ECG Database Programmer's Guide

Ninth Edition (revised and with additions for DB library version 9.7)
24 May 1997

George B. Moody

Harvard-MIT Division of Health Sciences and Technology
Biomedical Engineering Center

For information on obtaining the most recent versions of the software described in this guide, visit `http://ecg.mit.edu`, or write to:

MIT-BIH Database Distribution
Massachusetts Institute of Technology
77 Massachusetts Avenue, Room 20A-113
Cambridge, MA 02139
USA

An HTML version of this guide is available; point your web browser to `http://ecg.mit.edu/dbpg/` to view it. Additional printed copies of this guide are available for US$10 each (including shipping by surface mail) by writing to the address above. Please make checks payable to "Beth Israel Hospital Biomedical Engineering Division."

Preface

This guide documents the ECG database interface library (the *DB 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 designed for use with 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 DB 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 DB library has evolved to support the development of several 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 not yet completed MIMIC Database. Thus the DB 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 DB Software Package, and is described in the *ECG Database Applications Guide*; other documents describe the databases themselves, and existing programs that use them (see Appendix E [Sources], page 129, 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 DB library was designed consist of a small number of ECG *records*, each of which is quite large (typically a megabyte or more). Database records usually originate as multi-channel analog tape recordings that have been digitized and stored as disk files. For this historical reason, they are often referred to as *tapes*. 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*, which are three-digit numbers for MIT DB records, four-digit numbers for AHA DB records, and four-digit numbers prefixed by '`e`' for ESC DB records.

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-DOS 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 part of its name. (On writable UNIX and Macintosh file systems, the record name is usually the last part of the file name; the opposite convention is used on CD-ROM and MS-DOS file systems, because of their restrictions on the length of file name suffixes.) 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 on a magnetic disk a file of your own annotations for a record distributed on a CD-ROM, 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 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 74, 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 5.9 [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 '`atruth`' (under MS-DOS, '`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 DB library user as C structures, the fields of which specify time, beat type, and several user-definable variables. The DB 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 DB library are these:

- A *waveform editor*, such as `wave` (see Appendix E [Sources], page 129), 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 90) 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 99) read the digitized signals, analyze them, and then record their own annotations.

- An *annotation comparator*, such as `bxb` (see Appendix C [DB Applications], page 119), 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 DB 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 DB 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*. If the entire DB Software Package is available, the *ECG Database Applications Guide* will be useful as a reference for existing DB library-based applications; if you have only a CD-ROM with MS-DOS binaries, refer to '`bin.doc`' (in the '`bin`' directory of your CD-ROM) for information on the available applications.

You should have access to a computer that has the DB library and at least one or two database records on-line. (If you are installing the DB library on a new computer for the first time, please read the installation notes supplied with the DB library first, or see Appendix B [Installing the DB 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 *ECG Database Applications Guide*; note, however, that support for new file formats is added to the DB library from time to time, so that the information you find there may be incomplete.) The DB 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 DB library to read the files; then you will at least have a program that requires only recompilation with a new version of the DB library when file formats change. *In extremis*, use '`rdann`' and '`rdsamp`' — MS-DOS executables of which are included on all CD-ROM databases of ECGs — 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. 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 DB 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 DB library has not yet been installed on your system, see Appendix B [Installing the DB Software Package], page 115. Another appendix (see Appendix C [DB Applications], page 119) includes brief descriptions of the application programs that are distributed with the DB library as part of the DB software package.

The DB 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, Linux, AIX, AUX, SCO, Coherent, and more), MS-DOS, MS-Windows, VMS, and the Macintosh OS. This guide was written for UNIX users (with notes for MS-DOS users where differences exist), but others should find only minor differences. One obvious difference between MS-DOS and other versions of the DB library is the file-naming convention: since record names can be longer than the three characters permitted in MS-DOS file name "extensions", MS-DOS names for DB files *begin* with the record name and *end* with a file type specifier ('`hea`' instead of '`header`', '`atr`' instead of '`atruth`', etc.), rather than the other way around. This difference is not particularly significant, however, since DB file names *per se* are never needed as DB library function arguments. So that database files on CD-ROMs may be read on all supported systems, the MS-DOS convention for naming DB files is recognized in addition to the "native" convention under UNIX, VMS, and on the Macintosh.

Another appendix discusses porting the DB 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 125). 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 129).

Many friends have contributed to the development of the DB library. Thanks to Paul Albrecht, Ted Baker, Phil Devlin, Scott Greenwald, David Israel, Roger Mark, Joe Mietus, Warren Muldrow, and especially to Paul Schluter, whose elegant 8080 assembly language functions inspired these (long live `getann`!). Pat Hamilton and Bob Farrell contributed ports, to the Macintosh and the MS 32-bit Windows environments, respectively. 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.

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 DB library itself, have been extensively revised since they first appeared in 1981. Your comments and suggestions are welcome. Please send them to:

George B. Moody
MIT Room 20A-113
Cambridge, MA 02139
USA

(e-mail: george@hstbme.mit.edu)

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 DB Software Package; type `C-h i` within GNU `emacs` to start up `info` (see Appendix E [Sources], page 129, for information about obtaining GNU `emacs`).

An HTML version of this guide, suitable for viewing using any web browser, is included with the DB Software Package. The latest version may always be viewed at `http://ecg.mit.edu` using your web browser.

Recent changes

This node contains a brief summary of changes to the DB library and to this guide since the first printing of the ninth edition of this guide in July, 1995. See '`NEWS.TXT`' and '`NEWNEWS.TXT`', in the top-level directory of the DB Software Package distribution, for details on these changes, and for information on any more recent changes that may not be described here.

Changes in version 9.7

Support for compiling the DB library as a 32-bit MS Windows DLL has been added (thanks to Bob Farrell). Define the symbols `_WIN32` and `_WINDLL` when building the DLL, and define the symbol `_WIN32` when compiling an application that uses the DLL.

The string '`%r`' is replaced by the record name wherever it appears in the DB path (see Section 1.3 [DB path], page 13. The string '`%`*N*`r`', where *N* is a non-zero digit, is replaced by the first *N* characters of the record name. When followed by any other character, '`%`' is an 'escape'; thus, for example, to insert a literal '`%`' in the DB path, use '`%%`'.

It is now possible to write annotations out-of-order using **putann** (see Section 5.9 [Annotation Order], page 78).

