Since the program writes output signals, it must invoke `wfdbquit` to close the files properly.

Example 7: A General-Purpose FIR Filter

This program illustrates the use of `sample` to obtain random access to signals, a technique that is particularly useful for implementing digital filters. The first argument is the record name, the second and third arguments are the start time and the duration of the segment to be filtered, and the rest of the arguments are finite-impulse-response (FIR) filter coefficients. For example, if this program were compiled into an executable program called '`filter`', it might be used by

```
filter 100 5:0 20 .2 .2 .2 .2 .2
```

which would apply a five-point moving average (rectangular window) filter to 20 seconds of record `100`, beginning 5 minutes into the record. The output of the program is readable as record `out`, for which a header file is created in the current directory.

```
1  #include <stdio.h>
```

```
 2  #include <wfdb/wfdb.h>
 3
 4  main(argc, argv)
 5  int argc;
 6  char *argv[];
 7  {
 8      double *c, one = 1.0, vv, atof();
 9      int i, j, nc = argc - 4, nsig;
10      WFDB_Time nsamp, t, t0, t1;
11      static WFDB_Sample *v;
12      static WFDB_Siginfo *s;
13
14      if (argc < 4) {
15          fprintf(stderr,
16            "usage: %s record start duration [ coefficients ... ]\n",
17                  argv[0]);
18          exit(1);
19      }
20      if (nc < 1) {
21          nc = 1; c = &one;
22      }
23      else if ((c = (double *)calloc(nc, sizeof(double))) == NULL) {
24          fprintf(stderr, "%s: too many coefficients\n", argv[0]);
25          exit(2);
26      }
27      for (i = 0; i < nc; i++)
28          c[i] = atof(argv[i+4]);
29      if ((nsig = isigopen(argv[1], NULL, 0)) < 1)
30          exit(3);
31      s = (WFDB_Siginfo *)malloc(nsig * sizeof(WFDB_Siginfo));
32      v = (WFDB_Sample *)malloc(nsig * sizeof(WFDB_Sample));
33      if (s == NULL || v == NULL) {
34          fprintf(stderr, "insufficient memory\n");
35          exit(4);
36      }
37      if (isigopen(argv[1], s, nsig) != nsig)
38          exit(5);
39      t0 = strtim(argv[2]);
40      if (t0 < (WFDB_Time)0) t0 = -t0;
41      (void)sample(0, t0);
42      if (!sample_valid()) {
43          fprintf(stderr, "%s: inappropriate value for start time\n",
44                 argv[0]);
45          exit(6);
46      }
47      if ((nsamp = strtim(argv[3])) < 1) {
48          fprintf(stderr, "%s: inappropriate value for duration\n",
```

```
49                  argv[0]);
50          exit(7);
51      }
52      t1 = t0 + nsamp;
53      if (osigopen("16l", s, nsig) != nsig)
54          exit(8);
55
56      for (t = t0; t < t1 && sample_valid(); t++) {
57          for (j = 0; j < nsig; j++) {
58              for (i = 0, vv = 0.; i < nc; i++)
59                  if (c[i] != 0.) vv += c[i]*sample(j, t+i);
60              v[j] = (WFDB_Sample)vv;
61          }
62          if (putvec(v) < 0) break;
63      }
64
65      (void)newheader("out");
66      wfdbquit();
67      exit(0);
68  }
```

(See `http://physionet.org/physiotools/wfdb/examples/example7.c` for a copy of this program.)

Notes:

Lines 20–22:
If no coefficients are provided on the command line, the program will simply copy the selected segment of the input signals.

Lines 23–28:
If there are more coefficients than there are samples in the circular buffer, or if memory cannot be allocated for the coefficient vector, the program cannot work properly, so it exits with an error message. In lines 27 and 28, the ASCII strings that represent the coefficients are converted to `double` format and stored in the coefficient vector.

Lines 29–40:
The record name is `argv[1]`, and the start time is `argv[2]`; if the record can't be opened, the program exits. See the previous example for details on how `isigopen` is used. If the user provides an absolute start time (see [`timstr` and `strtim`], page 36), the negative value returned by `strtim` is converted to a sample number in line 40.

Lines 41–46:
Here, `sample` is invoked only for its side effect; if any samples can be read from the specified record beginning at sample number `t0`, then `sample(0, 0L)` returns a valid sample, so that the value returned by `sample_valid` is true (1). If not, the program exits.

Lines 47–52:
The *duration* argument should be a time interval in *HH:MM:SS* format; `strtim` converts it to the appropriate number of samples, and `t1` is set to the calculated end time in line 52.

Lines 53–54:
The output signals will be written to files in the current directory according to the specifications for record `16l` (see Section 5.8 [Piped and Local Records], page 76). If we can't write as many output signals as there are input signals, the program exits.

Lines 56–63:
Here's where the work is done. The outer loop is executed once per sample vector, the middle loop once per signal, and the inner loop once per coefficient. In line 59, we retrieve an input sample, multiply it by a filter coefficient, and add it to a running sum. The sum (`vv`) is initialized to zero in line 58 before we begin, and is converted to a `WFDB_Sample` in line 60 when we are finished. Once an entire output sample vector ia ready, it is written in line 62. The entire process is repeated until we reach input sample number `t1`, or we run out of input samples.

Line 65: The program creates a header file for record `out`, using the signal specifications from record `16l` and the sampling frequency from the input record.

Example 8: Creating a New Database Record

This program creates a new record from scratch. It asks the user for information about the signals to be sampled, then records them, and finally creates a `hea` file for the new record. Details of data acquisition are hardware-dependent and are not shown here.

```
 1  #include <stdio.h>
 2  #include <wfdb/wfdb.h>
 3
 4  main()
 5  {
 6      char answer[32], record[8], directory[32];
 7      int i, nsig = 0;
 8      WFDB_Time nsamp, t;
 9      double freq = 0.;
10      char **filename, **description, **units;
11      WFDB_Sample *v;
12      WFDB_Siginfo *s;
13
14      do {
15          printf("Choose a record name [up to 6 characters]: ");
16          fgets(record, 8, stdin); record[strlen(record)-1] = '\0';
17      } while (newheader(record) < 0);
18      do {
19          printf("Number of signals to be recorded [>0]: ");
20          fgets(answer, 32, stdin); sscanf(answer, "%d", &nsig);
```

```
21      } while (nsig < 1);
22      s = (WFDB_Siginfo *)malloc(nsig * sizeof(WFDB_Siginfo));
23      v = (WFDB_Sample *)malloc(nsig * sizeof(WFDB_Sample));
24      filename = (char **)malloc(nsig * sizeof(char *));
25      description = (char **)malloc(nsig * sizeof(char *));
26      units = (char **)malloc(nsig * sizeof(char *));
27      if (s == NULL || v == NULL || filename == NULL ||
28          description == NULL || units == NULL) {
29          fprintf(stderr, "insufficient memory\n");
30          exit(1);
31      }
32      for (i = 0; i < nsig; i++) {
33          if ((filename[i] = (char *)malloc(32)) == NULL ||
34              (description[i] = (char *)malloc(32)) == NULL ||
35              (units[i] = (char *)malloc(32)) == NULL) {
36              fprintf(stderr, "insufficient memory\n");
37              exit(1);
38          }
39      }
40      do {
41          printf("Sampling frequency [Hz per signal, > 0]: ");
42          fgets(answer, 32, stdin); sscanf(answer, "%lf", &freq);
43      } while (setsampfreq(freq) < 0);
44      do {
45          printf("Length of record (H:M:S): ");
46          fgets(answer, 32, stdin);
47      } while ((nsamp = strtim(answer)) < 1L);
48      printf("Directory for signal files [up to 30 characters]: ");
49      fgets(directory, 32, stdin);
50      directory[strlen(directory)-1] = '\0';
51      printf("Save signals in difference format? [y/n]: ");
52      fgets(answer, 32, stdin);
53      s[0].fmt = (answer[0] == 'y') ? 8 : 16;
54      printf("Save all signals in one file? [y/n]: ");
55      fgets(answer, 32, stdin);
56      if (answer[0] == 'y') {
57          sprintf(filename[0], "%s/d.%s", directory, record);
58          for (i = 0; i < nsig; i++) {
59               s[i].fname = filename[0];
60               s[i].group = 0;
61          }
62      }
63      else {
64          for (i = 0; i < nsig; i++) {
65               sprintf(filename[i], "%s/d%d.%s", directory,i,record);
66               s[i].fname = filename[i];
67               s[i].group = i;
```

```
68          }
69      }
70      for (i = 0; i < nsig; i++) {
71          s[i].fmt = s[0].fmt; s[i].bsize = 0;
72          printf("Signal %d description [up to 30 characters]: ", i);
73          fgets(description[i], 32, stdin);
74          description[i][strlen(description[i])-1] = '\0';
75          s[i].desc = description[i];
76          printf("Signal %d units [up to 20 characters]: ", i);
77          fgets(units[i], 22, stdin);
78          units[i][strlen(units[i])-1] = '\0';
79          s[i].units = (*units[i]) ? units[i] : "mV";
80          do {
81              printf(" Signal %d gain [adu/%s]: ", i, s[i].units);
82              fgets(answer, 32, stdin);
83              sscanf(answer, "%lf", &s[i].gain);
84          } while (s[i].gain < 0.);
85          do {
86              printf(" Signal %d ADC resolution in bits [8-16]: ", i);
87              fgets(answer, 32, stdin);
88              sscanf(answer, "%d", &s[i].adcres);
89          } while (s[i].adcres < 8 || s[i].adcres > 16);
90          printf(" Signal %d ADC zero level [adu]: ", i);
91          fgets(answer, 32, stdin);
92          sscanf(answer, "%d", &s[i].adczero);
93      }
94      if (osigfopen(s, nsig) < nsig) exit(1);
95      printf("To begin sampling, press RETURN;  to specify a\n");
96      printf(" start time other than the current time, enter\n");
97      printf(" it in H:M:S format before pressing RETURN: ");
98      fgets(answer, 32, stdin); answer[strlen(answer)-1] = '\0';
99      setbasetime(answer);
100
101     adinit();
102
103     for (t = 0; t < nsamp; t++) {
104         for (i = 0; i < nsig; i++)
105             v[i] = adget(i);
106         if (putvec(v) < 0) break;
107     }
108
109     adquit();
110
111     (void)newheader(record);
112     wfdbquit();
113     exit(0);
114 }
```

(See `http://physionet.org/physiotools/wfdb/examples/example8.c` for a copy of this program.)

Notes:

Lines 14–17:
This code uses `newheader` to determine if a legal record name was entered (since we don't want to digitize the signals and then find out that we can't create the header file). The header file created in line 17 will be overwritten in line 111.

Lines 57–62:
This code generates a file name and initializes the `fname` and `group` fields of the array of `WFDB_Siginfo` objects so that all signals will be saved in one file.

Lines 63–69:
This code generates unique file names and groups for each signal.

Lines 70–93:
Here, information specific to individual signals is gathered.

Line 94: If the signal files can't be created, this program can do nothing else useful, so it quits with an error message from `osigfopen`.

Lines 95–99:
Just before sampling begins, we set the base time. Note that an empty string argument for `setbasetime` gives us the current time read from the system clock.

Line 101: What goes here will be hardware dependent. Typically it is necessary to set up a timer for the ADC, allocate DMA buffers, specify interrupt vectors, and initiate the first conversion(s). This program might also be used to create a database record from prerecorded data in a non-supported format; in this case, we might simply open the file containing the prerecorded data here.

Lines 103–107:
Here is where the samples are acquired (using hardware-dependent code not shown here) and recorded (using `putvec`). At high sampling frequencies, it is critical to make this code as fast as possible. It could be made faster by judicious use of `register` and pointer variables if necessary. In an extreme case the entire loop, possibly including `putvec` itself, can be written in assembly language; since it is only a small fraction of the entire program, doing so is within reason.

Line 109: This final piece of hardware-dependent code typically clears the ADC control register, stops the timer, and frees any system resources such as DMA channels or interrupts.

Line 111: All of the information needed to generate the header file has been stored in WFDB library internal data structures by `osigfopen` and `putvec`; we call `newheader` here (before `wfdbquit`) to create the new `hea` file.

Line 112: It is still necessary to use `wfdbquit` to close the signal file(s), even after calling `newheader`. (In fact, it would be possible, though not likely to be useful, to record more samples and to generate another header file before calling `wfdbquit`.)

Example 9: A Signal Averager

The following program is considerably more complex than the previous examples in this chapter. It reads an annotation file (for which the annotator name is specified in its first argument, and the record name in the second argument) and selects beats of a specified type to be averaged. The program selects segments of the signals that are within 50 milliseconds of the time of the specified beat annotations, subtracts a baseline estimate from each sample, and calculates an average waveform (by default, the average normal QRS complex).

```
 1  #include <stdio.h>
 2  #include <wfdb/wfdb.h>
 3  #include <wfdb/ecgmap.h>
 4
 5  main(argc, argv)
 6  int argc;
 7  char *argv[];
 8  {
 9      int btype, i, j, nbeats = 0, nsig, hwindow, window;
10      WFDB_Time stoptime = 0;
11      long **sum;
12      WFDB_Anninfo a;
13      WFDB_Annotation annot;
14      WFDB_Sample *v, *vb;
15      WFDB_Siginfo *s;
16
17      if (argc < 3) {
18          fprintf(stderr,
19                  "usage: %s annotator record [beat-type from to]\n",
20                  argv[0]);
21          exit(1);
22      }
23      a.name = argv[1]; a.stat = WFDB_READ;
24      if ((nsig = isigopen(argv[2], NULL, 0)) < 1) exit(2);
25      s = (WFDB_Siginfo *)malloc(nsig * sizeof(WFDB_Siginfo));
26      v = (WFDB_Sample *)malloc(nsig * sizeof(WFDB_Sample));
27      vb = (WFDB_Sample *)malloc(nsig * sizeof(WFDB_Sample));
28      sum = (long **)malloc(nsig * sizeof(long *));
29      if (s == NULL || v == NULL || vb == NULL || sum == NULL) {
30          fprintf(stderr, "%s: insufficient memory\n", argv[0]);
31          exit(2);
32      }
33      if (wfdbinit(argv[2], &a, 1, s, nsig) != nsig) exit(3);
34      hwindow = strtim(".05"); window = 2*hwindow + 1;
35      for (i = 0; i < nsig; i++)
36          if ((sum[i]=(long *)calloc(window,sizeof(long))) == NULL) {
37              fprintf(stderr, "%s: insufficient memory\n", argv[0]);
38              exit(2);
39          }
```

```
40      btype = (argc > 3) ? strann(argv[3]) : NORMAL;
41      if (argc > 4) iannsettime(strtim(argv[4]));
42      if (argc > 5) {
43          if ((stoptime = strtim(argv[5])) < 0L)
44              stoptime = -stoptime;
45          if (s[0].nsamp > 0L && stoptime > s[0].nsamp)
46              stoptime = s[0].nsamp;
47      }
48      else stoptime = s[0].nsamp;
49      if (stoptime > 0L) stoptime -= hwindow;
50      while (getann(0, &annot) == 0 && annot.time < hwindow)
51          ;
52      do {
53          if (annot.anntyp != btype) continue;
54          isigsettime(annot.time - hwindow - 1);
55          getvec(vb);
56          for (j = 0; j < window && getvec(v) > 0; j++)
57              for (i = 0; i < nsig; i++)
58                  sum[i][j] += v[i] - vb[i];
59          nbeats++;
60      } while (getann(0, &annot) == 0 &&
61               (stoptime == 0L || annot.time < stoptime));
62      if (nbeats < 1) {
63          fprintf(stderr, "%s: no ‘%s’ beats found\n",
64                  argv[0], annstr(btype));
65          exit(4);
66      }
67      printf("Average of %d ‘%s’ beats:\n", nbeats, annstr(btype));
68      for (j = 0; j < window; j++)
69          for (i = 0; i < nsig; i++)
70              printf("%g%c", (double)sum[i][j]/nbeats,
71                     (i == nsig-1) ? '\n' : '\t');
72      exit(0);
73  }
```

(See `http://physionet.org/physiotools/wfdb/examples/example9.c` for a copy of this program.)

Notes:

Line 34: The "half-window" is 50 milliseconds wide, and the "window" (the duration of a segment to be entered into the average) is one sample more than twice that amount (i.e., 50 milliseconds to either side of the fiducial point defined by the annotation).