Changes in version 9.6.2

Version 5.0 and later versions of Apple's ISO 9660 CD-ROM driver for the Macintosh handle version number suffixes more sensibly than do earlier versions. When compiling the DB library on the Macintosh, the new driver behavior is assumed by default; if you are still using a pre-5.0 version of this driver, define the symbol FIXISOCD when compiling the DB library.

Changes in version 9.6.1

A new annotation type (`LINK`) has been defined (see Chapter 4 [Annotation Codes], page 67).

Changes in version 9.6

A new function was added to the DB library (see [getspf], page 50).

A number of changes in the implementation of existing DB library functions now make use of high-resolution mode (see Section 5.4 [Multi-Frequency Records], page 74) almost completely transparent to DB applications. The recommended practice is to call `setgvmode(DB_HIGHRES)` *before* calling `annopen`, `dbinit`, `getvec`, `sampfreq`, `strtim`, or `timstr`. By doing so, all `DB_Time` data visible to the application are in units of the high-resolution sampling intervals(see [setgvmode], page 49).

Note that the resolution of annotation times in annotation files created while in high-resolution mode remains that of the low-resolution sampling intervals. This may change in a future version of the DB library.

Changes in version 9.5

The number of signals that may be read simultaneously, `DB_MAXSIG`, was increased from 16 to 32.

Changes in version 9.4.2

Support for use of the DB library with Fortran programs was added (see [other languages], page 13).

Changes in version 9.4.1

In release 9.4.0 only, DB library function `setheader` did not properly interpret the skew setting made by `dbsetskew` in the case of an oversampled signal in a multi-frequency record. This has been corrected.

Changes in version 9.4

Four new functions were added to the DB library to simplify management of records with signal files containing prologs or intersignal skews.

see [dbgetskew], page 56

see [dbsetskew], page 57

see [dbgetstart], page 57

see [dbsetstart], page 57

Also, see Appendix A [Glossary], page 107, for definitions of *prolog* and *skew*.

`newheader` and `setheader` now write header files containing skew and byte offset fields if you have previously set non-zero values for these parameters using `dbsetskew` or `dbsetstart`.

If `dberror` is invoked before any errors have occurred, the character string it returns now contains the DB library version number. Versions earlier than 9.4 returned an empty string in this case.

1 Using the DB Library

This chapter gives a brief overview of the steps needed to compile, load, and run a program that uses the DB library. It assumes that you are able to log onto a UNIX-based computer on which the DB Software Package has been installed (see Appendix B [Installing the DB 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

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 with all source distributions of the DB Software Package. If you have only the MS-DOS binary version of the DB library on a CD-ROM, substitute the name of one of the records on your CD-ROM for '`100s`' in the program below.) We might begin by creating a source file called '`psamples.c`' that contains:

```
#include <ecg/db.h>

main()
{
    int i;
    DB_Sample v[2];
    DB_Siginfo s[2];

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

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

```
#include <ecg/db.h>
```

which defines function interfaces, data types (such as the `DB_Sample` and `DB_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 <ecg\\db.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 59). 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.

1.2 Compiling a Program with the DB Library

To compile the example program on a UNIX system, we can say:

```
cc -o psamples psamples.c -ldb
```

to produce an executable program called `psamples`. You may use any other compiler options you choose, but the '`-ldb`' option 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 DB 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 '`-ldb`' option.

Each MS-DOS C or C++ compiler has its own idiosyncratic syntax, so no general rule can be given for MS-DOS. With Microsoft C/C++, use:

```
cl psamples.c -link db
```

With Borland C/C++, use:

```
bcc -LLIBDIR psamples.c db.lib
```

where *LIBDIR* is the directory in which '`db.lib`' (the DB library) has been installed. (Substitute '`tcc`' for '`bcc`' if you are using Turbo C or C++.) See your compiler manual for further information.

Using the DB library with other languages

If you prefer to write your applications in C++, you may do so, but note that the DB library is written in C. (Most C++ compilers can be run in ANSI/ISO C compatibility mode in order to compile the DB library itself.) Each C++ source file that uses DB library functions must include '`<ecg/db.h>`', in order to instruct your compiler to use C conventions for argument passing and to use unmangled names for the DB 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 '`<ecg/db.h>`' so that the symbols '`db_CPP`' and '`db_PROTO`' are defined at the top of the file. 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.

A set of wrapper functions is also available for those who wish to use the DB library together with applications written in Fortran. These functions provide a thin 'wrapper' around the DB library functions, by accepting Fortran-compatible arguments (there are no structures, and all arguments are passed by reference rather than by value). You will still need a C or C++ compiler to compile the DB library and the wrapper functions themselves. The Fortran wrapper functions are not discussed in this guide; for further information, refer to '`fortran/readme.doc`' in the DB Software Package (included in version 9.4 and later versions).

1.3 The Database Path

When DB applications *read* database files, they must be able to find them in various directories that may vary from system to system. The DB library refers to a character string that consists of an ordered list of directories to be searched *for input files*. This string is called the *database path*, or the *DB path*. Under UNIX and VMS, the DB path is a colon-separated list of prefixes, in the format used for the Bourne shell's `PATH` variable. Under MS-DOS and on the Macintosh, the DB path is 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). When a DB file must be opened for input, the DB library attempts to locate it by attaching each of the components of the DB path (one at a time) as a prefix to the file name. If two or more matching files exist in different directories in the DB path, the DB library opens only the file that resides in the first of these directories named in the DB path. Any other matching files are effectively invisible to DB applications unless the DB path is rearranged.

When DB 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.)

Note particularly that the current directory is *not* necessarily part of the DB path. In order to be sure that your DB applications can read any DB files that you have previously written in the current directory, be sure that your DB path includes an empty (null) component, which corresponds to the current directory. In most cases, this null component should be the first in the DB path. Thus, if you write into the current directory a modified version of an existing DB file, any later actions that would read this file will read your modified version rather than the original.

A default DB path may be specified at the time the DB library is compiled, by defining a value for the symbol `DEFDBP` in '`dblib.h`'. On most systems, the environment variable `DB`, if set, specifies the value of the DB path, and overrides the default value if any. On the Macintosh, for which the concept of environment variables is foreign, the DB path may be set only by using `DEFDBP`.

If the DB library finds that the value assigned to the DB path is of the form '`@`*file*', it replaces that value with the contents of the specified *file*. (This feature was first introduced in DB library version 8.0.) Indirect DB 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 on the Macintosh, where recompilation of the DB library would otherwise be necessary in order to change the DB 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 `DB` environment variable would otherwise approach or exceed the 128-byte limit for MS-DOS commands.

If a DB 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 DB path, that directory is appended to the end of the DB path; in this case, if the DB 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 DB library version 6.2.)

The string '`%r`' is replaced by the current record name wherever it appears in the DB 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 DB path is '`:/cdrom/mimicdb/%3r:/cdrom/mitdb`', a request to read a file associated with record 055n will cause the DB library to look first in the current directory (since the DB 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 DB path; thus a literal '`%`' can be included in the DB path by 'escaping' it as '`%%`'. (Substitutions of '`%`'-strings in the DB path were first introduced in DB library version 9.7.)

Before the example program can be run successfully, `DB` must be set appropriately and placed in the environment where it can be examined by the running program. If you use the Bourne shell (`sh`), the GNU Bourne Again shell (`bash`), or the Korn shell (`ksh`), do so by typing

```
. setdb
```

(or by placing this command in your '`.profile`', which will cause `DB` to be set automatically whenever you log in). If you use the C-shell (`csh`) or a variant of it (such as `ecsh` or `tcsh`), type

```
source /usr/local/bin/cshsetdb
```

(or put this command in your '`.cshrc`'). These shell scripts should be available on any system on which the DB library itself is available, although they may not be in '`/usr/local/bin`' as shown above. If you cannot find the script you need, consult an expert (such as the person who installed the DB library on your system). Note that the default value for `DB` obtained in this way begins with the current directory, and includes all standard directories in which database records are kept. Thus, if you create your own database records or files, you may keep them in your own directory without special arrangements.

Under MS-DOS, you may set `DB` using the batch file '`dossetdb.bat`' (created during installation of the DB Software Package), or you may include the contents of that file in your '`autoexec.bat`' file, which will cause `DB` to be set automatically whenever your system is rebooted. Since environment space under MS-DOS is an operating system resource and is often in very short supply, you may need to allocate additional environment space within your '`config.sys`' file, with a line of the form

```
shell=c:\command.com /p /e:512
```

See your MS-DOS manual for further information.

These shell scripts usually set the `DBCAL` environment variable, which is important if you make use of records that contain signals other than ECGs. `DBCAL` names a *calibration file* located in one of the directories named by `DB`. (Beginning with DB library version 8.3, the symbol `DEFDBC` may be defined in '`dblib.h`' to specify the name of a default calibration file, to be used by the DB library if `DBCAL` 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.

1.4 Running the Example Program

If `DB` 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
```

and 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.

1.5 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 DB 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 '`db_`' are reserved for the use of the DB library. These names are used for functions, global variables that are intended for the private use of the DB 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 '`DB_`' are used for constants and data types defined within '`<ecg/db.h>`'. Use these identifiers as needed in your programs, but avoid redefining them.

1.6 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 DB library.

1. Find out where database records are kept on your system. What records are available?
2. Run the example program in this chapter *without* setting the DB path. Try doing so from your home directory, and from the directory where record '`100s`' is kept. Can you explain the results?
3. What happens when you compile the example program as shown, but with the `#include` statement omitted? with the '`-ldb`' ('`-link db`', etc.) omitted?
4. What is the type of the argument to `getvec`? Why can't `getvec` simply return the value it reads, as in '`v = getvec()`'?

2 DB Library Functions

This chapter describes the functions that are available to programs compiled with the '`-ldb`' option. 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, DB 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 [`dbquiet` and `dbverbose`], page 47). 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 DB library functions can find them given only the record (and annotator) names, provided that `DB` has been properly set (see Section 1.3 [DB path], page 13). DB library functions responsible for opening signal files find them by reading the '`header`' file (which contains their names) first.

The first part of this chapter describes functions that extract information from '`header`' files in order to gain access to signal and annotation files. The following two sections describe functions that deal with 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.

2.1 Selecting Database Records

annopen

`int annopen(char *`*record*`, DB_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

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 `DB_Anninfo` structures (see Section 3.3 [Annotator Information Structures], page 63), one for each annotator to be opened. *nann* is the number of `DB_Anninfo` structures in *aiarray*; *nann* must be no greater than twice `DB_MAXANN` (a constant defined in '`<ecg/db.h>`'), and no more than `DB_MAXANN` input and `DB_MAXANN` output annotators may be open at once. The caller must fill in the `DB_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";
DB_Anninfo a[3];

a[0].name = "a"; a[0].stat = READ;
a[1].name = "b"; a[1].stat = WRITE;
a[2].name = "c"; a[2].stat = 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 28) 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 DB (see Section 1.3 [DB path], page 13); output annotators are created in the current directory (but note that, under UNIX at least, it is possible to specify annotator names such as '/here' or 'zzz/there' or even '../somewhere/else'; see [Annotation Files], page 72, for details of how file names are constructed from annotator and record names). Several of the example programs in chapter 6 illustrate the use of annopen; for example, see [Example 1], page 81.

As a special case, if *nann* is 0, *aiarray* can be NULL. This can be useful to force open annotation files to be closed without closing open signal files.

isigopen

`int isigopen(char *record, DB_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

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 'header' file. *Siarray* is a pointer to an array of DB_Siginfo structures (see Section 3.1 [Signal Information Structures], page 60), one for each signal to be opened. The number of DB_Siginfo structures in *siarray*, *nsig*, need never be greater than DB_MAXSIG (a constant defined in '<ecg/db.h>'); 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 `DB_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 '`header`' 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:

```
...
static DB_Siginfo s[DB_MAXSIG];
int i, nsig;

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

Notice that the `for` loop is never executed if `isigopen` encounters an error, since `nsig` is less than 1 in such cases. The array of `DB_Siginfo` structures is declared `static`, a practice that is recommended for all arrays and data structures of any substantial size, to avoid run-time stack overflow.

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* or `DB_MAXSIG`. (On existing CD-ROMs containing database records, there is only one signal group for each record; as a consequence, `isigopen` fails if *nsig* is less than the number of available signals while attempting to read a CD-ROM record.) 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 60. Several of the example programs in chapter 6 illustrate the use of `isigopen`; for example, see [Example 5], page 86.

If *nsig* is not positive, `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 '`header`' file. Note, however, that there is no guarantee that all (or indeed any) of the signals named in the '`header`' file are available to be opened. The features described in this paragraph were first introduced in version 4.4 of the DB library.

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

osigopen

```
int osigopen(char *record, DB_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 '`header`' 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 '`header`' file. *siarray* is a pointer to an uninitialized array of `DB_Siginfo` structures; *siarray* must contain at least *nsig* members. The caller must allocate storage for the `DB_Siginfo` structures. On return, `osigopen` will have filled in the `DB_Siginfo` structures with the signal specifications.