Lines 35–39:
Here we allocate memory for the `sum` vectors that will be used to store the running totals. See *K&R*, page 167, for a description of `calloc`.

Line 40: If a third argument is present on the command line, it is taken as an annotation code mnemonic for the desired beat type; otherwise, the program will average `NORMAL` QRS complexes.

Line 41: If a fourth argument is present on the command line, it is taken as the start time; we arrange for the first annotation to be read by `getann` to be the first annotation that occurs after the chosen start time.

Lines 42–49:
This code similarly determines when the averaging should stop. Unless no stop time was specified on the command line and the signal length is not defined in the `hea` file for the record, `stoptime` will have a positive value in line 49, which makes a tiny adjustment so that if a beat annotation occurs within 50 milliseconds of the end of the averaging period, the beat will not be included in the average.

Lines 50-51:
This code addresses the (admittedly unlikely) prospect that the first annotation(s) might occur within the first 50 milliseconds of the record; any such annotations will be excluded from the average.

Lines 52–61:
Here we read annotations (the first is already in `annot` when we enter the loop, and subsequent annotations are read in line 60); select the desired ones (line 53); skip to the correct spot in the signals (line 54; the sample selected there is the one just before the beginning of the window); read a sample from each signal (line 55) into the `vb` vector, which will be used as a crude baseline estimate; read `window` samples from each signal (line 56), subtracting the baseline from each and adding the result into the running totals; update a beat counter (line 59); and check for loop termination conditions (line 61).

Lines 62–71:
This is the output section. If no beats of type `btype` were found, obviously no average can be printed; note that the message goes to the standard error output, so the user will notice it even if the standard output has been redirected to a file. In the usual case, the averages are printed out as a table, with a column allocated to each signal. Note the cast in line 70 (necessary to preserve precision), and the trick used in line 71 to print a tab after each column but the last in each line.

Example 10: A QRS Detector

This program reads a single ECG signal, attempts to detect QRS complexes, and records their locations in an annotation file. The detector algorithm is based on a Pascal program written by W.A.H. Engelse and C. Zeelenberg, "A single scan algorithm for QRS-detection and feature extraction", *Computers in Cardiology* **6**:37-42 (1979).

```
1  #include <stdio.h>
2  #include <wfdb/wfdb.h>
3  #include <wfdb/ecgcodes.h>
4
```

```
 5  #define abs(A)  ((A) >= 0 ? (A) : -(A))
 6
 7  main(argc, argv)
 8  int argc;
 9  char *argv[];
10  {
11      int filter, time=0, slopecrit, sign, maxslope=0, nsig, nslope=0,
12          qtime, maxtime, t0, t1, t2, t3, t4, t5, t6, t7, t8, t9,
13          ms160, ms200, s2, scmax, scmin = 0;
14      WFDB_Anninfo a;
15      WFDB_Annotation annot;
16      WFDB_Sample *v;
17      WFDB_Siginfo *s;
18
19      if (argc < 2) {
20          fprintf(stderr, "usage: %s record [threshold]\n", argv[0]);
21          exit(1);
22      }
23      a.name = "qrs"; a.stat = WFDB_WRITE;
24
25      if ((nsig = isigopen(argv[1], NULL, 0)) < 1) exit(2);
26      s = (WFDB_Siginfo *)malloc(nsig * sizeof(WFDB_Siginfo));
27      v = (WFDB_Sample *)malloc(nsig * sizeof(WFDB_Sample));
28      if (s == NULL || v == NULL) {
29          fprintf(stderr, "%s: insufficient memory\n", argv[0]);
30          exit(2);
31      }
32      if (wfdbinit(argv[1], &a, 1, s, nsig) != nsig) exit(2);
33      if (sampfreq((char *)NULL) < 240. ||
34          sampfreq((char *)NULL) > 260.)
35          setifreq(250.);
36      if (argc > 2) scmin = muvadu(0, atoi(argv[2]));
37      if (scmin < 1) scmin = muvadu(0, 1000);
38      slopecrit = scmax = 10 * scmin;
39      ms160 = strtim("0.16"); ms200 = strtim("0.2"); s2 = strtim("2");
40      annot.subtyp = annot.chan = annot.num = 0; annot.aux = NULL;
41      (void)getvec(v);
42      t9 = t8 = t7 = t6 = t5 = t4 = t3 = t2 = t1 = v[0];
43
44      do {
45          filter = (t0 = v[0]) + 4*t1 + 6*t2 + 4*t3 + t4
46                  - t5         - 4*t6 - 6*t7 - 4*t8 - t9;
47          if (time % s2 == 0) {
48              if (nslope == 0) {
49                  slopecrit -= slopecrit >> 4;
50                  if (slopecrit < scmin) slopecrit = scmin;
51              }
```

```
52              else if (nslope >= 5) {
53                  slopecrit += slopecrit >> 4;
54                  if (slopecrit > scmax) slopecrit = scmax;
55              }
56          }
57          if (nslope == 0 && abs(filter) > slopecrit) {
58              nslope = 1; maxtime = ms160;
59              sign = (filter > 0) ? 1 : -1;
60              qtime = time;
61          }
62          if (nslope != 0) {
63              if (filter * sign < -slopecrit) {
64                  sign = -sign;
65                  maxtime = (++nslope > 4) ? ms200 : ms160;
66              }
67              else if (filter * sign > slopecrit &&
68                       abs(filter) > maxslope)
69                  maxslope = abs(filter);
70              if (maxtime-- < 0) {
71                  if (2 <= nslope && nslope <= 4) {
72                      slopecrit += ((maxslope>>2) - slopecrit) >> 3;
73                      if (slopecrit < scmin) slopecrit = scmin;
74                      else if (slopecrit > scmax) slopecrit = scmax;
75                      annot.time = strtim("i") - (time - qtime) - 4;
76                      annot.anntyp = NORMAL; (void)putann(0, &annot);
77                      time = 0;
78                  }
79                  else if (nslope >= 5) {
80                      annot.time = strtim("i") - (time - qtime) - 4;
81                      annot.anntyp = ARFCT; (void)putann(0, &annot);
82                  }
83                  nslope = 0;
84              }
85          }
86          t9 = t8; t8 = t7; t7 = t6; t6 = t5; t5 = t4;
87          t4 = t3; t3 = t2; t2 = t1; t1 = t0; time++;
88      } while (getvec(v) > 0);
89
90      wfdbquit();
91      exit(0);
92  }
```

(See `http://physionet.org/physiotools/wfdb/examples/example10.c` for a copy of this program.)

Notes:

Line 5: A macro that evaluates to the absolute value of its argument.

Lines 11–12:

The names of these variables match those in the original Pascal program.

Lines 33–35:

Most of this program is independent of sampling frequency, but the filter (lines 45–46) and the threshold are as specified by the authors of the original program for human ECGs sampled at 250 Hz (e.g., the AHA DB). If the sampling frequency of the input record is significantly different, we use `setifreq` to specify that we want `getvec` to give us data resampled at 250 Hz. The output annotation file is created in line 35 only after invoking `setifreq` if necessary.

Lines 36–38:

The threshold is actually a slope criterion (with units of amplitude/time); these lines normalize the threshold with respect to the signal gain. The default value is used unless the user supplies an acceptable alternative. The variables `scmin` and `scmax` are lower and upper bounds for the adaptive threshold `slopecrit`.

Lines 41–42:

Here we read the first sample and copy it into the variables that will be used to store the ten most recent samples.

Lines 45–46:

This FIR filter differentiates and low-pass filters the input signal.

Lines 47–56:

Here we adjust the threshold if more than two seconds have elapsed since a QRS was detected. In line 49, `slopecrit` is set to 15/16 of its previous value if no slopes have been found; in line 53, it is set to 17/16 of its previous value if 5 or more slopes were found (suggesting the presence of noise).

Lines 57–61:

If the condition in line 48 is satisfied, we may have found the beginning of a QRS complex. We record that a slope has been found, set the timer `maxtime` to 160 msec, and save the sign of the slope and the current time relative to the previous beat.

Lines 62–85:

This code is executed once we have found a slope. Each time the filter output crosses the threshold, we record another slope and begin looking for a threshold crossing of the opposite sign (lines 63–66), which must occur within a specified time. We record the maximum absolute value of the filter in `maxslope` (lines 67–69) for eventual use in updating the threshold (lines 72–74). Once a sufficient interval has elapsed following the last threshold crossing (line 70), if there were between 2 and 4 slopes, we have (apparently) found a QRS complex, and the program records a `NORMAL` annotation (lines 75–76). If there were 5 or more slopes, the program records an artifact annotation (lines 80–81). If only 1 slope was found, it is assumed to represent a baseline shift and no output is produced.

Lines 86–88:

At the end of the loop, the samples are shifted through the `tn` variables and another sample is read.

Exercises

These exercises are based on the material in the previous chapters. Answers to some of them are at the back of the book, but try to work through them first.

1. Type in the first program from the previous chapter, compile it, and run it. If you know that you will need to read WFDB files from non-standard locations, remember to set and export the environment variable `WFDB` (see Section 1.4 [WFDB path], page 12). It is a good idea to include this step in your `.profile`, `.cshrc`, or `autoexec.bat`. As input, try record `100s`, input annotator `atr`, and output annotator `normal`. The program should finish in five seconds or less. The annotations will have been written into a file called `100s.nor` in the current directory. Now type "`rdann -r 100s -a atr`" and observe the output for a few seconds, then try "`rdann -r 100s -a nor`" and notice the difference.
2. Modify the program from the previous exercise so that the non-QRS annotations are put into a second output annotation file. Remember that you will need three annotation files in all (one input and two output).
3. The next five short exercises are to be worked out on paper, although you may wish to check your work on the computer. All of them assume that we are given a signal sampled at 100 Hz with the following specifications:

   ```
   fname = "signal.dat"
   desc = "BP"
   units = "mmHg"
   gain = 10
   initval = 80
   group = 0
   fmt = 212
   spf = 1
   bsize = 0
   adcres = 12
   adczero = 0
   baseline = -300
   nsamp = 1000000
   cksum = 3109
   ```

   For starters, convert a sample value of 280 into physical units.
4. Convert 120 mmHg into adus.
5. What are the maximum and minimum possible sample values in adu? in mmHg?
6. How large is `signal.dat`, in bytes? How much space could we save if we converted it to format 8 (eight-bit first-differences)? What is the maximum slew rate (in mmHg/second) that we can represent in that format?
7. Oops! We have just discovered that the maximum slew rate in our signal is 1500 mmHg/sec. Is there any way to store it at full precision in one of the supported formats, that saves space compared to its present format?
8. Figure out how to plot or display the first 1000 points from signal 0 of a record in amplitude vs. time format. You may wish to begin with the example program from

the first chapter. Arrange for the record name to be read from the command line (see *K&R*, pp. 114–115, if you don't know how to do this).

9. Try plotting VCGs by modifying the program from the previous exercise to plot pairs of samples from each of two signals rather than sample number/value pairs.

10. Modify the program from the previous exercise, or Example 2 from the previous chapter, so that you can specify a segment of the record to be processed with start and end times. For example, the command

    ```
    your-program record 10:0 10:10
    ```

    should skip the first ten minutes, then process the next ten seconds of signals from *record*.

11. Using `isigsettime` on a format 8 signal introduces a random offset into the signal, since the contents of a format 8 signal file are first differences rather than amplitudes. For an AC-coupled signal such as an ECG, this is usually inconsequential, but a DC-coupled signal such as a blood pressure signal is usually useful only if absolute levels are known. If we store such a signal in format 8, we must read it sequentially from the beginning in order to get correct sample values. If we intend to do a lot of non-sequential processing of such a signal, it may be worthwhile to build a table containing the correct sample values at periodic intervals; then we can use `isigsettime` to skip to a sample in the table, and read forward sequentially from that point. Write a program to build such a table, and wrappers for `isigsettime` and `getvec` to give random access to format 8 signal files without introducing offset errors. On your system, how many sample intervals should be allowed between table entries in order to obtain an `isigsettime` equivalent that executes in an average of 100 msec or less?

12. This exercise and the next assume that you have access to the web, so that you can obtain the freely available input records needed from PhysioNet. Since the 360 Hz sampling frequency used in the MIT-BIH Arrhythmia Database is an integer multiple of the 60 Hz mains frequency, it is quite easy to design a 60 Hz notch filter that can be applied to the database to suppress power-line interference (for example, by averaging pairs of samples that are 180 degrees out of phase). Write a program that filters two input signals and writes out the filtered data using `putvec` (see [Example 7], page 89, for a model program). Try it out on MIT DB record `122` (if you have a NETFILES-enabled WFDB library, use the default WFDB path, and open record `mitdb/122`; otherwise, download the record from `http://physionet.org/physiobank/database/mitdb/`.) Use your programs from the previous exercises to display your output and compare it with the original signals.

13. (Non-trivial) Write a QRS detector that is independent of sampling frequency without using `setifreq`. Some useful constants (for adult human ECGs): average normal QRS duration = 80 milliseconds, average QRS amplitude = 1 millivolt, average R-R interval = 1 second; assume that upper and lower limits for these quantities are within a factor of 3 of the average values. Run your detector on MIT-BIH Arrhythmia Database record `200`. (If you have a NETFILES-enabled WFDB library, use the default WFDB path, and open record `mitdb/200`; otherwise, download the record from `http://physionet.org/physiobank/database/mitdb/`.) Read the documentation on the annotation comparator, `bxb`, and figure out how to use it to compare the annotation file produced

by your program against the reference annotator `atr`. How does your detector compare to Example 10?

Appendix A Glossary

AC-coupled signal
: A signal, such as an ECG, for which only variations in level, rather than absolute levels, are significant. Such signals are usually passed through high-pass filters before they are digitized, in order to remove any DC component (baseline offset), so that the gain can be chosen optimally for the range of variation in the signal.

ADC
: Analog-to-digital converter.

ADC resolution
: The number of significant bits per sample. Typical ADCs yield between 8 and 16 bits of resolution.

ADC zero
: The value produced by the ADC given a 0 volt input. For bipolar ADCs, this value is usually 0, but for the unipolar (offset binary) converter used for the MIT DB, the ADC zero was 1024.

adu
: The unit of amplitude for samples.

AHA DB
: The American Heart Association Database for the Evaluation of Ventricular Arrhythmia Detectors, consisting of 80 records identified by four-digit record names.

AHA format
: The format formerly used for interchange of AHA DB and MIT DB records on 9-track tape between institutions, not used for on-line files because it is relatively wasteful of storage space compared to other WFDB-compatible formats. (q.v.).

Annotation
: A label, associated with a particular sample, which describes a feature of the signal at that time. Most annotations are QRS annotations and indicate the QRS type (normal, PVC, SVPB, etc.). Annotations are written by `putann` and read by `getann`.

Annotation code
: An integer in the range of 1 to `ACMAX` (defined in `<wfdb/ecgcodes.h>`) inclusive, which denotes an event type.

Annotation file
: A set of annotations in time order.

Annotator name
: A name associated with an annotation file. The annotation file name is constructed from the record name by appending a '`.`' and the annotator name. On CDROMs and MS-DOS file systems, the annotator name is restricted to three characters.

Annotator [number]
: An integer by which an annotation file, once opened, is known. Input annotators and output annotators each have their own series of annotator numbers assigned in serial order beginning with 0.

Application program
: In this guide, a program that uses the WFDB library to do something.

`atr`
: The annotator name for the reference annotation files (originally, `atruth`, i.e., the "truth" annotations).

Base counter value
: The counter value (q.v.) that corresponds to sample 0. The base counter value is read by `getbasecount`, and set by `setbasecount` (or by any of the functions that read header files). If not defined explicitly, the base counter value is taken to be 0.

Base time
: The time of day that corresponds to sample 0 in a given record. For MIT, AHA, and ESC DB records, the base time was not recorded and is taken to be 0:0:0 (midnight).

Baseline [amplitude]
: The sample value that corresponds to the baseline (isoelectric level or physical zero level) in the signal. This quantity may drift during the record for a variety of reasons, in which case the `baseline` field of the `WFDB_Siginfo` object that describes the signal is only an approximation. The baseline is *not* the same as the ADC zero (q.v.), which is a fixed characteristic of the digitizer.

Calibration file
: A file containing data used to build a calibration list (q.v.).