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

```
...
DB_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 77). See [Example 6], page 88, and see [Example 7], page 90, 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(DB_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)

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 `DB_Siginfo` structures (see Section 3.1 [Signal Information Structures], page 60), one for each signal to be opened. *nsig* is the number of `DB_Siginfo` structures in *siarray*; *nsig* must be no greater than `DB_MAXSIG` (a constant defined in '`<ecg/db.h>`').

Before invoking `osigfopen`, the caller must fill in the fields of the `DB_Siginfo` structures in *siarray* (see Chapter 3 [Data Types], page 59; 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 74). For ordinary (non-multiplexed) signal files, specify a unique `fname` and `group` for each signal. See [Example 8], page 93, for an illustration of the use of `osigfopen`.

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.

dbinit

```
int dbinit(char *record, DB_Anninfo *aiarray, unsigned int nann,
```

```
                DB_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 63)

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

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

is exactly equivalent to

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

See [Example 9], page 96, for an illustration of the use of `dbinit`. See [`osigopen`], page 23, and see [`osigfopen`], page 24, for methods of opening output signal files.

2.2 Reading and Writing Signals and Annotations

getvec

`int getvec(DB_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 DB_Samples (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 `dbinit`; this number will never exceed `DB_MAXSIG`, defined in '`<ecg/db.h>`'.) 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 <ecg/db.h>

main()
{
    int i, j, nsig;
    DB_Sample v[DB_MAXSIG];
    static DB_Siginfo s[DB_MAXSIG];

    nsig = isigopen("100s", s, DB_MAXSIG);
    if (nsig < 1)
        exit(1);
    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);
```

```
}
```

Notice how the value returned by `isigopen` is used to determine how many input signals there are. The array of `DB_Siginfo` structures is declared `static`, a practice that is recommended for all arrays and data structures of any substantial size, to avoid run-time stack overflow. Several of the example programs in chapter 6 illustrate the use of `getvec`; for example, see [Example 6], page 88.

getframe

`int getframe(DB_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 DB_Samples (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 `DB_Siginfo` structures associated with the input signals (see [`isigopen`], page 21).

putvec

`int putvec(DB_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 DB_Samples 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`; this number must never exceed `DB_MAXSIG`, defined in '`<ecg/db.h>`'.) On entry, *vector[i]* contains the next sample from signal *i*. For example, this modified version of the previous example (see [`getvec`], page 26) copies the first ten samples of each available input signal:

```
#include <ecg/db.h>

main()
{
    int i, nsig;
    DB_Sample v[DB_MAXSIG];
    static DB_Siginfo s[DB_MAXSIG];

    if ((nsig = isigopen("100s", s, DB_MAXSIG)) < 1 ||
        osigopen("8l", s, nsig) < 1)
        exit(1);
    for (i = 0; i < 10; i++)
        if (getvec(v) < 0 || putvec(v) < 0)
            break;
    dbquit();
    exit(0);
}
```

All programs that write signals or annotations *must* invoke `dbquit` to close the output files properly (see [`dbquit`], page 46). This example uses record '`8l`' (see Section 5.8 [Piped and Local Records], page 77) for the output signal specifications; the output signal files will be named '`data0`' and '`data1`' in the current directory. The array of `DB_Siginfo` structures is declared `static`, a practice that is recommended for all arrays and data structures of any substantial size, to avoid run-time stack overflow. Several of the example programs in chapter 6 illustrate the use of `putvec`; for example, see [Example 6], page 88.

getann

`int getann(DB_Annotator` *an*`, DB_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 64) 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 ('`atruth`') annotation file for record '`100s`':

```
#include <ecg/db.h>

main()
{
    DB_Anninfo a;
    DB_Annotation annot;

    a.name = "atruth"; a.stat = 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 Section 3.3 [Annotator Information Structures], page 63, for information on the contents of the `DB_Anninfo` structure, and see [`mstimstr`], page 35, and see [`annstr`], page 33, for details of the functions used to print portions of the annotations read by `getann` in this example.

ungetann

`int ungetann(DB_Annotator` *an*`, DB_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 DB library version 5.3.)

putann

`int putann(DB_Annotator` *an*`, DB_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 DB library, annotations may be written in any order (see Section 5.9 [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 `dbquit` to close the output files properly (see [`dbquit`], page 46). Several of the example programs in chapter 6 illustrate the use of `putann`; for example, see [Example 1], page 81.

2.3 Non-Sequential Access to DB 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(DB_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 35). This function will fail if a pipe is used for input and `|`*t*`|` is less than the current sample number. See [Example 7], page 90, and see [Example 9], page 96, for illustrations of the use of `isigsettime`.

isgsettime

```
int isgsettime(DB_Group sgroup, DB_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 76).

iannsettime

```
int iannsettime(DB_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 35). 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`.

The version of `iannsettime` that is included on the first edition of the MIT-BIH Arrhythmia Database CD-ROM contains a bug that can occasionally cause incorrect results. To avoid this problem when using that version of the library, make at least one call to `getann` before using `iannsettime`. The bug was corrected in version 4.0 of the DB library.

2.4 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 33). 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 <ecg/db.h>
#include <ecg/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");
        }
    }
```

`ACMAX` is defined in '`<ecg/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 DB library version 5.3.)

strann and strecg

`int strann(char *`*string*`)`
`int strecg(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 DB library version 5.3.)

setannstr, setanndesc, and setecgstr

`int setannstr(int `*code*`, char *`*string*`)`
`int setanndesc(int`*code*`, char *`*string*`)`
`int setecgstr(int `*code*`, 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 '`<ecg/ecgcodes.h>`'). These functions do not copy their string arguments, which must therefore be kept valid by the caller.

An important difference between `setannstr` (or `setanndesc`) and `setecgstr` is that `annopen` and `dbinit` 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 DB application may create annotation files with self-contained code tables, which can be read properly by other DB 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 DB 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`, `dbinit`, 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 `dbquit`, 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(DB_Time t)
```

```
char *mstimstr(DB_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 '`header`' file or the caller (see [`setbasetime`], page 49); 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

```
DB_Time strtim(char *string)
```

Return:

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

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

`(DB_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 49); 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 50) 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 90).

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(DB_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

`DB_Date strdat(char *`*string*`)`

Return:

`(DB_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 `dbinit`, either of which will determine the gain from the '`header`' 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 `DEFGAIN` (defined in '`<ecg/db.h>`') ADC units per millivolt, and a baseline of zero, are assumed. If the physical units (see Section 3.1 [Signal Information Structures], page 60) 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(DB_Signal s, DB_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 DB library version 6.0.)

physadu

```
DB_Sample physadu(DB_Signal s, double v)
```

Return:

`(DB_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 DB library version 6.0.)

adumuv

```
int adumuv(DB_Signal s, DB_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

```
DB_Sample muvadu(DB_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*.

2.5 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(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 `DB`, see Section 1.3 [DB path], page 13) into the calibration list. If *file* is `NULL`, the file named by `DBCAL` 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 DB library version 6.0.)

getcal

`int getcal(char *`*desc*`, char *`*units*`, DB_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 `DB_Calinfo` structure (see Section 3.2

[Calibration Information Structures], page 62) pointed to by *cal*. The caller must allocate storage for the `DB_Calinfo` structure, and must not modify the contents of the strings addressed by the `sigtype` and `units` fields of the `DB_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 DB library version 6.0.)

putcal

`int putcal(DB_Calinfo *`*cal*`)`

Return:

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

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

newcal

`int newcal(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 DB file name, i.e., it may contain letters, digits, or underscores. (This function was first introduced in DB 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 `dbquit` does *not* perform the function of `flushcal`. (This function was first introduced in DB library version 6.0.)

2.6 Miscellaneous DB Functions

newheader

`int newheader(char *record)`

Return:

`0`	Success
`-1`	Failure: unable to create '`header`' file

This function creates a '`header`' file (in the current directory). Use `newheader` just after you have finished writing the signal files, but before calling `dbquit`. If *record* begins with '`+`', the '`+`' is discarded and the remainder of *record* is taken as the record name. Otherwise, all of *record* 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 93, 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.

setheader

`int setheader(char *record, DB_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. In the following code fragment, the '`header`' file for record '`old`' is used to create a '`header`' file for record '`new`':

```
...
int nsig, status;
DB_Siginfo s[DB_MAXSIG];

nsig = isigopen("old", s, -DB_MAXSIG);
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 '`header`' file for record '`old`' must be copied explicitly; see [`getinfo`], page 54, and see [`putinfo`], page 54. (This function was first introduced in DB library version 5.0.)

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 75. *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 DB application can read the concatenated signal files of the constituent segment simply by opening the multi-segment record (using `isigopen` or `dbinit`). Note that the signal files themselves are not modified in any way, nor are they copied; rather, the other DB 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/dbcollate.c`' ('`app/dbcoll8.c`' under MS-DOS) in the DB Software Package. (This function was first introduced in DB library version 9.1.)

dbquit

```
void dbquit(void)
```

This function closes all open database files. It also resets the following:

- the factor used for converting between samples and seconds (reset to 1), and the base time (reset to 0, i.e., midnight); see [`timstr` and `strtim`], page 35
- the parameters used for converting between adus and physical units (reset to `DEFGAIN` adu/mV, a quantity defined in '`<ecg/db.h>`'); see [`aduphys` and `physadu`], page 39
- internal variables used to determine output signal specifications; see [`newheader`], page 44.

If any annotations have been written out-of-order (see Section 5.9 [Annotation Order], page 78), this function attempts to run '`sortann`' (see the *ECG Database 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 `dbquit`.

iannclose

```
void iannclose(void)
```

This function closes all input annotation files. It was first introduced in DB library version 9.1.

oannclose

```
void oannclose(void)
```

This function closes all output annotation files. It was first introduced in DB library version 9.1.

If any annotations have been written out-of-order (see Section 5.9 [Annotation Order], page 78), this function attempts to run '`sortann`' (see the *ECG Database 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.

dbquiet

```
void dbquiet(void)
```

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

dbverbose

```
void dbverbose(void)
```

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

dberror

```
char *dberror(void)
```

Return:

`(char *)` pointer to error string

This function returns a pointer to a string containing the text of the most recent DB library error message (or to a string containing the DB library version number, if there have been no errors). Function `dberror` 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 DB library generates a message box for error messages, unless `dbquiet` has been used to silence them.) This function was first introduced in DB 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.

sampfreq

```
DB_Frequency sampfreq(char *record)
```

Return:

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

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

`(DB_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 35) 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 49).

setsampfreq

```
int setsampfreq(DB_Frequency freq)
```

Return:

`0`
: Success

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

This function sets the sampling frequency used by the time-conversion functions (see [timstr and strtim], page 35). Use setsampfreq before creating a new 'header' file (see [newheader], page 44). See [Example 8], page 93, 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 (see [newheader], page 44). See [Example 8], page 93, for an illustration of the use of setbasetime.

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 74). If *mode* is DB_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 DB_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 DB_LOWRES mode by default. DB_LOWRES and DB_HIGHRES are defined in '<ecg/db.h>'.

In DB library version 9.6 and later versions, `setgvmode` also affects how annotations are read and written. If `setgvmode(DB_HIGHRES)` is invoked *before* using `annopen`, `dbinit`, `getvec`, `sampfreq`, `strtim`, or `timstr`, then all `DB_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, `DB_Time` data are in units of frame intervals.)

getspf

```
int getspf(void)
```

Return:

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

Unless the application has used `setgvmode(DB_HIGHRES)` 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).