Calibration list
: A memory-resident linked list of `WFDB_Calinfo` objects (see Section 3.2 [Calibration Information Structures], page 60). Each such structure specifies the size and type of the calibration pulse, and the customary plotting scale, for a particular type of signal.

CDROM
: A read-only medium used for distribution of a number of databases readable using the WFDB library, including the ESC and Long-Term ST databases, among others. CDROMs are physically identical in appearance to audio compact disks.

Closing [a record]
: The process of completing I/O associated with a record.

Counter frequency
: The difference between counter values (q.v.) that are separated by an interval of one second. The counter frequency is constant throughout any given record. It may be undefined, in which case it is treated as equivalent to the sampling frequency (q.v.) by the WFDB library. The counter frequency is read by `getcfreq`, and set by `setcfreq` (or by any of the functions that read header files).

Counter value
: A number that serves as a time reference, in a record for which a counter frequency is defined. A counter value may be converted to the time in seconds from the beginning of the record by subtracting the base counter value (q.v.) and dividing the remainder by the counter frequency. The units of '`c`'-prefixed `strtim` arguments are counter values.

Database files
: Those files (annotation files, header files, signal files, and calibration files) that are accessed via the WFDB library.

Database path
: The names of the directories in which header, annotation, and calibration files are kept. (Signal files may be located in these directories or elsewhere; header files specify their locations.) To modify the database path, the environment variable `WFDB` must be set by the user and exported accordingly.

DC-coupled signal
: A signal, such as a blood pressure signal, for which absolute levels are significant. Such signals must be digitized without being passed through high-pass filters, in order to preserve absolute levels.

ESC DB
: The European ST-T Database, consisting of 90 records identified by '`e`'-prefixed four-digit record names.

Frame
: A set of samples, containing all samples that occur within a given frame interval. For an ordinary record, a frame contains exactly one sample of each signal; for a multi-frequency record, a frame contains *at least* one sample of each signal, and more than one sample of each oversampled signal (q.v.).

Frame interval
: A time interval during which at least one sample exists for each signal. For an ordinary record, the frame interval and the sampling interval are identical. For a multi-frequency record, the frame interval is chosen to be an integer multiple of each sampling frequency used.

Frame rate
: The basic sampling frequency defined for a multi-frequency record; the reciprocal of the frame interval. The frame rate is usually the lowest sampling frequency used for any signal included in the record.

Gain
: In this context, the number of adus (q.v.) per physical unit, referred to the original analog signal. Gain in this sense is directly proportional to the degree of amplification (the usual meaning of the word) of the analog signal prior to digitization. Gain may vary between signals in a record.

`hea`
: The suffix (extension) that designates a header file (originally `header`).

header file
: A file accessible via the WFDB library that describes the signal files associated with a given database record. A header file has a name of the form '*record*`.hea`', where *record* is the record name (q.v.).

High-resolution mode
: An alternative mode for reading a multi-frequency record using `getvec`, that can be selected using `setgvmode`. In high-resolution mode, `getvec` replicates samples of signals digitized at less than the maximum sampling frequency, so that each sample of any oversampled signals appear in at least one sample vector.

Info string
: Free text within a header file. Info strings can be read using `getinfo` and written using `putinfo`.

Local record
: A record for which the signal files reside in the current directory, typically used for user-created signals. Records `8l` and `16l` are local records.

Location [of an annotation]
: Every annotation has `time`, `num`, and `chan` attributes that define its location within a virtual array of annotations. See *Canonical order of annotations*.

Canonical order of annotations
: Normally, annotations are arranged in time order within an annotation file. Annotations that have identical `time` attributes are arranged in `num` and `chan` order. Annotations that have identical *locations* (i.e., identical `time`, `num`, and `chan` attributes) should not normally occur in a single annotation file; if this does happen, the last annotation at any given location is treated as a replacement of any previous annotations at that location.

Low-resolution mode
: The default mode for reading a multi-frequency record using `getvec`. In low-resolution mode, `getvec` returns one sample per signal per frame, by decimating any oversampled signals to the frame rate.

MIT DB
: The Massachusetts Institute of Technology–Beth Israel Hospital Arrhythmia Database, consisting of 48 records identified by three-digit record names.

MIT format
: One of a set of WFDB-compatible formats for header, signal, and annotation files that were first used for the MIT-BIH Arrhythmia Database. The term WFDB-compatible format should be used unless referring specifically to the formats used for the MIT DB (single-segment header format, signal format 212, and bit-packed annotation format).

Modification label
: An "invisible" annotation at the beginning of an annotation file. A modification label defines an annotation mnemonic and a corresponding description. When `annopen` (or `wfdbinit`) opens an annotation file that contains modification labels, it automatically calls `setannstr` and `setanndesc` to add the mnemonics and descriptions to the translation tables used by `annstr`, `strann`, and `anndesc`. When `annopen` (or `wfdbinit`) creates an annotation file, it automatically generates modification labels, for each annotation code that has been (re)defined using `setannstr` or `setanndesc`. For this reason, you should normally make all of your calls to `setannstr` and `setanndesc` *before* calling `annopen` or `wfdbinit`. (An exception is if you are simply *translating* mnemonics and descriptions into another language, rather than *redefining* them.) Version 5.3 and later versions of the WFDB library support reading and writing modification labels; earlier versions read modification labels as `NOTE` annotations.

Multi-frequency record
: A record containing signals sampled at two or more sampling frequencies. Version 9.0 and later versions of the WFDB library support reading and writing multi-frequency records.

Multi-segment record
: A composite record that is the concatenation of two or more ordinary (single-segment) records. Multi-segment records do not have their own signal files (the signal files of their constituent segments are read when it is necessary to read signals of multi-segment records), but they have their own header files (created using `setmsheader`), and may have their own annotation files as well (annotation files for the constituent segments of a multi-segment record are *not* concatenated automatically when the record is read). The WFDB Software Package includes `wfdbcollate`, an application that can create multi-segment records from sets of single-segment records. Version 9.1 and later versions of the WFDB library support reading and writing multi-segment records.

Multiplexed signal file
: A set of vectors in time order, each consisting of two or more integer samples, thus representing an equal number of signals.

NETFILES
: WFDB files made available by an FTP or HTTP (web) server; readable by applications linked with a NETFILES-enabled WFDB library. A NETFILES-enabled WFDB library can be created by compiling the WFDB library sources with the symbol `WFDB_NETFILES` defined (to anything; its value is not important, only that it is defined) and then linking them with the `libcurl` library available from `http://curl.haxx.se/`.

9-track tape
: A medium used for archival storage of WFDB records, which was once nearly universally available on minicomputers and larger systems. The important parameters are tape density (typically 800 or 1600 bpi) and block size (typically some multiple of 512 bytes). Higher tape density and larger block size permit more data to be stored on a tape.

Opening [a database record or a file]
: The process of making a database record or a file accessible, if necessary by creating it.

Oversampled signal
: In a multi-frequency record, any signal recorded at a sampling frequency greater than the frame rate (q.v.).

Physical unit
: The natural unit of measurement of the original analog signal (e.g., millivolts, liters per second, degrees). To convert samples into physical units, subtract the ADC zero and divide the remainder by the gain.

Physical zero
: The level (in physical units) that corresponds to the baseline (in adu), normally zero physical units. For example, physical zero for a pressure signal with units of mmHg is 0 mmHg.

PhysioNet
: The home of the WFDB library, and a source for recorded physiologic signals and software for use with them. All materials on PhysioNet are freely available. The main PhysioNet server is `http://physionet.org/`, located at MIT in Cambridge, Massachusetts; PhysioNet mirror sites are located around the world (see `http://physionet.org/mirrors/` for a list).

Piped record
: A database record for which a signal file is designated as `-`, signifying that it is to be read from the standard input or written to the standard output. Records `8` and `16` are piped records, as are those defined within the `pipe` subdirectory of the system-wide database directory (q.v.)

Prolog
: Extraneous bytes at the beginning of a signal file that are not to be read as samples. Signal files created using the WFDB library do not contain prologs, but signal files created using other means may contain prologs. To read such a signal file using the WFDB library, provided that the sample data are in a supported format, it is sufficient to record the length of the prolog (in bytes) in the appropriate locations in a header file that names the signal file. If you need to create such a header file, refer to the description of the byte offset field in *header(5)* (the specification of the header file format in the *WFDB Applications Guide*) or see [wfdbsetstart], page 53.

Record
: An extensible set of files that may include signal files, annotation files, and a header file, all of which are associated with the same original signals. Only the header file is mandatory. Although records are sometimes called tapes for historical reasons, records are now more commonly maintained on CDROMs or magnetic disks than on tape.

Record name
: A character string that identifies a database record. Record names of MIT DB records are 3-digit numerals, those of AHA DB records are 4-digit numerals, and those of ESC DB records are 4-digit numerals with a prefixed '`e`'. Record names may contain up to `WFDB_MAXRNL` (defined in `<wfdb/wfdb.h>`) characters, including any combination of letters, digits, and underscores. Case (the difference between '`e`' and '`E`', for example) is significant in record names, even under operating systems such as MS-DOS that do not treat case as significant in file names.

Reference annotation file
: An annotation file supplied by the creator of a record to document its contents as accurately and thoroughly as possible. The annotator name `atr` is reserved for reference annotation files.

Sample
: An integer (of at least 16 bits) that corresponds to a voltage measured at a given instant by an analog-to-digital converter. Samples are written by `putvec` and read by `getvec`.

Sample interval
: The unit of time; the interval between consecutive samples of a given signal.

Sample number
: An attribute of a sample defined as the number of samples of the same signal that precede it; thus the first sample of any signal has sample number 0. Sample numbers are long integers (32 bits). Samples that have the same sample number in different signals of a given record may be treated as having been observed simultaneously.

Sampling frequency
: The number of samples of a given signal that represent one second of the original analog signal. The sampling frequency is constant throughout a signal file, and is the same for all signals in a given record.

Signal
: A continuously varying function of time that is approximated by discrete samples.

Signal file
: A set of samples in time order, which represent a signal or signal group. Signal files usually have names of the form *record*`.dat`, but this is only a convention and is not required.

Signal group
: A set of signals that are multiplexed together and stored in the same file. It is possible to reset input pointers for all signals in a given signal group (see [`isgsettime`], page 32), but not independently for individual signals within a signal group.

Signal group number
: A number by which a signal file, once opened, is known.

Signal number
: An integer by which a signal, once opened, is known. Input and output signals each have their own series of signal numbers assigned in serial order beginning with 0.

Skew
: The time difference between samples having the same sample number but belonging to different signals. Ideally the skew is zero (or less than one sample interval), but in some cases this is not so. For example, if the signals were originally recorded on multitrack analog tape, very small differences in the azimuth of the recording and playback heads may result in measurable skew among signals. If the skew can be measured (for example, by reference to features of two signals with a known time difference), it can be recorded in the header file for a record; once this has been done, `getvec` and `getframe` correct for skew automatically. If you need to correct for skew, see *skewedit(1)* and *header(5)* (in the *WFDB Applications Guide*), or see [wfdbsetskew], page 53. Prospectively, if you anticipate that skew may be a problem, it is a good idea to apply an easily identifiable synchronization pulse to all your inputs simultaneously while recording; you can then locate this pulse in each digitized signal and use these measurements to correct for skew.

Standard time format
: Any string format legal as an argument for `strtim` (see [`timstr` and `strtim`], page 36).

System-wide database directory
: The directory that contains local copies of the default WFDB calibration file, WFDB sample record `100s`, and local, piped, and tape header files. This directory is created when the WFDB Software Package is installed, and by default it is included in the WFDB path (as the second component, following the user's current directory). It is called the "system-wide" database directory because it is shared by all users of the system on which it resides. Under Unix, the system-wide database directory is usually `/usr/database` or `/usr/local/database`; under MS-DOS or MS-Windows, it is usually `c:\database`.

Tape
: A database record.

Time
: In this guide, synonymous with sample number (q.v.). Thus the "time of an annotation" is the sample number of the sample to which the annotation "points".

WFDB-compatible format
: Any of the standard formats readable and writable by the WFDB library, for storage of WFDB records in PhysioBank and on CDROMs.

WFDB library
: A set of functions (subroutines), able to read and write database files, callable by C and C++ programs, and described in this guide.

WFDB path
: The database path (q.v.).

Appendix B Installing the WFDB Software Package

This appendix briefly describes how to install the WFDB Software Package on a new system. The package includes C-language sources for the WFDB library and for a variety of applications (see Appendix C [WFDB Applications], page 117) including `WAVE`, sources for this manual, the *WFDB Applications Guide*, and the *WAVE User's Guide*, and a one-minute sample record ('`100s`').

The latest version of the WFDB Software Package can always be downloaded in source form from `http://physionet.org/physiotools/wfdb.shtml`, the WFDB home page on PhysioNet. Binaries for popular operating systems and development snapshots are also usually available there.

The process for installing the package is the same on all platforms, and is documented in detail in the quick-start guides for the popular platforms that can be found on the WFDB home page. In brief:

1. *Install any prerequisites needed for your platform.* These include `gcc` (the GNU Compiler Collection), related software development tools such as `make`, the `libcurl` library (if NETFILES support is desired), the XView libraries (needed for WAVE only), and X11 (needed by XView). All of these components are free (open-source) software available for all popular platforms, including GNU/Linux, Mac OS X, MS Windows, and Unix. The quick start guides list recommended packages and where to find them.
2. *Download and unpack the WFDB Software Package.* Versions for all platforms are built from a single package of portable sources; the most recent package is always available at `http://physionet.org/physiotools/wfdb.tar.gz`.
3. *Configure the package for your system.* The `configure` script creates a customized building procedure for your system and allows you a few choices about where to install the package.
4. *Make and verify a test build.* The package includes a set of test scripts that are run to verify basic operations of the WFDB library and many of the applications, permitting them to be tested before installation.
5. *Make, install, and test a final build.*

See the quick start guide for your platform for detailed step-by-step instructions.

Important: Although you may be able to compile the WFDB Software Package using a proprietary compiler, this is *not supported*.

The WFDB library and languages other than C

If you wish to use or develop WFDB applications in C, C++ or Fortran, everything necessary is included in the WFDB Software Package.

The separate `wfdb-swig` package provides wrappers for the WFDB library so that it can be used by applications written in a variety of other languages supported by SWIG, the Simplified Wrapper Interface Generator. At the time of writing, the languages that are known to work with `wfdb-swig` wrappers are Perl, Python,, Java, and C#. Other languages supported by SWIG include Guile, mzScheme, PHP, Ruby, and Tcl. You will need to install `wfdb-swig` wrappers for your platform and language after installing the WFDB Software Package. Download the `wfdb-swig` package from `http://physionet.`

`org/physiotools/wfdb-swig.shtml`, the `wfdb-swig` home page on PhysioNet, and build and install the wrappers you need following the instructions on that page.

The WFDB Toolbox for Matlab provides a set of WFDB applications for Matlab, based on the SWIG Java wrappers; install it with one click from the link you will find on its home page, `http://physionet.org/physiotools/matlab/wfdb-swig-matlab/`.

Appendix C WFDB Application Programs

This appendix briefly describes the application programs that are included with the WFDB Software Package. Except where noted otherwise, these applications are usable on all systems for which the WFDB library is available. For details on using these programs, refer to the *WFDB Applications Guide*. (On Unix systems, the contents of the *Applications Guide* may also be available as on-line `man` pages.)

How to use these programs

These programs are kept in directories that vary from system to system; they may not be in the default search path. If you cannot find them, consult an expert (such as the person who installed the WFDB library on your system). If you use these programs often, you may wish to include the directory in which they are kept in your search path.

To use any of these programs, you will need to set the database path first (see Section 1.4 [WFDB path], page 12), unless the default database path ('`. /usr/database http://physionet.org/physiobank/database`') is suitable. Programs that accept *time* arguments or commands (usually shown as *from* and *to* below) use `strtim` to convert these strings into sample intervals; hence they accept any of the varieties of standard time format described earlier (see [`timstr` and `strtim`], page 36). Programs that accept annotation mnemonics as arguments or commands (usually shown as *code* below) use `strann` to interpret them; for a list of legal mnemonics, see Chapter 4 [Annotation Codes], page 67. Where record or annotator names are required as command arguments, they are indicated below as *record* or *annotator*.

In the remainder of this appendix, you will find usage examples and capsule descriptions of the standard WFDB application programs. The square brackets ('*[ ]*') in some of the usage examples surround arguments that may be omitted; the brackets themselves are not to be included in the command line. Where an ellipsis ('*...*') appears, it indicates that the previous argument may be repeated. If invoked without any arguments, or with a '`-h`' (help) option, most of these programs print a brief synopsis of how they are used.

Annotation File Processing

```
ann2rr -a annotator -r record [ options ... ]
rr2ann -a annotator -r record [ options ... ]
rdann -a annotator -r record [ -f from -t to -p type ... ]
wrann -a annotator -r record
sumann -a annotator -r record
tach -a annotator -r record [ options ... ]
```

Programs `ann2rr` and `rr2ann` respectively list RR (inter-beat) intervals in text format from an annotation file, and create an annotation file from a text-format list of RR intervals.

The program `rdann` is an annotation printer similar to the one shown in chapter 6 (see [Example 3], page 84). The optional *from* and *to* arguments (in standard time format) specify a portion of the annotation file to be printed, and one or more *type* arguments (annotation mnemonics) can be given to restrict the output to annotations that are of the specified type(s).

The output of `rdann` can be converted back into an annotation file by providing it as the standard input of `wrann`. This can be useful for editing annotation files in some cases; they can be converted to ASCII format by `rdann`, edited using any text editor, and converted back into annotation files by `wrann`.

A summary of the contents of an annotation file can be obtained using `sumann`. The summary includes the number of annotations of each type, and the duration and number of episodes of each rhythm and signal quality.

`tach` generates a uniformly sampled, smoothed, instantaneous heart rate sequence from an annotation file.

Evaluation of ECG Analyzers

```
bxb -r record -a reference-annotator test-annotator [ options ... ]
rxr -r record -a reference-annotator test-annotator [ options ... ]
mxm -r record -a reference-annotator test-annotator [ options ... ]
epic -r record -a reference-annotator test-annotator [ options ... ]
sumstats file
plotstm file
ecgeval
nst [ options ... ]
```

The motivation for developing the MIT and AHA databases was to provide material for evaluating the accuracy of arrhythmia detectors, particularly with respect to ventricular arrhythmias. Between 1984 and 1987, the Association for the Advancement of Medical Instrumentation (AAMI) sponsored the development of a recommended practice (designated ECAR) for using the databases for this purpose. The aim of ECAR was to specify the evaluation methodology in sufficient detail to permit reproducible testing, and to encourage informed comparisons of the performance of ventricular arrhythmia detectors in the analysis of these standard test recordings. More recently, the AAMI has developed, and ANSI has adopted as American National Standards, a standard (ANSI/AAMI EC38:1998) for ambulatory electrocardiographs, and a companion standard (ANSI/AAMI EC57:1998) for testing and reporting performance results of cardiac rhythm and ST segment measurement algorithms. EC38 and EC57 specify standard protocols for evaluating the automated analysis algorithms that are included in many such devices. These protocols include those developed for the earlier recommended practice, and extend them to evaluation of supraventricular arrhythmia and ischemia detection. EC38 and EC57 specify the use of `bxb`, `rxr`, `mxm`, and `epic` to perform evaluations, and further specifies the use of the MIT DB (as well as two other databases formerly available on the MIT-BIH Arrhythmia Database CDROM and now freely available on PhysioNet), the AHA DB, and (for devices that perform analysis of the ST segment) the ESC DB. If you are interested in this subject, obtain copies of the American National Standards for *Ambulatory Electrocardiographs* (ANSI/AAMI EC38:1998) and for *Testing and Reporting Performance Results of Cardiac Rhythm and ST Segment Measurements Algorithms* (ANSI/AAMI EC57:1998; see Appendix E [Sources], page 127).

To evaluate an arrhythmia detector using this software, obtain for each WFDB record to be used in the test an annotation file containing the detector's analysis of each beat. These are referred to as the 'test' annotation files (or the 'algorithm' annotation files, in EC38 and EC57). The placement of the beat annotations must match those in the reference

annotations within 150 msec; thus it is not necessary to place annotations precisely at the PQ junction (as in the AHA DB reference annotations) or on the major local extremum (as in the MIT DB reference annotations). If the detector is capable of shut-down (i.e., if it inhibits its QRS detection function during periods that it judges are unreadable), the test annotation files should include a `NOISE` annotation with `subtyp = -1` at the beginning of each period of shut-down, and a `NOISE` annotation with any other `subtyp` at the end of each such period. (If the record ends while the detector is shut down, the annotation file should include a final 'end of shut-down' annotation as above to permit correct shut-down accounting.) If the detector is capable of ventricular fibrillation detection, the test annotation files should also include `VFON` and `VFOFF` annotations; it is not necessary to mark flutter waves (use `FLWAV` annotations to do so if desired). See the `man` page for `epic`, in the *WFDB Applications Guide*, for information on marking atrial fibrillation, ischemic ST episodes, and ST deviation measurements in test annotation files. Any annotations that appear in the first five minutes of an annotation file are treated as belonging to the detector's learning period, and are not used in the evaluation. The evaluation software examines such annotations only to determine the detector's state (normal, shut down, or in VF) at the beginning of the test period.

Program `bxb` implements the beat-by-beat comparison algorithm described in EC38 (section 4.2.14.2.2) and EC57 (section 4.3.2). By default, the output is in a self-explanatory matrix format. The '`-L`' option, which must be followed by two file names, specifies that the output of `bxb` should be written in line format, for further processing by `sumstats`. The line-format output includes column headings only if the output file must be created from scratch. In this way, `bxb` can be used repeatedly to build up a line-format tables for multiple records. Among the other options is '`-o`', which causes `bxb` to generate an output annotation file (with annotator name `bxb`) indicating agreements and discrepancies between the input annotators.

`rxr` can be used to performed the run-by-run comparison described in EC38 (section 5.2.14) and in EC57 (sections 4.4.3 and 4.4.4). `mxm` compares heart rate, HRV, or other measurements, as described in EC38 (section 4.2.14.2.3). `epic` evaluates VF and AF detection, and ST analysis, as described in EC38 (sections 5.2.14), and EC57 (sections 4.5 and 4.6). These programs also accept a '`-L`' option to produce line-format output as for `bxb`.

`sumstats` derives the record-by-record, episode-by-episode, and aggregate performance statistics described in EC38 (section 4.2.14.3) and in EC57 (sections 3.5.2 and 3.5.3) from line-format output files produced by `bxb`, `rxr`, `mxm`, and `epic`. The input file must include the column headings so that `sumstats` can recognize the file type. The output includes a copy of the input, with aggregate statistics appended at the end. `plotstm` generates a PostScript scatter plot of ST measurement comparisons gathered by `epic`, as described in EC57 (section 4.6.2).

The easiest way to use these programs is to run `ecgeval`, which generates a script (batch) file to run `bxb`, `rxr`, etc., for each record in a database. See *Evaluating ECG Analyzers* (in the *WFDB Applications Guide*) for details.

By adding noise to annotated ECG records, the noise tolerance of an arrhythmia detector can be measured. This idea was described by the author, along with W.K. Muldrow and R.G. Mark, in "A noise stress test for arrhythmia detectors", *Computers in Cardiology* **11**:381-384 (1984). Program `nst` adds calibrated amounts of noise to ECGs (or other signals), generating an output record in WFDB format. `nst` was used to generate the graded

series of noisy ECG records in the MIT-BIH Noise Stress Test Database (see `http://physionet.org/physiobank/database/nstdb/`). These records are among those specified as standard test material by EC38 (section 4.2.14.2) and EC57 (section 3.2).

Signal Processing Applications

```
rdsamp -r record [ options ... ]
wrsamp -r record [ options ... ]
snip -i input-record -n new-record [ options ... ]
xform -i input-record [ options ... ]
fir [ options ... ] -c coefficient ...
sigamp -r record [ options ...]
sqrs -r record [ options ... ]
sample [ options ... ]
calsig -r record [ options ... ]
```

`rdsamp` prints samples from the specified record; '`-f`' and '`-t`' options may be used to specify a range of sample numbers, and a subset of signal numbers may be selected using the '`-s`' option. The output of `rdsamp`, or any similar text, can be converted into a WFDB record using `wrsamp`.

To copy an excerpt of a longer record, use `snip`, which creates new header and signal files for *new-record* in the current directory. The beginning and end of the excerpt are specified using '`-f`' and '`-t`' options as for `rdsamp`. Annotator names may follow a '`-a`' option; in this case excerpts from the specified annotation files are copied as well (the annotations are appropriately time-shifted).

`xform` is a more general version of `snip`; its main uses are for reformatting, rescaling, and sampling rate conversion. You may create a `hea` file specifying the desired format, sampling frequency, ADC zero levels, signal gains, etc., and supply it to `xform` using the '`-o`' option; if you do not do so, `xform` obtains the required information interactively. `xform` accepts all of the options used by `snip`, as well as several others.

Program `fir` is a general-purpose FIR filter for WFDB records, similar to the one discussed in chapter 6 (see [Example 7], page 89).

`sigamp` measures signal amplitudes (either baseline-corrected RMS amplitudes or peak-to-peak amplitudes); it may be useful for calibrating signals (together with `calsig`) or for determining signal gains for `nst`.

`sqrs` is a slightly modified version of the QRS detector discussed in chapter 6 (see [Example 10], page 98). Options allow specification of the signal and interval to be analyzed and the detection threshold.

Program `sample` is an MS-DOS application that uses a Microstar Laboratories DAP 1200- or 2400-series ISA (AT bus) analog interface board (see Appendix E [Sources], page 127) to generate database records from analog signals, or to generate analog signals from database records. If you wish to use other hardware for these purposes, refer to chapter 6 (see [Example 8], page 92) and to the source for `sample` as models.

If you create your own database records using `sample` or other means, program `calsig` may be useful for determining signal gains and offsets if your signals include standard calibration pulses or identifiable signal levels. `calsig` incorporates two independent algorithms for measuring calibration pulses; it rewrites header files based on its measurements.

Graphical Applications

```
wave -r record [ -a annotator ]
pschart [ [ options ... ] script ... ]
psfd [ [ options ... ] script ... ]
```

`wave` is an X Window System client application for viewing and editing WFDB records. (`wave` is included in the WFDB software package.) `wave` can be run on all popular platforms, including FreeBSD, GNU/Linux, Mac OS X, MS Windows, Solaris, and other systems for which X11 servers are available. Run `wave` without any arguments to obtain instructions for printing its on-line manual.

`pschart` and `psfd` produce annotated "chart recordings" and "full-disclosure" plots that can be printed on PostScript devices. These programs were used to prepare the *MIT-BIH Arrhythmia Database Directory* and the *European ST-T Database Directory*. The popular Chart-O-Matic web service (`http://physionet.org/cgi-bin/chart`) creates browser-viewable plots using `pschart`.

Appendix D Extensions

This section may be helpful if you wish to extend the capabilities of the WFDB library, or if you wish to port it to another environment. In order to make use of the information in this section, you should have the WFDB library sources (see Appendix E [Sources], page 127). The sources are distributed among four 'include' (`.h`) files and five `.c` files:

`wfdb.h`	Constant and structure definitions, and function prototypes
`ecgcodes.h`	Annotation codes
`ecgmap.h`	Annotation code mapping macros
`wfdblib.h`	External definitions for private WFDB library functions
`wfdbinit.c`	Functions `wfdbinit`, `wfdbquit`, and `wfdbflush`
`signal.c`	Functions for signals
`calib.c`	Functions for signal calibration
`annot.c`	Functions for annotations
`wfdbio.c`	Low-level I/O and operating system-dependent functions

The first three of these files are the standard 'include' files that are usually obtained by '`#include <wfdb/`*`file`*`.h>`' statements. When modifying the WFDB library, however, make any necessary changes in the copies of these files that are kept in the library source directory. Install the modified versions of the `.h` files in the system's `include` directory after installing the modified WFDB library.

The cleanest mechanism for adding additional fields to `hea` files is to include them in 'info' strings (see [`getinfo`], page 50), rather than by modifying the code that reads and writes `hea` files (in `signal.c`).

A common problem is the need to import signal files generated by other software. Often this problem can be solved by writing a format conversion program that uses input functions provided with the other software to read the signal files, and `putvec` to write them in one of the formats supported by the WFDB library. This solution is unlikely to be satisfactory if you have many large signal files to import, however, and you may wish to arrange for `getvec` to read the imported files directly. This may be done by defining a new signal file format, as outlined below.

To define a new format for signal files, choose a numeric code to represent your format. (Values between 900 and 999 are reserved for user-defined signal file format codes.) In `wfdb.h`, add your format code to `FMT_LIST` and increment `NFMTS`. In `signal.c`, define functions (macros if possible for efficiency) for reading and writing single samples; these should be named `r`*`nnn`* and `w`*`nnn`*, where *nnn* is your format code. Follow the examples in `signal.c`; it will almost certainly be easier to make use of the existing macros `r8` and `w8` than to begin from scratch. Add additional `case` statements in `getvec` and `putvec`, again following the existing models. You will also need to add a `case` in `isgsettime`, including a formula to determine the number of bytes needed per sample, given the number of signals multiplexed. (All currently-defined formats use fixed-length encoding. If you wish to implement variable-length encoding, it may be easiest to implement an indexed-search method for `isgsettime` in such cases.) If the ADC resolution exceeds the number of bits in a C `int` on your system, change the `typedef` for '`WFDB_Sample`' in `<wfdb/wfdb.h>` as necessary; be aware that this change is likely to require additional changes to application programs (use `lint` or an ANSI C compiler to check your code).

Although the WFDB library generally assumes that signal files are "pure", it is possible to read imported signal files that contain prologs (data that precede the first sample). To do so, you must construct a header file in which the `format` fields encode the length of the prolog in bytes (you can do this manually, or use `wfdbsetstart`, see [wfdbsetstart], page 53, for this purpose). For example, a signal file with a 512-byte prolog followed by format 16 samples would be specified using '`16+512`' in the `format` field or fields (if the file contains more than one signal, the `format` fields for all signals in the file must be identical). Note that this facility is provided only for signal file import; the WFDB library is not equipped to create signal files with embedded prologs.

In a similar fashion, though with substantially more effort in most cases, you may define a new format for annotation files. Add additional `stat` values for reading and writing to the list in `wfdb.h`. In `annot.c`, add additional `case` statements and code to `annopen`, `getann`, `putann`, and `wfdb_anclose`. If you are designing a new format, you may wish to specify a 'magic number' with which your files will begin, to allow `annopen` to recognize the format automatically; a good choice of such a number is one in which the first byte is non-zero (to distinguish it from AHA format files) and the high six bits of the second byte are zero (to distinguish it from WFDB format files).

Some users may wish to define additional annotation codes. An easy and portable way to accomplish this is to use `setannstr` and `setanndesc` within programs that create your annotation files, before opening them using `annopen` (or `wfdbinit`). Annotation files created in this way contain modification labels at the beginning that document the non-standard code definitions, and that permit them to be read properly by standard WFDB applications. Another solution is to modify the WFDB library. This method has the disadvantage that all of your applications that read annotation files must be recompiled, and they may no longer read standard annotation files properly. If despite this disadvantage you prefer to modify the WFDB library, begin by defining symbolic names and numeric values for your new codes in `ecgcodes.h`. (Values between 42 and 49 are reserved for user-defined annotation codes. Unused values less than 42 may be assigned in future versions of the WFDB library, and values greater than 49 are reserved to indicate the presence of optional fields such as `subtyp`.) Next, decide how the new codes are to be mapped by `isqrs`, `map1`, `map2`, `mamap`, and `annpos`, and set the appropriate entries in each of the code map arrays in `ecgmap.h`. Finally, add mnemonic and descriptive strings for the new codes in the `cstring`, `astring`, and `tstring` arrays in `annot.c`.

The modular design of the library makes it fairly easy to remove unneeded functionality in order to conserve memory for special applications. The `calib.c` package is not referenced by any other WFDB library modules. For signal processing applications that do not involve annotations, the entire `annot.c` package may be removed (with trivial modifications to the functions in `wfdbinit.c`). If you wish to add functions to the library, you will find that it will be easier to maintain your modified version and to merge updates if you preserve the existing arrangement of functions, which requires no global variables. Rather than defining global variables, consider implementing query functions (global-scope functions that read or write local variables). If you wish to define new types of binary files, consider using the low-level I/O routines in `wfdbio.c` for reading and writing them in a machine-independent format.