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 DB 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 DB library version 5.2, are used to obtain or set the values of these parameters.

getcfreq

```
DB_Frequency getcfreq(void)
```

Return:

`(DB_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(DB_Frequency freq)
```

This function sets the counter frequency. Use `setcfreq` before creating a '`header`' file (see [`newheader`], page 44). 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 44). The effect of `setbasecount` is nullified by later invoking any of the functions that read '`header`' files.

setdb

```
void setdb(char *string)
```

This function may be used to set or change the database path (see Section 1.3 [DB path], page 13) 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 directory names separated by colons ('`:`') under UNIX or by semicolons ('`;`') under MS-DOS. An empty component, indicated by an initial or terminal separator, or by two consecutive separators, will be understood to specify the current directory. 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 DB path is to be read. This file may contain either the DB path, as described in the previous paragraph, or another indirect DB path specification. Indirect DB path specifications may be nested no more than ten levels deep (an arbitrary limit imposed to avoid infinite recursion). Evaluation of indirect DB paths is deferred until `getdb` is invoked, either explicitly or by the DB library while attempting to open an input file (e.g., using `annopen` or `isigopen`). (The features described in this paragraph were first introduced in DB library version 8.0.)

The `setdb` function does not copy the string itself, which must therefore be kept valid by the caller, nor does it modify the environment variable `DB`, which will be inherited by any processes spawned from the caller. For portability, as well as efficiency, it is better to use `setdb` than to manipulate `DB` directly (via `putenv`, for example); furthermore, doing the latter is ineffective (within the current process) once the first DB file has been opened. See [`getdb`], page 52, for an example of the use of `setdb`.

getdb

```
char *getdb(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 = getdb();
if (newp = malloc(strlen("/usr/mydb:") + strlen(oldp) + 1)) {
    sprintf(newp, "/usr/mydb:%s", oldp);
    setdb(newp);
}
...
```

adds the directory '`/usr/mydb`' to the beginning of the database path. (To perform a similar operation under MS-DOS, replace the colons by semicolons. 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 '\\'.)

dbfile

`char *dbfile(char *`*type*`, char *`*record*`)`

Return:

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

This function attempts to locate an existing DB file by searching the database path (see Section 1.3 [DB path], page 13). Normally, the file is specified by its *type* (e.g., '`header`', or an annotator name such as '`atruth`') 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 `dbfile` if the name is passed in the *type* variable and *record* is `NULL`. The string returned by `dbfile` 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 file cannot be found, `dbfile` returns `NULL`. (This function was first introduced in DB library version 4.3.)

dbflush

```
void dbflush(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`

This function reads an "info" string from the '`header`' file for the specified *record*. Info strings are null-terminated and do not contain newline characters. The '`header`' files of some records may contain no info strings; others may contain more than one info string. To read additional info strings after the first, use `getinfo(NULL)`. 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));
...
```

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

putinfo

int putinfo(char **s*)

Return:

`0` Success

`-1` Failure: `header` not initialized

This function writes *s* as an "info" string into the '`header`' file that was created by the most recent invocation of `newheader`. 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` by successive invocations of `putinfo`. Note that `newheader` or `setheader` must be used before `putinfo`. (This function was first introduced in DB library version 4.0.)

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 `dbinit` and before invoking `dbquit`). 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 `DB_Siginfo` structure (see Section 3.1 [Signal Information Structures], page 60) is non-zero. (This function was first introduced in DB 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 `dbquit`). 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 '`header`' file read by `osigopen`, or in the `bsize` field of the `DB_Siginfo` structure (see Section 3.1 [Signal Information Structures], page 60) passed to `osigfopen`. (This function was first introduced in DB library version 5.0.)

dbgetskew

`int dbgetskew(DB_Signal` *s*`)`

Return:

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

This function returns the *skew* (as recorded in the '`header`' 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, `dbgetskew` is useful primarily for programs that need to rewrite existing '`header`' 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;
DB_Signal s;
static DB_Siginfo siarray[DB_MAXSIG];

...

if ((nsig = isigopen(record, siarray, DB_MAXSIG)) < 1)
    exit(1);
for (s = 0; s < nsig; s++) {
    dbsetskew(s, dbgetskew(s));
    dbsetstart(s, dbgetstart(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. `dbgetskew` 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 DB library version 9.4.)

dbsetskew

```
void dbsetskew(DB_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 `dbsetskew`, see [dbgetskew], page 56. Note that `dbsetskew` 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 DB library version 9.4.)

dbgetstart

```
long dbgetstart(DB_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 `dbgetstart` 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 '`header`' file. Since the prolog is not readable using the DB library, and since functions such as `isigopen` and `isigsettime` take the prolog into account when calculating byte offsets for `getframe` and `getvec`, `dbgetstart` is useful primarily for programs that need to rewrite existing '`header`' files, where it is necessary to preserve the previously recorded byte offsets. For an example of how this can be done, see [dbgetskew], page 56. (This function was first introduced in DB library version 9.4.)

dbsetstart

`void dbsetstart(DB_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 `dbsetstart`, see [dbgetskew], page 56. Note that `dbsetstart` 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 '`header`' file at the time the signals were opened. (This function was first introduced in DB library version 9.4.)

3 Data Types

Simple data types used by the DB library are defined in '`<ecg/db.h>`'. These include:

`DB_Sample`
: a signed integer type (at least 16 bits) used to represent sample values, in units of adus.

`DB_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 `DB_Time` variable indicates how it is to be printed by `timstr` or `mstimstr`.

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

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

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

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

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

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

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

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

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

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

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

Until DB library version 8.2, these types were known as `struct siginfo`, `struct calinfo`, `struct anninfo`, and `struct ann` respectively. Except under Solaris (which has its own definition for `siginfo`), these older names are still defined in '`<ecg/db.h>`', but their use is discouraged.

3.1 Signal Information Structures

The *siarray* argument for `isigopen`, `osigopen`, `dbinit`, and `osigfopen` is a pointer to an array of objects of type `DB_Siginfo`. The first three of these functions fill in the `DB_Siginfo` objects to which *siarray* points, but the caller must supply initialized `DB_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 1.3 [DB path], page 13). `fname` may include relative or absolute path specifications if necessary; the use of an absolute pathname, combined with an initial null component in `DB`, 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 `DB_MAXDSL` (defined in '`<ecg/db.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 `DB_MAXUSL` (defined in '`<ecg/db.h>`') characters (not including the null).

`DB_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 `DEFGAIN` (defined in '`<ecg/db.h>`') may be assumed.

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

`DB_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 `DB_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 '`<ecg/db.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 between 2 and `DB_MAXSPF` (defined in '`<ecg/db.h>`'). 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. All signals in a given record must have the same number of samples. 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 '`header`' 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 `DB_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. There may be no more than `DB_MAXSIG` (defined in '`<ecg/db.h>`') input signals and `DB_MAXSIG` output signals open at once. See [Example 5], page 86, for an illustration of how to read signal specifications from `DB_Siginfo` structures.

3.2 Calibration Information Structures

The *cal* argument for `getcal` and `putcal` is a pointer to an object of type `DB_Calinfo`. A `DB_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 `DB_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 `DB_Calinfo` structure must exactly match the `units` field of the `DB_Siginfo` structure that describes a matching signal.

`double scale`
: the customary plotting scale, in physical units per centimeter. DB 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 `scales`.

`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 '`<ecg/db.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 `DB_Calinfo` structures. It is accessible only via `calopen`, `getcal`, `putcal`, `newcal`, and `flushcal`.

3.3 Annotator Information Structures

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

`char *name`
: the annotator name. The name '`atruth`' 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`', '`data`*n*', '`d`*n*', or '`header`' (all of which are commonly used as parts of DB 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 1.3 [DB path], page 13). Under UNIX, it is possible to create or find annotation files in any accessible directory by specifying annotator names that begin with absolute or relative directory pathnames (note that such prefixes are exempt from the restriction on legal characters in annotator names). Under MS-DOS, this is not possible; furthermore, note that the only the first three characters of the annotator name are used to construct the MS-DOS file name, so be careful to keep annotator names distinct from each other and from commonly used MS-DOS file name "extensions".

`int stat`
: the file type/access code. Usually, `stat` is either `READ` or `WRITE`, to specify standard ("MIT format") annotation files to be read by `getann` or to be written by `putann`. Both MIT DB and AHA DB annotation files are kept on-line in MIT format. The symbols `READ` and `WRITE` are defined in '`<ecg/db.h>`'. An AHA-format annotation file can be read by `getann` or written by `putann` if the `stat` field is set to `AHA_READ` or `AHA_WRITE` before calling `annopen` or `dbinit` (see [Example 2], page 83). Other formats may be supported via a similar mechanism; consult '`<ecg/db.h>`' for more information.

The number of `DB_Anninfo` objects in *aiarray* is given by the *nann* argument of `annopen` and `dbinit`. The annotation-reading function, `getann`, knows the annotators by number only; `annopen` and `dbinit` assign input annotator numbers beginning with 0 in the order in which they are given in the array of DB_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. There may be no more than DB_MAXANN (defined in '<ecg/db.h>') input annotators and DB_MAXANN output annotators open at once. See [annopen], page 20, for an example of how to set the contents of an array of DB_Anninfo objects.

3.4 Annotation Structures

The *annot* argument of getann and putann is an object of type DB_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 'atruth' 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 (defined in '<ecg/ecgcodes.h>'). See Chapter 4 [Annotation Codes], page 67, for a list of legal annotation codes.

signed char subtyp
signed char chan
signed char num
: numbers between −128 and 127. In MIT DB 'atruth' 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 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 'atruth' 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 DB 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 DB_Annotation.

4 Annotation Codes

Application programs that deal with annotations should include the line

```
#include <ecg/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	P	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 DB library version 4.0. Codes `SYSTOLE`, `DIASTOLE`, and `MEASURE` were first introduced in DB library version 7.0.
2. In MIT and ESC DB '`atruth`' 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 '`<ecg/ecgcodes.h>`', but are identical to `WFON` and `WFOFF` respectively.
5. The `LINK` code was first introduced in DB 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 DB browser such as `WAVE`.

The annotation codes in the table above are the predefined values of the `anntyp` field in a `DB_Annotation`. Other values in the range of 1 to `ACMAX` (defined in '`<ecg/ecgcodes.h>`') are legal but do not have preassigned meanings. The constant `NOTQRS`, also defined in '`<ecg/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 <ecg/ecgmap.h>
```

which will make their definitions, and those in '`<ecg/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 '`<ecg/ecgmap.h>`' for definitions of these symbols; this macro was first introduced in DB 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 '`<ecg/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 DB library version 6.0.)

The macros below convert between AHA and MIT annotation codes; they are also defined in '`<ecg/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 DB 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 `DB` (see Section 1.3 [DB path], page 13).

Recall that a DB 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 CD-ROM, 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 in their file names, either as the first part of the name (the default on writable UNIX and Macintosh file systems) or as the suffix or extension (on CD-ROM and MS-DOS file systems). Applications that use the DB library under UNIX or the Macintosh OS can locate files named using either of these conventions. No explicit action (other than choosing the file name, and locating the file in the DB path) is needed in order to associate a new file with an existing DB record.

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

5.1 File Types

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

Header Files

Header files have names of the form '`header.`*record*' on writable UNIX and Macintosh file systems, 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.) On CD-ROM and MS-DOS file systems, because of restrictions on the length of extensions, header files have names of the form '*record*`.HEA`'. 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 `dbinit` read the names of the signal files and their attributes as given in the array of `DB_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 '`<ecg/db.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 '*annotator*`.`*record*' on writable UNIX and Macintosh file systems (on CD-ROM and MS-DOS file systems, '*record*`.`*annotator*'). Those named '`atruth.`*record*' ('*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 62). The DB Software Package includes a standard calibration file, '`dbcal`', in the '`udb`' directory.

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 74), 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 `DB_Anninfo` object is set to `AHA_READ` or `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. This format is compatible with neither the original AHA format nor with any of the formats supported directly by the DB library. Programs '`a2m`' and '`ad2m`', supplied with the DB 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 '`header`' file is taken to be the standard input or output, as appropriate. If the name of a signal file is specified in the '`header`' file (or in the array of `DB_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 `DB_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 DB 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 `DB_Siginfo` objects returned by `isigopen` or `dbinit`. 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 74).

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). CD-ROM signal files 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 DB 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 `DB_Siginfo` structure for the signal.

A frame can be read as it was written (see [`getframe`], page 27) 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 DB library version 9.6 and later versions), all time quantities passed to and from the DB 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, with only a one-line change in the source.

5.5 Multi-Segment Records

A multi-segment record consists of two or more concatenated segments. Each segment is an ordinary DB record, with its own header file and signal file(s). In any given multi-segment record, all signals must appear in the same order within each segment (signals may not be omitted), and 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 DB 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 DB 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 DB 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. Version 9.1 of the DB library is the first to support reading and writing multi-segment records.

5.6 Simultaneous Access to Multiple Records

Selection functions that accept *record* arguments (`annopen`, `isigopen`, `osigopen`, and `dbinit`) 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 `dbquiet` was invoked). The time-conversion functions (see [`timstr` and `strtim`], page 35) 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 `DB_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 `DB_Siginfo` object must contain the appropriate block size in bytes. In such cases, the DB 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 77) 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. The names of most of these records ('`512`', '`1024`', '`4096`', '`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 '`b`' uses 8-bit difference format, 6144 bytes/block, and is otherwise similar to the others. Records '`ahatape`' and '`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 also exist; they are named '`8x`*n*' and '`16x`*n*', where n is the number of signals (n = 1, 2, ..., `DB_MAXSIG`).

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 `DB_MAXSIG` format 16 files, and record '`8l`' ("eight-ell") specifies up to `DB_MAXSIG` format 8 files, named '`data0`', '`data1`', '`data2`', ..., '`data`*n*' in the current directory. When opened using `isigopen` or `dbinit`, these signal files will be readable by `getvec` as signals 0, 1, 2, ... n 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 Annotation Order

DB 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. Early versions of the DB library (before version 6.2) defined canonical order as time order. More recent versions of the DB library define canonical order as `time` and `chan` order (thus annotations are arranged first in `time` order, and any simultaneous annotations are arranged according to the value of their `chan` fields, from smallest to largest).

The combination of the `time` 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 DB library earlier than version 9.7. In these versions of the DB 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 DB 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 `dbquit` or `oannclose`. This is accomplished by running '`sortann`' (see the *ECG Database Applications Guide*) as a separate process using the ANSI C `system` function. If this function is not available, or if '`sortann`' cannot be run, `dbquit` (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 DB 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`, `num`, or `aux` fields of a previously-written annotation simply by writing another annotation to the same location (i.e, with the same `time` 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` 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 `DBNOSORT` (to any value) at run time, or define the symbol DBNOSORT when compiling the DB library, if you wish to suppress automatic annotation sorting by dbquit and oannclose.

6 Programming Examples

The programs in this chapter are useful as models for a variety of applications that use the DB 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* `-ldb`

where *file.c* is the name of the file containing the source; see Chapter 1 [Using the DB Library], page 11, for further information. The sources for these examples are included in the DB 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.