Porting the WFDB library to another environment is a straightforward operation if an ANSI C compiler is available in the target environment. Since all direct access to database

files is performed using the (private) function `wfdb_open`, it is possible to include file name translation in that function if needed, to accommodate file naming schemes that may be imposed by the operating system or other requirements. If the notion of environment variables is foreign to the target environment, `getwfdb` can be modified to read the WFDB path from a file. You may wish to modify the private function `wfdb_error` (which is responsible for all error reporting from WFDB library functions) if the 'standard error output' is unavailable or inadequate for use in the target environment. All of these functions are contained within `wfdbio.c`; it is unlikely that any other code will require changes for a port.

If you encounter errors while compiling `signal.c`, you may wish to try using the functions provided in that file as alternatives to the standard macros `r16` and `w16`; the fully-expanded versions of these macros are quite complex and are known to cause difficulty for at least one (now obsolete) C compiler. (Define the symbol '`BROKEN_CC`' while compiling `signal.c` in order to obtain the function versions of `r16` and `w16`.) While compiling `signal.c`, it may be necessary to disable code optimization for some C compilers; no current compilers are known to have such limitations, however.

Appendix E Sources

This section is a compendium of sources for databases and related materials that may be useful to readers of this guide. Please send any corrections to the author (`wfdb@physionet.org`).

WFDB Programmer's Guide (this guide)
WFDB Applications Guide
WAVE User's Guide
MIT-BIH Arrhythmia Database
MIT-BIH Arrhythmia Database Directory
MIT-BIH Polysomnographic Database
MIMIC Database
MGH/Marquette Foundation Waveform Database CROMs
Long-Term ST Database
Other reference databases of physiologic signals
WFDB Software Package
XView toolkit (needed for WAVE)

WWW: `http://physionet.org/`

PhysioNet offers free access via the web to large collections of recorded physiologic signals and related open-source software. The PhysioNet web site is a public service of the PhysioNet Resource funded by the National Institutes of Health's NIBIB and NIGMS. The master PhysioNet web server is located at MIT in Cambridge, Massachusetts; about ten public mirrors are located elsewhere in the US and around the world (see `http://physionet.org/mirrors/` for a list).

European ST-T Database CDROM
European ST-T Database Directory
VALE Database Directory

National Research Council (CNR) Institute of Clinical Physiology
Dept. of Bioengineering and Medical Informatics
via Trieste, 41
56126 PISA, Italy

email: `taddei@ifc.pi.cnr.it`
telephone: +39 050 501145
telefax: +39 050 503596

Over half of this database has been contributed to PhysioNet (see above), from which it may be downloaded freely.

AHA Database for Evaluation of Ventricular Arrhythmia Detectors

ECRI
5200 Butler Pike
Plymouth Meeting, PA 19462 USA

email: bduffin@ecri.org
WWW: http://www.ecri.org/
telephone: +1 610 825 6000

American National Standard ANSI/AAMI EC38:1998, Ambulatory Electrocardiographs
American National Standard ANSI/AAMI EC57:1998 Testing and Reporting Performance Results of Cardiac Rhythm and ST Segment Measurement Algorithms

Association for the Advancement of Medical Instrumentation
1110 N Glebe Road, Suite 220
Arlington, VA 22201 USA

WWW: http://www.aami.org/
telephone: +1 703 525 4890
telefax: +1 703 276 0793

Computers in Cardiology

WWW: http://www.cinc.org/

CinC is the major scientific meeting at which current research in ECG signal processing and modelling is discussed; the proceedings of the conference are probably the single best source of information in print about these topics. CinC conferences have taken place annually since 1974, usually in September. They are usually convened in North America and in Europe in alternate years. The deadline for submission of abstracts is on or about 1 May each year. Proceedings of CinC conferences since 2006 are available on-line at http://cinc.mit.edu/archives/, and usually appear about a month after the date of the conference. CinC will be in in Bologna (Italy) in 2008, and in Park City, Utah (USA) in 2009.

Since 2000, Computers in Cardiology and PhysioNet have jointly sponsored an annual series of open challenges that invite participants to address topics of interest to researchers and clinicians, often involving the study of WFDB-compatible data sets made available via PhysioNet, and culminating with research presentations and awards at CinC. See http://physionet.org/challenge/ for further information.

Proceedings of Computers in Cardiology (ISSN 0276-6574)

CinC proceedings from 2001 to present are available on-line at http://www.cinc.org/archives/.

IEEE members can find CinC proceedings from 1988 to the present using IEEExplore (http://ieeexplore.ieee.org/). Many universities provide access to these services for their students, faculty, and staff. Printed volumes of CinC proceedings are available from:

IEEE Customer Service
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331 USA

email: customer.service@ieee.org
WWW: http://www.ieee.org/ieeestore/
telephone: 1 800 678 IEEE (USA and Canada) or +1 732 981 0060
telefax: +1 732 981 9667

GNU emacs
gcc (the GNU portable C/C++ compiler)
ghostscript
GNU tar
GNU gzip (free and improved replacement for 'compress')
Larry Wall's 'patch' program, with GNU revisions
GNU groff, gtbl, and related text formatting utilities
GNU info and makeinfo (standalone hypertext browser and formatter)

Free Software Foundation
51 Franklin Street, Fifth Floor
Boston, MA 02110-1301 USA

email: gnu@gnu.org
WWW: http://www.gnu.org/
telephone: +1 617 542 5942

GNU software is included in all Linux distributions (in fact, since Linux is the name of the kernel only, and the largest component of these distributions is actually GNU software, it is proper to refer to them as GNU/Linux distributions). GNU software for all popular (and many other) operating systems is available on CDROM or tape from the address above, and is also freely available by anonymous FTP from ftp.gnu.org and and many other archive sites. Please support the FSF with a donation if you use GNU software.

TEX for Unix systems

This software is available by anonymous FTP from CTAN (Comprehensive TeX Archive Network) mirrors, including ftp.tex.ac.uk, ftp.dante.de, and ctan.tug.org. Many of the sources of GNU software (above) also make TEX, etc. available. CTAN is indexed on the World Wide Web (one such index is http://www.ctan.org/).

The Unix TEX distribution is also distributed on CDROM and in other tape formats by the Free Software Foundation (address above) and others. It is also included with most GNU/Linux distributions (see below).

Several commercial implementations of TEX for MS-DOS, MS-Windows, and Mac OS are widely available; visit the web site of the TEX Users Group (below) for pointers.

General information on TEX

TeX Users Group
PO Box 2311

Portland, OR 97208-2311 USA

WWW: `http://www.tug.org/`
email: `office@tug.org`
telephone: +1 503 223 3960
telefax: +1 503 223 9994

libcurl

WWW: `http://curl.haxx.se/`

The `libcurl` library is a modern and enhanced replacement for the `libwww` libraries (see below). It provides the low-level functions needed to support the WFDB library's (optional) NETFILES capability.

W3C libwww libraries

WWW: `http://www.w3.org/Library/`

The `libwww` libraries, created and maintained by the World Wide Web Consortium, were used in the original implementation of NETFILES. Since these libraries are no longer maintained, the WFDB library now uses `libcurl` instead.

X11 (the X Window System, Version 11)

email: `xorg_info@x.org`
WWW: `http://www.x.org/`
telephone: +1 781 376 8200
telefax: +1 781 376 9358

Sources for XView are available from PhysioNet.

GNU/Linux

GNU/Linux is a POSIX-compliant reimplementation of the Unix operating system, written by Linus Torvalds and a cast of thousands. It runs on Intel 386, 486, and Pentium PCs, among others. For information about GNU/Linux, visit the web site of the Linux Documentation Project:

WWW: `http://www.linuxdoc.org/`

GNU/Linux is freely available by anonymous FTP in source and binary form from many sites, including:

www.kernel.org
metalab.unc.edu
ftp.funet.fi

Many low-cost (typically US$10 to US$30) distributions of GNU/Linux on CDROMs are widely available. Among the more popular are:

Debian (non-commercial)

WWW: `http://www.debian.org/`

Fedora (non-commercial)

WWW: `http://fedoraproject.org/`

Gentoo (non-commercial)

WWW: `http://www.gentoo.org/`

Knoppix (non-commercial, live CD)

WWW: `http://www.knoppix.org/`

Mandriva

WWW: `http://www.mandriva.com/`

Red Hat

WWW: `http://www.redhat.com/`

Slackware (non-commercial)

WWW: `http://www.slackware.com/`

SuSE (Novell)

WWW: `http://www.novell.com/linux/`

Ubuntu

WWW: `http://www.ubuntu.com/`

Compilers and software development systems

Any ANSI/ISO C compiler can be used to compile the WFDB library and applications that use it. Under Unix and GNU/Linux, high-quality free compilers and development tools are universally available and taken for granted. Even if you must work in the MS-Windows or MS-DOS environment, however, there

is no reason to purchase expensive, bloated, and inferior proprietary compilers and software development systems, since there are several excellent, highly recommended, and completely free alternatives based on the GNU C/C++ compiler (gcc).

Using any of these packages does not limit you to creating free (open source) software, although you are certainly encouraged to do so. If you wish to develop and sell proprietary software using `gcc`, this is certainly possible, with fewer and less severe restrictions than you will encounter if using a commercial compiler.

Cygwin

WWW: `http://www.cygwin.com/`

This is a freely available software development platform for MS-Windows 9x/NT/2000/ME/XP, based on GNU `gcc` and a large set of Unix utilities developed by the GNU project and ported to MS-Windows by Cygnus Software (now part of Red Hat, see above). Cygwin itself is open-source software and is highly recommended in preference to commercial C/C++ compilers if you must work in the MS-Windows environment. The WFDB Software Package binaries available via PhysioNet are created using Cygwin.

MinGW

WWW: `http://www.mingw.org/`

This is another freely available software development platform for MS-Windows, also based on `gcc` and many of the same utilities as Cygwin. The difference is that applications built by MinGW gcc use the native MS-Windows C library, while those built by Cygwin gcc generally use Cygwin's Unix-compatible standard C library.

djgpp

WWW: `http://www.delorie.com/djgpp/`

A freely available software development platform for MS-DOS, including `gcc`, a free 32-bit DOS extender, and many of the same utilties as Cygwin and MinGW.

Microstar DAP analog interface boards for PCs

Microstar Laboratories
2265 116th Avenue N.E.
Bellevue, WA 98004 USA

email: `info@mstarlabs.com/`
WWW: `http://www.mstarlabs.com/`
telephone: +1 425 453 2345
telefax: +1 425 453 3199

Web browsers

The most popular Web browsers may be downloaded at these locations:

Firefox
FTP: `ftp.mozilla.org`

WWW: `http://www.mozilla.org/firefox/`

Google Chrome
WWW: `http://www.google.com/chrome/`

MS Internet Explorer
FTP: `ftp.microsoft.com`
WWW: `http://www.microsoft.com/`

Opera
FTP: `ftp.opera.com`
WWW: `http://www.opera.com/`

Answers to Selected Exercises

3. 280 adu = (280 adu −(−300 adu)) / 10 adu/mmHg = 58 mmHg.

4. 120 mmHg = 120 mmHg * 10 adu/mmHg + (−300 adu) = 900 adu.

5. The range of sample values is −2047 to +2047 adu, or −174.7 to +234.7 mmHg. The special value −2048 adu, if found in the input, is replaced with `WFDB_INVALID_SAMPLE` (-32768) to indicate missing or out-of-range samples.

6. We don't know how big `signal.dat` is, because we don't know how many other signals are multiplexed with the `BP` signal. If there are no others, `signal.dat` is 1,500,000 bytes (`nsamp` * 1.5 bytes/sample). One-third of the space occupied by `signal.dat` could be saved if it were converted to format 8. The maximum slew rate representable in format 8 is 127 adu/sample interval * 100 sample intervals/sec / 10 adu/mmHg = 1270 mmHg/sec.

7. One way to save a little space is to resample the signal at 120 Hz, and then change to format 8 (maximum slew rate = 1524 mmHg/sec). This can be done using `xform`; it reduces the storage requirement by one-fifth.

8. If you have installed PhysioToolkit's `plt` package, a simple solution is to write the sample numbers and values on the standard output in two-column ASCII format. The plotting is then performed by the pipeline:

```
your-program | plt 0 1
```

Recent Changes

This section contains a brief summary of changes to the WFDB library and to this guide since the first printing of the tenth edition of this guide in June, 1999. See `NEWS`, in the top-level directory of the WFDB Software Package distribution, for information on any more recent changes that may not be described here.

WFDB 10.7

New features in version 10.7

The WFDB library now supports storing signals in compressed form, using the FLAC (Free Lossless Audio Codec) algorithm. Compressed signal files are designated by the the format code 508, 516, or 524 (depending on the signal resolution.) In order to read and write signals in these formats, the FLAC library and header files must be installed when compiling the WFDB Software Package.

If the symbol `WFDB_LARGETIME` is defined before including `wfdb.h`, then the type `WFDB_Time` is defined as a `long long` rather than a `long` (see Section 3.9 [Large time values], page 66).

New macros, defined in `wfdb.h`, can be used to construct format strings for `printf()` (see Section 3.7 [Displaying numeric values], page 64) and `scanf()` (see Section 3.8 [Parsing numeric values], page 65), independent of the underlying data types.

If high-resolution mode is enabled using `setgvmode()`, then `sampfreq(`*r*`)` will return the highest sampling frequency of any signal in record *r*, and subsequent calls to `mstimstr()` and `strtim()` will likewise interpret sample and frame numbers just as if the record were actually opened using `isigopen()`.

If an error occurs while writing an output file, the library usually cannot detect the error until the file is closed. (For example, after calling `putvec()`, the output samples will be buffered, but there might not be enough disk space to store them.) In order to check that the output was written successfully, an application can call '`osigfopen(NULL, 0)`' (for signal and header files), '`annopen("", NULL, 0)`' (for annotation files), or '`setinfo(NULL)`' (for info files). These function calls also work with previous versions of the library, but would always return 0.

In calibration files, the signal description '`*`' is treated as a wildcard entry that will match any signal with the specified units.

When reading a multi-frequency record using `getvec()` or `sample()`, the resampled values will be correctly rounded to the nearest integer, and will be calculated correctly for large numbers of samples per frame.

The Fortran wrapper functions have improved support for working with "native Fortran" string data (when `FIXSTRINGS` is defined), allowing Fortran programs to work with strings containing embedded spaces. These functions will now avoid buffer overflows, both with and without `FIXSTRINGS`.

The macros `SALLOC`, `SUALLOC`, `SREALLOC`, `SSTRCPY`, and `MEMERR` will avoid evaluating their arguments more than once (except for the pointer argument.) These macros will now handle zero-byte allocations consistently, and will check for integer overflows.

`osigopen()` permits the *siarray* argument to be `NULL` (like `isigopen()`.)

Bugs fixed in version 10.7.0 (10 June 2022)

A number of corner cases have been fixed in `getframe()`. It now correctly handles skewed signals in variable-layout records, records with multiple signal files, and records with signal file prologs. It also correctly handles multi-segment records with signal file prologs, and multi-segment records where some signal files are missing. The `wfdbgetskew()` and `wfdbsetiskew()` functions now work as documented.

Functions for reading header, calibration, and info files will now correctly handle lines of any length, rather than being limited to 255 or 126 characters per line.

In the WFDB path, colons can be used within the "authority" portion of a URL, in order to designate a port number ('`http://example.org:8080/`'), an IPv6 address ('`http://[::1]/`'), or a password ('`http://x:y@example.org/`'). Colons that appear after the third slash in a URL are treated as path separators, as usual.

Many library functions accept an argument that is a pointer to a string, array, or structure of some kind. If the object pointed to is not modified, the argument is now declared with the `const` qualifier.

Previously, the `putvec()` function would sometimes modify the array provided by the caller; now, the array is never modified.

The `sample()` function will return correct results when samples are retrieved in random order. Previously, if the caller requested sample 1, followed by sample 5000, followed by sample 2, `sample()` would return sample 1, followed by sample 4998, followed by sample 5001.

A multi-segment record in which the first segment has non-zero length, but the first segment *header* file is missing length or checksum fields, is now handled correctly as a fixed-layout record.

`setgvmode()` will handle `WFDB_GVPAD`, as documented; this was broken in WFDB library versions 10.5.3 through 10.6.2.

`getwfdb()` will no longer add or modify existing environment variables. In previous versions of the library, the first time `getwfdb()` was called, it would set the `WFDB`, `WFDBCAL`, `WFDBANNSORT`, and `WFDBGVMODE` environment variables, if these variables were not already defined.

`setwfdb()` will correctly handle indirect paths (e.g., '`@file`').

`timstr()` or `mstimstr()` would, in some cases, return an incorrect date following a call to `setbasetime()` or `datstr()`. This has been fixed.

WFDB 10.6

Changes in version 10.6.2 (8 March 2019)

Changes to the internal functions `get_ann_table()` and `put_ann_table()` ensure sensible and consistent behavior if a custom annotation type is defined (using `setannstr()`) but no description is provided (using `setanndesc()`). Prior to version 10.6.0, the library would typically set the description to `"(null)"` in this case, but this was not guaranteed. In version 10.6.0, this behavior was changed in an attempt to fix the undefined behavior, but the result was an annotation file that could not be read correctly. Version 10.6.2 fixes both

of these problems; annotation files written by any older version can be read by version 10.6.2, and annotation files written by version 10.6.2 can be read by any older version.

Changes in `isgsettime()` and `isgsetframe()` avoid incorrect behavior if the specified time value is large enough to cause integer overflow.

Changes in version 10.6.1 (28 November 2018)

The internal function `edfparse()` will correctly calculate the ADC resolution of EDF signals, and will correctly interpret a negative "number of data records" (meaning that the length of the record was unknown at the time the header was written.)

`sample()` now returns `WFDB_INVALID_SAMPLE` if the requested sample is not in the buffer and cannot be read (i.e., `isigsettime()` fails.) `sample_valid()` returns 0 in this case. In previous versions of the library, `sample()` would call `exit()` in this situation, forcing the program to exit immediately.

In some circumstances, `isigopen()` will now return -3; unlike a return value of -1 or -2, this indicates that no new signals have been opened and all previously-opened signals have been closed.

The implementation of NETFILES has been optimized in several ways, to avoid making unnecessary HTTP requests, and to avoid unnecessarily disconnecting from and reconnecting to a remote server, especially when opening a remote file for the first time.

The library will now honor HTTP redirections. (Previously, the `libwww`-based implementation would follow redirections properly, but the `libcurl`-based implementation would not.) Up to five redirections will be followed for a particular file, and (if range requests are used) the new URL will be cached for up to five minutes.

The internal function `readheader()` will correctly parse "minimum version" requirements (denoted by `#wfdb` at the start of a header file.)

The library will cope better with records that are incorrectly formatted or partially unreadable: `getskewedframe()` will avoid crashing if a signal file ends unexpectedly in a multi-segment record; `readheader()` will reject segment names beginning with '`+`'; `isigopen()` will reject records or segments where the total number of samples per frame, or the maximum signal skew, is too large; `isgsetframe()` and `getskewedframe()` will treat an unreadable segment as an error.

Several bugs have been fixed in `isigopen()`. It will now correctly handle the sitation where `nsig` is zero or negative, the record has multiple segments, and the first segment header cannot be read or contains no signals. It will correctly handle some situations where a record contains three or more signal groups, and not all signals can be opened (for example, when the first signal file contains more than `nsig` signals, but the second and third signal files together contain `nsig` or fewer signals.) When multiple records are opened at once ("`+` mode"), it will correctly honor the limit provided by the caller, and will not open more than `nsig` new signals.

The internal function `wfdb_fopen()` will treat any path containing `://` as a possible URL, and will correctly handle non-URL paths that end with `:` or `:/`.

Changes in version 10.6.0 (26 January 2018)

The new functions `getiafreq()` and `setiafreq()` allow the application to change the time scale for input annotations, as `setifreq()` does for input signals. The new function `getiaorigfreq()` returns the "native" time resolution of the input file.

New macros, defined in `wfdb.h`, can be used to determine the limits of the numeric types used by the WFDB library. For example, `WFDB_SAMPLE_MAX` is the maximum value of a `WFDB_Sample` variable.

`wfdb.h` now includes prototypes for the internal functions `wfdb_me_fatal()` and `wfdb_error()`, which are used by the `MEMERR` macro.

Support for using `libwww` to read remote files has been removed. `libwww` support was introduced with version 10.0.1 in 2000, and was still supported as an alternative after `libcurl` support was added in version 10.3.16. However, `libwww` has not been actively developed for many years; we don't recommend its use anymore, and removing support is necessary in order to simplify and add new features to WFDB in the future.

The WFDB library now supports reading variable-layout, *multi-frequency* records, provided that each signal's sampling frequency is constant across the entire record. Although previous versions of the library were able to read such records in some cases, the behavior of `getvec()` and `isigsettime()` was buggy and inconsistent.

When reading a variable-layout multi-segment record, values returned by `getvec()`, `getframe()`, or `sample()` are rounded to the nearest integer. If that value is outside the range of a `WFDB_Sample` variable, the value `WFDB_SAMPLE_MIN` or `WFDB_SAMPLE_MAX` is returned.

When reading a signal from an EDF file in which the minimum representable value is greater than zero (corresponding to a negative baseline), the `baseline` returned by `isigopen()` is correctly rounded to the nearest integer. In previous versions, the baseline was off by 1, causing applications to calculate an incorrect physical value.

When reading a signal file in format 311, the file may end with two samples encoded as three bytes (so the total number of samples is $3n+2$, and the total number of bytes is $4n+3$.) Files created by the WFDB library itself do not use this format (an extra zero sample will be added in this case, for backwards compatibility), but such files may be created by other applications.

`isigsettime()` works correctly when reading a multi-segment record with multiple signal files per segment.

`setifreq()` works correctly if there are no input signals open.

`tnextvec()` returns -1 if it reaches the end of the record without finding a valid sample. Previously, in the case of a fixed-layout record, it would return zero in this case.

`getann()` correctly handles annotation files with huge `time` values (where the unscaled value exceeds the range of a `WFDB_Time` variable.) If the *scaled* value exceeds that range, it is replaced with `WFDB_TIME_MIN` or `WFDB_TIME_MAX`. `getann()` will also correctly round the `time` value if it is negative, and will correctly handle non-NOTE annotations at time *zero*.

If an input annotation file has no explicit time resolution, but the application calls `setifreq()` before `annopen()`, the input annotations will be rescaled accordingly. (In previous versions of the library, this would only work for annotation files with an explicit time resolution.)

When writing signals in format 212, 310, or 311, if the total number of samples is not a multiple of 2 or 3, `wfdbquit()` will correctly write out the remaining samples, adding padding if necessary. `wfdbflush()` will do likewise, provided that the output is a regular file and there is no mandatory block size.

When writing an annotation file, the time resolution and annotation type definitions will not be written until the first time `putann()` is called for that annotator. As a result, it is possible to call `setafreq()`, `setannstr()`, or `setanndesc()`, after calling `annopen()` and before the first `putann()`.

`putann()` correctly handles consecutive annotations that are more than 2,147,483,647 samples apart.

The Fortran wrapper functions have been updated for compatibility with modern Fortran compilers on 64-bit systems.

Password-protected remote files can now be accessed when WFDB_PAGESIZE is set to zero.

WFDB 10.5

Changes in version 10.5.24 (28 May 2015)

The environment variable `WFDBPASSWORD` is used for user/password information, in place of the former (inflexible and insecurely implemented) `PNWUSER` and `PNWPASS` variables.

The environment variable `CURL_CA_BUNDLE` defines the set of certificate authorities that are trusted to issue certificates for web servers, if any `https://` entries are used in the WFDB path.

If the environment variable `WFDB_NET_DEBUG` is set, then whenever the WFDB library requests or receives data from a remote server, details of the operation will be written to the standard error output.

On most platforms, the library is now installed in `/usr/local/lib` by default, rather than `/usr/local/lib64` (as in 10.5.23) or `/usr/lib64` (as in previous versions.)

Changes in version 10.5.23 (13 March 2014)

Changes in `configure`, `Makefile.tpl`, `conf/linux.def`, and `conf/linux-slib.def` simplify installation of shared WFDB libraries and the applications that use them on Linux platforms.

(WFDB library version 10.5.22 was identical to version 10.5.21.)

Changes in version 10.5.21 (18 November 2013)

In previous releases, WFDB library function `strtim()` did not always handle bracketed string inputs properly. Thanks to Benjamin Moody, who reported the problem and provided a patch to fix it.

Changes in version 10.5.20 (2 September 2013)

Absolute pathnames are not tested in `wfdb_open()` unless the WFDB path contains an empty component.

Changes in version 10.5.19 (21 July 2013)

This release includes fixes in lib/signal.c for several bugs that sometimes caused `findsig()`, `getvec()`, and `sample()` to return incorrect values when reading variable-layout multi-segment records with missing signals.

Changes in version 10.5.18 (16 February 2013)

Function `wfdb_addtopath` now works properly if the path contained only one component on entry.

Changes in version 10.5.17 (2 January 2013)

The WFDB library internal function `wfdb_addtopath()` (in `lib/wfdbio.c`) adds the path component of its string argument (i.e., everything except the file name itself) to the WFDB path, inserting it there if it is not already in the path. This function is called by another WFDB library internal function, `wfdb_open()`, which finds and opens files for all other WFDB library functions that open files; `wfdb_open()` passes the paths of files it successfully opens to `wfdb_addtopath()`, permitting other files in the same locations to be found. In previous releases, `wfdb_addtopath()` appended the new path component to the end of the WFDB path. As originally noted by David Brooks, this is suboptimal since the files comprising a given record are most often kept in the same directory. In this release, the new component is inserted at the head of the path, or as the second path component if `"."` (the current directory) is the first component, thus improving the likelihood that subsequent files to be opened will be found in the first or second location `wfdb_open()` checks.

Changes in version 10.5.16 (27 September 2012)

A bug in WFDB library versions 10.5.11 through 10.5.15 resulted in an attempt to close an already-closed header file after invoking `putinfo()`. Thanks to Benjamin Moody for identifying the bug and contributing code to correct it (in `lib/signal.c` and `lib/wfdbinit.c`).

Changes in version 10.5.15 (25 September 2012)

Changes to the internal function `readheader()` in WFDB library version 10.5.14 made the library unable to open EDF files. This bug has been fixed.

Changes in version 10.5.14 (13 August 2012)

WFDB applications can now read shared and private PhysioNetWorks projects securely, just as they have been able to read PhysioBank data since version 10.0.1 (November 1999). Low-level functions `wfdb_open()` (in `lib/wfdbio.c`) and `readheader()` (in `lib/signal.c`) incorporate changes to implement this new capability, as well as another new feature that allows record names to be specified using absolute pathnames or URLs. As always, a final `.hea` is not considered to be part of a record name, but it may be included or omitted as desired.

Changes in version 10.5.13 (13 May 2012)

Versions of the WFDB library up to 10.5.10 ignored embedded empty lines within the `info` sections of `.hea` files, but versions 10.5.10 and 10.5.11 treat them as markers of the end of the `info` section. This version restores the previous treatment of embedded empty lines. Thanks to Justin Leo Cheang Loong for reporting this issue.

Changes in version 10.5.12 (25 April 2012)

When called with a NULL argument, `getinfo()` sometimes behaves differently in WFDB library version 10.5.11 than it does in previous versions. This release restores the previous behavior. Thanks to Benjamin Moody for reporting this issue.

Changes in version 10.5.11 (6 April 2012)

This release of the WFDB library introduces support for `.info` files. These are files containing `info` strings in the same format as those that can be stored in `.hea` files. The `.info` file for a record named `record` is named `record.info`, and it may be located anywhere in the WFDB path. Function `getinfo()`, as in previous releases, returns the first info string belonging to the record named as its argument, or the next info string belonging to the previously specified record if its argument is NULL. Beginning with this release, successive calls to `getinfo()` return the next info string contained in the record's `.info` file if it exists, and if there are no more in the record's `.hea` file. Function `putinfo()` writes an info string to the currently open output `.hea` file, unless function `setinfo()` (new in this release) has been invoked, in which case the info string is appended to the record's `.info` file in the current directory. `getinfo()` reads all of the record's info strings the first time it is invoked, returning them one at a time; `wfdb_freeinfo()` frees the memory allocated for `getinfo()`'s info strings and closes the `.info` file opened by `putinfo()`, if any. After invoking `wfdb_freeinfo()`, a subsequent call to `getinfo()` reads the info strings again (or those of a different record, if a new record has been opened).

Virginia Faro-Maza identified and corrected a bug in WFDB library function `iannsettime()`, in `lib/annot.c`, that caused some annotations to be missed when two or more annotation files are open simultaneously.

WFDB library function `isigopen()`, in `lib/signal.c`, was reverted to that of version 10.5.9.

Benjamin Moody contributed patches for `app/snip.c` to ensure that the output will be written using a format that can accommodate the sample range.

The signal calibration file, `data/wfdbcal`, has been updated with new definitions.

Changes in version 10.5.10 (15 November 2011)

The WFDB library function `isigopen()`, in `lib/signal.c`, searches each component of the WFDB path for the signal file(s) named in the associated header file, until a match is found. Since signal files are usually located in the same directories as header files, they can be located most quickly by looking first in those directories. Thanks to David Brooks for suggesting this optimization and for a sample implementation.

Attempts to set the default size for HTTP range requests (using the environment variable `WFDB_PAGESIZE`) were ignored in previous versions of the WFDB library when compiled with libcurl. Thanks again to David for noting this limitation, which has been eliminated in this release by a change in `www_init()` (in `lib/wfdbio.c`).

Changes in version 10.5.9 (10 September 2011)

When an application passes an array containing `WFDB_INVALID_SAMPLE` values to `putvec()`, the function translates these into the corresponding invalid-sample sentinel values used by the file format. (This is the inverse of the transformation done by `getvec()` and

`getframe()`, so the effect is that at the application level, invalid samples are always represented by the value `WFDB_INVALID_SAMPLE`.)

Previous versions of the library did not perform this transformation correctly for formats 80 and 160 (they used a value of zero rather than -128 or -32768, respectively.) This bug was mainly an issue for programs such as `snip` and `xform` that read and modify existing data files.

Thanks to Benjamin Moody for identifying these problems and providing patches to remedy them.

Changes in version 10.5.8 (12 March 2011)

Previous versions of the WFDB library did not behave properly if `setbasetime()` was invoked before `setsampfreq()`. Also, when dealing with time-of-day strings, previous versions of `mstimstr()` and `strtim()` round the base time to a number of samples since midnight, and do not work correctly if the number of samples per day is not an integer.

Benjamin Moody contributed patches to `lib/signal.c` that remedy these deficiencies. The patches include three new internal functions (`fstrtim()`, `ftimstr()`, and `fmstimstr()`) which are equivalent to the WFDB library functions `strtim()`, `timstr()`, and `mstimstr()`, but take a second argument specifying the sampling frequency. These internal functions are used by `setbasetime()` and `setheader()` to record the base time with millisecond precision, independent of the actual sampling frequency, and independent of the effects of `setifreq()`, if any. Moreover, `mstimstr()` returns a string representation of the base time plus the given number of sampling intervals, `mstimstr(0)` returns the exact base time, and `strtim()` returns the sample number that is closest to the given time. In addition to being more precise, both functions now work correctly even if the number of samples per day is not an integer. Applications using this version of the WFDB library may call `setbasetime()` and `setsampfreq()` in either order.

Changes in version 10.5.7 (16 December 2010)

When opening records with the same name in different directories successively within a single process, the persistence of WFDB path changes made by WFDB library function `wfdb_addtopath()` interfered with locating the correct files in the second and subsequent records. The solution included addition of a new WFDB library function, `resetwfdb()`, which restores the WFDB path to the value returned by the first invocation of `getwfdb()` in the current process (or `NULL` if `getwfdb()` has not been invoked); library function `wfdbquit()` now invokes `resetwfdb()`. In addition, the safe-string copy macro `SSTRCPY` (defined in `wfdb.h`) now properly handles the case of copying null pointers. Thanks to Benjamin Moody for identifying the problem, providing test inputs, and contributions to the solution.

Changes in version 10.5.6 (29 November 2010)

WFDB records with names of the form `nnn/nnn` can now be identified using the short form `nnn/` in applications built using the WFDB library (e.g., `rdsamp -r mimicdb/037/` and `rdsamp -r mimicdb/037/037` are now equivalent).

In `<wfdb.h>`, the maximum lengths of record names, units strings, and signal description (`.desc`) strings have been increased (to 50, 50, and 100 characters respectively). In `lib/wfdbio.c`, the maximum length of a WFDB file name (including path information) has been increased to 1024 characters.

(WFDB library version 10.5.5 was identical to version 10.5.4.)

Changes in version 10.5.4 (13 July 2010)

Function `getseginfo()` has been introduced in WFDB library version 10.5.4, to allow applications to obtain information about the segments belonging to the current (multi-segment) input record.

In previous versions, integer arithmetic overflow was possible when converting format 32 samples using `aduphys()`, if the difference between the baseline value and the sample to be converted exceeded the range of signed 32-bit integers. Although `rdsamp` does not use `aduphys()` similar code in `rdsamp` also exhibited this problem, which has now been corrected; thanks to Ikaro Silva for reporting it and providing a test case.

Changes in version 10.5.3 (22 June 2010)

Function `getgvmode()` has been introduced in WFDB library version 10.5.3, to allow querying the current operating mode of `getvec()`.

Changes in version 10.5.2 (18 April 2010)

When reading annotations of multifrequency records opened in high-resolution mode, a time shift was introduced by `getann()` in WFDB library versions 10.4.5 to 10.5.1. This problem has been corrected in 10.5.2.

Certain malformed segment .hea files were able to cause null pointer errors in `isigopen()`. This problem has been corrected. Thanks to Mauro Villarroel for reporting this problem together with a test case.

Changes in version 10.5.1 (19 March 2010)

In version 10.5.0, signals in formats 80 and 160 with amplitudes of 0 were incorrectly treated as invalid by `getframe()` and `getvec()`. Thanks to Isaac Henry for reporting this problem, which has been corrected in `lib/signal.c`.

Changes in version 10.5.0 (16 March 2010)

The WFDB library now supports signals with 24 and 32 bits of precision, using new formats 24 and 32, as well as BDF and BDF+ files (24-bit EDF and EDF+ variants), so that WFDB applications can now read these formats. Note that these formats, unlike all previously defined formats, require more than 16 bits per sample value. If the WFDB software is compiled on a 16-bit platform (unusual except for embedded processors), the excess high bits of signals in these formats are not read on input, and they are replaced by zeroes on output, unless the `WFDB_Sample` data type has been redefined as `long` (in `wfdb.h`). This is not done by default since it would increase memory and computational requirements unnecessarily in embedded applications that do not require 24- or 32-bit precision.

Since support for extended precision samples cannot be introduced without this limitation in backward compatibility for 16-bit platforms, the minor version number of the library has been incremented to 5. Most users will not be affected by this change, however, apart from the new functionality it provides.

Memory allocation macros that have been defined previously in `wfdblib.h` have been moved to `wfdb.h` so they are accessible to WFDB applications, including user-written appli-

cations. For information about using these macros (MEMERR, SFREE, SUALLOC, SALLOC, SREALLOC, and SSTRCPY), see Section 2.9 [memory allocation macros], page 54.

A buffer overflow in the WFDB library's internal function `edfparse()` (in `signal.c`) has been corrected, thanks to a bug report and patch from Joonas Paalasmaa.

WFDB 10.4

Changes in version 10.4.25 (21 January 2010)

In WFDB library versions 10.4.5 through 10.4.24, `strtim("e")` did not work properly when the open input record was an EDF file and high resolution input mode had not been selected. This problem has been corrected.

(WFDB library version 10.4.24 was identical to version 10.4.23.)

Changes in version 10.4.23 (7 August 2009)

Several changes in `lib/signal.c` eliminate unintended interactions that occurred in previous versions between the input modes that can be selected using `setgvmode()`. The effects of these interactions were first observed in newly written code; existing WFDB applications were apparently unaffected.

On Cygwin/Windows, the WFDB library is now installed in both `/usr/bin` and `/usr/lib`, to simplify building user applications that use the WFDB library and to simplify running compiled WFDB applications outside of the Cygwin environment.

Changes in version 10.4.22 (28 July 2009)

When reading multifrequency records in WFDB_LOWRES mode, invalid sample values occurring in signals that need to be decimated were not always handled properly in WFDB library function `getvec()`, resulting in spurious artifacts during intervals that contained a mixture of valid and invalid samples. This bug has been corrected; thanks to Omar Abdala for a report that brought this problem to light.

A declaration for WFDB library function `findsig()` has been added to `wfdb.h`. Thanks to Thomas Heldt for reporting a warning that resulted from its omission in previous versions.

Changes in version 10.4.21 (14 May 2009)

Changes in `lib/Makefile.tpl` were needed in order to pass configuration constants to the functions added to `lib/wfdbio.c` in 10.4.20, but were omitted from that release; they have been included in this version.

Changes in version 10.4.20 (4 May 2009)

Several new functions have been added to `lib/wfdbio.c`, to make configuration constants accessible at run time; these are needed by the WFDB Toolkit for Matlab.

(WFDB library version 10.4.19 was identical to version 10.4.18.)

Changes in version 10.4.18 (15 March 2009)

The WFDB library no longer reports spurious checksum errors when reading EDF files, which don't have checksums.

Changes in version 10.4.17 (5 March 2009)

Internal WFDB library function `wfdb_checkname` now allows hyphens (`-`) within record and annotator names.

Changes in version 10.4.16 (3 March 2009)

WFDB library function `strtim` now rounds rather than truncating when the sampling frequency is not an integer.

Changes in version 10.4.15 (26 February 2009)

WFDB library function `mstimstr` now outputs time to the nearest millisecond, rather than truncating its calculation to the next lowest number of milliseconds.

A new WFDB library function, `wfdbputprolog`, can write a prolog at the beginning of a signal file. WFDB applications ignore embedded prologs.

Changes in version 10.4.14 (23 February 2009)

WFDB library function `setwfdb()` now exports the WFDB library environment variables (`WFDB`, `WFDBCAL`, `WFDBANNSORT`, and `WFDBGVMODE`) on all modern platforms, so that a process started by a WFDB application inherits the WFDB environment of its parent. (This is not possible on platforms that do not provide a working `putenv()` function, such as MacOS 9 and earlier, and 16-bit versions of MS-Windows.) Some previous versions also included this capability, but the older implementation caused a memory leak, and it was disabled in version 10.4.6. Thanks to Omar Abdala and Dan Scott for reporting the problem and help in identifying its cause.

Changes in version 10.4.13 (16 February 2009)

A new WFDB library function, `tnextvec()`, finds the next valid sample from a chosen signal, occurring at or after a specified time. This function is particularly useful when reading variable-layout multi-segment records that may have lengthy gaps in signals of interest.

For portability, the WFDB library has always stored parameters such as sampling frequency and gain as strings rather than as floating-point numbers. Although the resultant loss of precision has been almost always negligible, it has been observable in the specific case of converting very large time intervals from sample intervals to seconds, when the sampling frequency in Hz cannot be represented exactly in binary (e.g., if the sampling frequency is once per minute, or 1/60 Hz). This situation cannot be wholly avoided, but it can be minimized. Changes in this release, in `setheader()` and in functions that record time units in annotation files, increase the precision with which non-integer parameters are recorded, so that loss of precision as a result of converting them to and from strings should almost never be observable.

Changes in version 10.4.12 (20 January 2009)

The rule for sorting annotations within a file has been changed to allow a much larger number of simultaneous annotations (i.e., annotations in a given annotation file with identical `time` fields) than was previously possible. Since version 6.1, the WFDB library sorts simultaneous annotations according to the value of their `chan` fields, allowing for 256 simultaneous annotations at any given time. Beginning in version 10.4.12, the library sorts

simultaneous annotations according to their `num` fields, then sorts those with identical `num` fields according to their `chan` fields, allowing up to 65,536 simultaneous annotations at any given time.

A new WFDB library function, `findsig`, returns the signal number of the input signal matching its string argument, or -1 if no such input signal exists. If the string argument could be interpreted as an input signal number, it is taken as such; otherwise, the string argument must be an exact match to a signal name (`desc` field in the `siginfo` structure).

Previous versions of WFDB library function `setifreq` entered an infinite loop if invoked (contrary to specifications) before opening an input record. `setifreq` now detects the error, emits an appropriate warning, and returns.

If a WFDB application that uses WFDB library version 10.4.5 through 10.4.11 attempted to read an annotation file before reading the sampling frequency of the associated record (for example, by invoking `isigopen` or `sampfreq`), the annotation times might all appear to be zero. This may occur when reading annotations created using WFDB 10.4.5 or later. The times supplied when creating the file are correctly written but may be incorrectly read in these cases. This problem was corrected in this release; thanks to Thomas Heldt for reporting it and providing a reproducible example of it.

(WFDB library version 10.4.11 was identical to version 10.4.10.)

Changes in version 10.4.10 (31 October 2008)

EDF digital maximum and minimum values are now read properly in 64-bit builds; previous versions had a bug in `edfparse` (an internal WFDB library function defined in `lib/signal.c`) that did not appear in 32-bit builds. Thanks to Joe Mietus for reporting the problem.

Changes in version 10.4.9 (10 October 2008)

The WFDB library once again correctly interprets a hyphen (`-`) (used in place of a record name, annotator name, signal file name, or calibration file name) as a reference to the standard input or output, for platforms that support POSIX standard I/O (see Section 5.2 [Standard I/O], page 73). This feature was broken in 10.4.5 as a side effect of changes in `wfdb_open` (an internal WFDB library function defined in `lib/wfdbio.c`).

(WFDB library version 10.4.8 was identical to version 10.4.7.)

Changes in version 10.4.7 (15 July 2008)

Yinqi Zhang reported and contributed a fix for a memory leak in `make_vsd()` (an internal WFDB library function defined in `signal.c`).

Changes in version 10.4.6 (9 April 2008)

The WFDB functions `setafreq()` and `getafreq()` (for setting and getting the time resolution of newly-created output annotation files in ticks per second) were new in version 10.4.5, but were undocumented. They are now described in this Guide, and wrappers for these functions are now included in `fortran/wfdbf.c`.

An important change in the WFDB library: memory allocation errors are now treated as fatal by default (in previous versions, the functions that encountered them returned error values that permitted the application to handle them). These errors occur when there is

insufficient memory available to the WFDB library. To obtain the old behavior, in which the calling function will continue execution if possible after a memory allocation error, invoke `wfdbmemerr(0)`. By default, however, such an error will cause the process to terminate. In either case, the WFDB library emits an appropriate error message to aid in troubleshooting.

New macros for handling dynamically allocated memory are defined in `lib/wfdblib.h` and used throughout the WFDB library, eliminating most known memory leaks. Three known leaks remain (in `setecgstr()`, `setannstr()`, and `setanndesc()`); these are documented and harmless in current applications. Thanks to Yinqi Zhang for reporting a leak in `copysi()` (an internal WFDB library function defined in `signal.c`), which prompted the cleanup.

WFDB functions `strecg()`, `setecgstr()`, `strann()`, `setannstr()`, and `setanndesc()` now handle NULL string inputs properly. (Previous versions passed NULL strings to `strcmp()`, with undesirable results.)

Changes in version 10.4.5 (6 February 2008)

Bob Farrell and Tony Ricke chased down and provided fixes for memory leaks in several WFDB library functions, and also provided revisions to permit additional type checking and to avoid type mismatch warnings.

Changes in the build system make it easier to build WFDB using Cygwin gcc (with or without the Cygwin POSIX library).

When creating annotation files, if the input sampling frequency differs from the frame rate of the input record (either because of using `WFDB_HIGHRES` mode while reading a multi-frequency record, or because of having used `setifreq()` to modify the sampling frequency), a comment is written to the beginning of the annotation file indicating the resolution of the annotation times in ticks per second (thus allowing the application to store its annotations with whatever time resolution is desired). When reading an annotation file, if such a resolution comment is found, getann adjusts the times of annotations to match the currently defined sampling frequency. The resolutions are kept independently for each annotation file, so (for example) `bxb` can compare two annotation files written with different resolutions.

The ability to set the time resolution of annotation files has required a minor change in the semantics of `setifreq()`. It is now necessary to invoke `setifreq()` before creating an annotation file that will have a resolution matching the (modified) input sampling frequency. Since `setifreq()` must be invoked after opening the input signals, this implies that `wfdbinit()` cannot be used to open both input signals and output annotation files if `setifreq()` is to be used; rather, the sequence should be `isigopen()`, `setifreq()`, and finally `annopen()`.

If a string that includes a '`.`' is supplied to a WFDB library function where a record name is expected, the WFDB library assumes that it is the name of a file located in the WFDB path. If the name ends in '`.hea`', the file is assumed to be a WFDB-format header file, and its record name is assumed to be the first part of the string, exclusive of the '`.hea`'.

This version also includes support for reading EDF files natively. If a string supplied as a record name contains a '`.`' but does not end in '`.hea`', it is assumed to be the record name of an EDF file of the same name.

(WFDB library versions 10.4.3 and 10.4.4 were identical to version 10.4.2.)

Changes in version 10.4.2 (4 May 2006)

Mathias Gruber reported a line in wfdbio.c that used void pointer arithmetic (permitted as an extension by gcc but not allowed by ANSI/ISO C or most other C compilers). This operation has been replaced by ANSI/ISO C-conformant code.

Changes in version 10.4.1 (6 April 2006)

A bug caused incorrect output from WFDB library function strtim() when called with the argument `"i"`, following use of setifreq() to change the effective sampling frequency, resulting in incorrect output from example 10 in the WFDB Programmer's Guide. This has now been corrected.

Changes in version 10.4.0 (2 March 2006)

Version 10.4.0 and later versions of the WFDB library are intended to be compiled using ANSI/ISO C (and C++) compilers only; previous versions also supported the use of traditional (K&R) C compilers. The most obvious change resulting from this decision is in the use of prototypes in function declarations, an innovation of ANSI C that permits better error-checking by compilers. The ANSI/ISO C standard is now more than 15 years old, and it has been over 10 years since a C compiler that does not support function prototypes was used for development of the WFDB library. Code in `wfdbio.c` that provides limited support for compilers that do not provide an ANSI/ISO C library has been retained for now, and `wfdb.h` still includes a set of K&R C function declarations; both of these features are deprecated, however, and may be removed in future versions of the WFDB library. Users who still need to use a K&R C compiler to compile the library itself may find 'unprotoize' (included in the GNU gcc distribution) to be helpful.

The mapping of lowest expressible sample values to `WFDB_INVALID_SAMPLE` performed by `getframe()` (in `lib/signal.c`) did not work properly for signal formats 80 and 160 (in which samples are recorded as unsigned integers); this has now been corrected.

The symbol `WFDB_GVPAD` is newly defined in `<wfdb/wfdb.h>`. It may be added to `WFDB_HIGHRES` or `WFDB_LOWRES` and given as input to `setgvmode()`. The effect of doing so is that missing samples, and samples recognized as invalid, are replaced by `getframe()`, `getvec()`, and `sample()` with the most recently read valid values rather than by the special value `WFDB_INVALID_SAMPLE`. This behavior allows applications such as digital filters to remain ignorant of missing data without significant performance penalties.

`sample()` now checks that its signal number input is valid, and returns `WFDB_INVALID_SAMPLE` if not. In previous versions, `sample` returned a sample from signal 0 if the requested signal was unavailable.

`sample_valid()` now returns -1 in the case of a signal that becomes unavailable before the end of the record (previous versions returned 1 in this case).

The FIR filter example (see [Example 7], page 89) now works properly. The previous version always began processing the input at sample 0, regardless of start time specified in its argument list.

WFDB 10.3

Changes in version 10.3.17 (20 August 2005)

This version is the first to support reading variable-layout records (multi-segment records in which the number, arrangement, gains, and baselines of the signals may vary from one segment to the next; see Section 5.5 [Multi-Segment Records], page 74).

Rounding errors in the WFDB library's `mstimstr` function have been reduced. Previous versions did not always round appropriately when the sampling frequency was much less than 1 Hz.

The maximum length for a record name (`WFDB_MAXRNL`, defined in `wfdb.h`) has been increased from 11 to 20.

A new constant, `WFDB_INVALID_SAMPLE`, is now defined in `wfdb.h`. It is used to identify padding inserted to fill in for missing data. When writing in any format that uses fewer than 16 bits per sample, `putvec` maps `WFDB_INVALID_SAMPLE` to the lowest (most negative) value expressible in that format; when reading a signal file in such a format, `getframe` performs the inverse mapping, so that missing data can be identified regardless of the data format.

A side effect of this change is that (for example) any samples that had the most negative value (for example, -2048 in a format 212 signal file) are now flagged as invalid. To treat such samples as invalid is reasonable, however, since these occur only when the input level falls below the working range of the analog-to-digital converter.

Changes in version 10.3.16 (13 June 2005)

Benjamin Moody has added an interface between the WFDB library and `libcurl` as an alternative to the existing `libwww` interface, and has updated `configure` and `conf/*.def` to search for and use `libcurl` if it is available. The primary advantages of `libcurl` over `libwww` are that `libcurl` is smaller and faster, it supports access to password-protected files, and it is actively maintained. Both libraries are freely available on all popular platforms.

Isaac Henry has updated `configure` to support building a native MS-Windows version of the WFDB library using either Cygwin gcc or MinGW gcc.

A number of minor changes, mostly involving conditional use of `malloc.h`, `stdlib.h`, and `string.h`, were made to eliminate warnings from gcc 4.x.

Changes in version 10.3.15 (31 January 2005)

Rules for generating the binary tarball for MS-Windows have been fixed so that the Cygwin DLLs are now included with correct permissions.

Installation of shared libraries under GNU/Linux requires an extra step if SELinux is enabled (as under Fedora Core 2 and later); this has been incorporated into `conf/linux-slib.def`.

Changes in version 10.3.14 (29 December 2004)

Guido Muesch reported that `getspf()` did not always return correct results if the frame frequency does not have an exact representation as a double precision floating point number. This problem has now been corrected.

Changes in version 10.3.13 (5 May 2004)

Using an indirect WFDB path (i.e., setting the WFDB environment variable to a value such as '`@FILE`', where '`FILE`' contains the desired path) was broken in WFDB library versions 10.3.9 through 10.3.12; it now works again, thanks to a patch contributed by Fred Geheb.

Changes in version 10.3.12 (9 March 2004)

Okko Willeboordse pointed out an incompatibility between the native MS-Windows API and the ANSI/ISO C library function `mkdir`, which is used by the WFDB library. This does not present a problem when compiling the WFDB library using the supported Cygwin/gcc compiler under MS-Windows, nor does any related problem occur on any other platform. It should now be a little easier to compile the WFDB library using unsupported compilers, thanks to a new `MKDIR` macro that hides the incompatibility (see `lib/wfdblib.h0`).

Piotr Wlodarek initiated a discussion about memory leaks in the WFDB library, citing as an example the 'trivial example program in C' from this Guide, which does not free memory it allocates in `isigopen()` when reading the signal specifications. This problem can be avoided by invoking `wfdbquit()` in the example program, just before exiting. Further discussion of this point has been added to this Guide following the presentation of the 'trivial example', and in the description of `wfdbquit`.

Changes in version 10.3.11 (17 October 2003)

In `signal.c`, several bugs have been identified and fixed. Thanks to Piotr Wlodarek, who found a buffer overrun in `isigopen`. Also, `isgsettime` sometimes performed incorrect seeks on multifrequency records that had been opened in high-resolution mode; this has been fixed, together with a related bug that caused the value returned by `strtim("e")` to be calculated incorrectly in some such cases.

Changes in version 10.3.10 (3 August 2003)

In version 10.3.9, the functions `setannstr`, `setanndesc`, and `setecgstr` did not contain necessary checks to avoid invoking `strcmp` with a `NULL` argument. These checks have been added in version 10.3.10. Thanks to Thomas Heldt for reporting this problem.

Changes in version 10.3.9 (16 July 2003)

The WFDB library functions `setwfdb`, `setannstr`, `setanndesc`, and `setecgstr` now copy their input string arguments, so that it is no longer necessary for WFDB applications to keep these strings valid. If you have created applications that rely on being able to modify these strings, it will be necessary to invoke the corresponding functions again before such changes will take effect within the WFDB library.

Previous versions of the WFDB library function `putinfo` did not flush their output until either a new header file was created (via `setheader` or `newheader`) or the process exited. This has now been corrected, and putinfo output is now flushed before `putinfo` returns. Thanks to Jonas Carlson for reporting this problem.

Changes in version 10.3.8 (12 July 2003)

The WFDB library function `setbasetime()` now properly accepts arguments specifying midnight (e.g., '`0:0:0`'), which previous versions rejected, and the function `setheader()` records such times correctly in the `.hea` files it creates.

(WFDB library version 10.3.7 was identical to 10.3.6.)

Changes in version 10.3.6 (7 April 2003)

The fix applied in `isigclose()` in 10.3.5 was incomplete but is now (really!) fixed. Applications that use `sample()` should call `wfdbquit()` to be certain that `sample`'s buffer is freed before exiting.

Some long-standing problems in the code (in `lib/wfdbio.c`) that handles http range requests for NETFILES-enabled versions of the library have been partially addressed. The underlying issue is that http servers do not always return the range of bytes requested; when this happens, it is not difficult to determine that there is a problem, but it is tricky to figure out what to do about it. Based on experiments with several different http servers, the strategy for handling such problems within the WFDB NETFILES code has been improved substantially, but there may be further room for improvement.

Changes in version 10.3.5 (31 March 2003)

Fixed a bug in WFDB library function `isigclose()` (in `lib/signal.c`) that had caused `sample()`'s buffer to be freed inappropriately when switching segments in a multi-segment record. Thanks to Dave Schaffer for the bug report and for a test case that illustrated the bug.

(WFDB library versions 10.3.3 and 10.3.4 were identical to 10.3.2.)

Changes in version 10.3.2 (25 February 2003)

Fixed a WFDB library bug that caused annotation sorting to fail if a new header file had been written. Thanks to Winton Baker for reporting this problem and for providing an example that illustrated the bug.

(WFDB library version 10.3.1 was identical to 10.3.0.)

Changes in version 10.3.0 (26 November 2002)

Fixed bugs in `lib/signal.c` that caused improper accounting of signal group numbers when reading from two or more records at the same time (as in `nst`), a bug that caused a segfault in `nst`, and a bug that referenced uninitialized memory in `newheader` if `nsig` = 0.

The WFDB Software Package has been ported to Mac OS X (Darwin), version 10.2 (the port should also work under 10.1 but this has not been tested and will not be supported).

It is now possible to generate a shared WFDB library (DLL) under MS-Windows using Cygwin/gcc.

Added functions `sample` and `sample_valid` to the WFDB library (in `lib/signal.c`). `sample(s, t)` returns the sample at time (sample number) `t` from signal `s`, handling all necessary buffering internally and allowing the caller to treat the signal file as a virtual array of randomly accessible samples. `sample_valid` can be invoked to check if the most recent value returned by `sample` was valid (e.g., to see if the end of the input was reached). For an example of the use of these functions, see `app/wqrs.c`.

WFDB 10.2

Changes in version 10.2.9 (27 October 2002)

Fixed a bug in example 9 in this guide (introduced in version 10.2.0).

Updated `lib/wfdbdll.def` and the `Makefile.dos` files in several directories. These have not been tested in recent years and may need further revisions; feedback is welcome.

Corrected persistent problems with generating PDF versions of the manuals for the desired page size, and added hyperlinks to the PDF version of this guide.

(WFDB library version 10.2.8 was identical to 10.2.7.)

Changes in version 10.2.7 (14 October 2002)

Added a workaround to `wfdb_fclose` (in `lib/wfdbio.c`) so that closing `stdin` after using `freopen` doesn't trigger a core dump.

If out-of-order annotations were written and automatic annotation sorting was suppressed, the warning produced by `oannclose` (in `lib/annot.c`) once again includes the correct `sortann` command needed to put the annotations into order. (This feature was broken by a previous revision.)

Changes in version 10.2.6 (24 June 2002)

The new functions `setifreq` and `getifreq` allow an application to choose any convenient sampling frequency for reading input signals. Samples read from signal files using `getvec` are buffered, resampled, and delivered to the calling application as if the original signals had been sampled at the desired frequency. Times expressed in sample intervals passed to or from other WFDB library functions (`getann`, `putann`, `mstimstr`, `timstr`, and `strtim`) are rescaled as needed to match intervals corresponding to the chosen frequency. Thanks to Pat Hamilton for the inspiration!

The WFDB library now records the base time with millisecond precision (previous versions did so with one-second precision), and `xform` provides starting times to the library function `setbasetime` with millisecond precision. Thanks to Allavatam Venugopal for providing examples that illustrated the need for these features.

Fixed deskewing buffer initialization in `getframe`, broken by the 10.2.0 update, which introduced an infinite loop when reading a record that requires skew correction starting at sample 0. Thanks to Andrew Walsh for finding an example that triggered this bug.

Fixed rounding errors in `adumuv`, `muvadu`, and `physadu`. Previous versions rounded negative values toward zero; to obtain consistent conversions, however, it is necessary to round all values down (e.g., from -1.5 to -2 rather than up to -1).

Fixed a memory leak in `wfdb_fclose` (in `lib/wfdbio.h`). Thanks to Ion Gaztañaga.

Changes in version 10.2.5 (10 March 2002)

Additions and fixes in `wfdbf.c` (the Fortran wrappers for the WFDB library).

Changes in version 10.2.4 (20 December 2001)

Code in `wfdbio.c` that required the use of the string `header` to identify a header file has been revised so that the standard `hea` is now usable for this purpose in all cases.

Changes in version 10.2.3 (14 December 2001)

Portability fixes in `wfdblib.h`. (WFDB library version 10.2.2 was identical to 10.2.1.)

Changes in version 10.2.1 (16 November 2001)

Most users will no longer need to set the WFDB path explicitly, as a result of several minor changes in the default path and in the installer for the WFDB Software Package.

The environment variable `WFDBNOSORT` was replaced by `WFDBANNSORT`, and the environment variable `WFDBGVMODE` was introduced (see Section 5.10 [Annotation Order], page 78, and see Section 5.4 [Multi-Frequency Records], page 73, for details).

Changes in version 10.2.0 (15 October 2001)

There are no longer any fixed limits on the numbers of signals or annotation files that can be opened simultaneously, or on the number of samples per signal per frame. In previous versions of the WFDB library, the symbols `WFDB_MAXSIG`, `WFDB_MAXANN`, and `WFDB_MAXSPF` (all defined in `<wfdb/wfdb.h>`) specified limits on these parameters that could be modified only by recompiling the WFDB library. These symbols are still defined for compatibility with older applications that use them (typically to determine the size of static arrays).

Since version 10.1.1, record names may include path information (see the notes for version 10.1.1 below), but if such names are used to generate names of WFDB output files, the user has been required to ensure that the target directory exists. This requirement is eliminated in version 10.2.0. If an output file is specified to be located in a non-existent directory, the WFDB library will attempt to create the directory (including, if necessary, any non-existent parent directories). This feature simplifies the use of record names that include directory information, as is common when reading data from a CDROM or a web server such as PhysioNet. For example, using the WFDB path (`. http://physionet.org/physiobank/database`), if the current directory, `.`, does not contain a subdirectory named `mitdb`, the command:

```
sqrs -r mitdb/100
```

will read its input from `http://physionet.org/physiobank/database/mitdb/`, will create a directory named `mitdb` within the current directory, and will write its output annotation file (`100.qrs`) into this newly-created directory. If we then use the command:

```
rdann -r mitdb/100 -a qrs
```

the header file is still read from the remote directory, but the annotation file is read from `./mitdb`. (The programs `sqrs` and `rdann` are standard applications that use the WFDB library; see the *WFDB Applications Guide* for details.)

Also new is the WFDB test suite (located in the `checkpkg` directory of the WFDB source tree, at the same level as the `lib` directory containing the WFDB library sources). This set of programs can be used to help verify that a newly-installed version of the WFDB library behaves properly.

WFDB 10.1

Changes in version 10.1.6 (1 August 2001)

The WFDB library requires that the record name specified in the first line of a header file must match the name of the record with which the header file is associated (this is done in order to detect corrupted or erroneously renamed header files). Version 10.1.6 requires that only the final portion of the record name (stripped of any path information) must match.

Changes in version 10.1.5 (11 June 2000)

More changes in the `make` description files, for Cygwin compatibility.

Changes in version 10.1.4 (6 June 2000)

The symbol `WFDB_NETFILES` replaces the old `NETFILES`.

Changes in version 10.1.3 (26 April 2000)

More changes in the `make` description files, to support a configuration script.

Changes in version 10.1.2 (11 March 2000)

Changes in the `make` description files.

Changes in version 10.1.1 (30 January 2000)

Record names may contain (absolute or relative) path information as a prefix, and if (as a result) an input file is found in a location that does not appear explicitly in the WFDB path, that location is appended to the end of the WFDB path. For example, if the WFDB path is `. http://physionet.org/physiobank/database`, and the record name `mitdb/100` is supplied to `wfdbinit`, the WFDB library will find the header file at `http://physionet.org/physiobank/database/mitdb/100.hea`, and will then add `http://physionet.org/physiobank/database/mitdb/` to the end of the WFDB path so that the signal file (specified as `100.dat` in the header file) can be found.

Changes in version 10.1.0 (15 January 2000)

Version 10.1.0 supports a new signal file format (311), and contains numerous minor changes in the NETFILES support code introduced in 10.0.1.

WFDB 10.0

Changes in version 10.0.1 (19 November 1999)

Beginning with version 10.0.1, the WFDB library supports reading not only local files, but also remote files made available by web (HTTP) or FTP servers. To make use of this feature, link your application with both the WFDB library and the `libwww` library (freely available for all versions of Unix, and for most recent versions of MS Windows, from `http://www.w3.org/Library`, or from `http://www.physionet.org/physiotools/libwww/`). (In some cases, notably under GNU/Linux, `libwww` is linked together with the dynamically-loaded version of the WFDB library, so that you do not need to link `libwww` explicitly.) All access to remote files is read-only. If you do not wish to allow access to remote files, or if `libwww` is not available for your OS, simply do not define the symbol NETFILES when compiling the WFDB library. For further details, see `wfdbio.c` in the WFDB library sources.

The WFDB environment variable may now contain whitespace (space, tab, or newline characters) as path component separators under any OS. Multiple consecutive whitespace characters are treated as a single path component separator. Use a '`.`' to specify the current directory as a path component when using whitespace as a path component separator. A semicolon ('`;`') is also acceptable as a path component separator under any OS. A colon ('`:`') is still acceptable as a path component separator under Unix (Linux, etc.), provided only that the colon is not immediately followed by '`//`'.

If the WFDB path includes components of the forms `http://somewhere.net/mydata` or `ftp://somewhere.else/yourdata`, the sequence '`://`' is explicitly recognized as part of a URL prefix (under any OS), and the '`:`' and '`/`' characters within the '`://`' are not interpreted further. Note that the MS-DOS '\' is *not* acceptable as an alternative to '`/`' in a URL prefix. To make WFDB paths containing URL prefixes more easily (human) readable, use whitespace for path component separators.

Previous versions of the WFDB library that were compiled for environments other than MS-DOS used file names in the format *type.record*. This file name format is no longer supported.

Changes in version 10.0.0 (25 June 1999)

Beginning with version 10.0.0, the name of the library is WFDB. All earlier versions were named DB. All library symbols have been similarly renamed, with `WFDB` and `wfdb` replacing `DB` and `db` everywhere, in names of library functions, constants, type and structure definitions, library source file names, and names of environment variables (e.g., the DB environment variable is now the WFDB environment variable).

Version 10.0.0 of the WFDB library is functionally identical with the final release (version 9.7.4) of the DB library, except for the name changes. It should be possible to recompile existing applications written for DB library version 9.x without modification, and to link them with WFDB library version 10.0.0. This is possible because two sets of `#include` files are provided with the WFDB library. The first set, accessible via `#include <wfdb/...>`, works with applications written as described in this guide. The alternate set, accessible via `#include <ecg/...>`, is compatible with DB 9.x applications as described in previous editions of this guide.

Concept Index

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Function and Macro Index

For a number of entries below, the function name is followed by the version number of the WFDB library in which the function first appeared. Functions for which no such number appears have been present in all numbered versions of the WFDB library.

A

C

D

E

F

G

I

M

N

O

P

R

S

T

U

W