```
#include <stdio.h>
#include <ecg/db.h>
#include <ecg/ecgmap.h>

main()
{
    DB_Anninfo an[2];
    char record[8], iann[10], oann[10];
    DB_Annotation annot;

    printf("Type record name: ");
    fgets(record, 8, stdin); record[strlen(record)-1] = '\0';
    printf("Type input annotator name: ");
    fgets(iann, 10, stdin); iann[strlen(iann)-1] = '\0';
    printf("Type output annotator name: ");
    fgets(oann, 10, stdin); oann[strlen(oann)-1] = '\0';
    an[0].name = iann; an[0].stat = READ;
    an[1].name = oann; an[1].stat = WRITE;
    if (annopen(record, an, 2) < 0) exit(1);
    while (getann(0, &annot) == 0)
        if (isqrs(annot.anntyp)) {
            annot.anntyp = NORMAL;
            if (putann(0, &annot) < 0) break;
        }
    dbquit();
```

```
26  }
```

Notes:

Line 2: All programs that use the DB library must include ‘<ecg/db.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 ‘<ecg/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 DB_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 DB 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 DB_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. READ and WRITE are defined in ‘<ecg/db.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 ‘<ecg/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 ‘<ecg/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 '`atruth`' 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 <ecg/db.h>
3
4  main(argc, argv)
5  int argc;
6  char *argv[];
7  {
8      DB_Anninfo an[2];
9      DB_Annotation annot;
10
11     if (argc < 2) {
12         fprintf(stderr, "usage: %s record\n", argv[0]);
13         exit(1);
14     }
15     an[0].name = "atruth"; an[0].stat = READ;
16     an[1].name = "aha";    an[1].stat = AHA_WRITE;
17     if (annopen(argv[1], an, 2) < 0) exit(2);
18     while (getann(0, &annot) == 0 && putann(0, &annot) == 0)
19         ;
20     dbquit();
21     exit(0);
22 }
```

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 '`atruth`' 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 <ecg/db.h>
3
4  main(argc, argv)
5  int argc;
6  char *argv[];
7  {
8      DB_Anninfo a;
9      DB_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 = 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 (%ld) %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                 annot.aux+1 : "");
26     exit(0);
27 }
```

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.

```
#include <stdio.h>
#include <ecg/db.h>
#include <ecg/ecgmap.h>

main(argc, argv)
int argc;
char *argv[];
{
    int rr, *rrhist, rrmax;
    long t;
    DB_Anninfo a;
    DB_Annotation annot;
    void *calloc();

    if (argc < 3) {
        fprintf(stderr, "usage: %s annotator record\n", argv[0]);
        exit(1);
    }
    a.name = argv[1]; a.stat = READ;
    if (annopen(argv[2], &a, 1) < 0) exit(2);
    if ((rrmax = (int)(3*sampfreq(argv[2]))) <= 0) exit(3);
    if ((rrhist = (int *)calloc(rrmax+1, sizeof(int))) == NULL) {
        fprintf(stderr, "%s: insufficient memory\n", argv[0]);
        exit(4);
    }
    while (getann(0, &annot) == 0 && !isqrs(annot.anntyp))
        ;
    t = annot.time;
    while (getann(0, &annot) == 0)
        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((long)rr));
37          printf("%4d %s (or longer)\n", rrhist[rr], mstimstr((long)rr));
38          exit(0);
39      }
```

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:

```
#include <stdio.h>
#include <ecg/db.h>

main(argc, argv)
int argc;
char *argv[];
```

```
7  {
8      static DB_Siginfo s[DB_MAXSIG];
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], s, DB_MAXSIG);
16     if (nsig < 1) exit(2);
17     printf("Record %s\n", argv[1]);
18     printf("Starting time: %s\n", timstr(0L));
19     printf("Sampling frequency: %g Hz\n", sampfreq(argv[1]));
20     printf("%d signals\n", nsig);
21     for (i = 0; i < nsig; i++) {
22         printf("Group %d, Signal %d:\n", s[i].group, i);
23         printf(" File: %s\n", s[i].fname);
24         printf(" Description: %s\n", s[i].desc);
25         printf(" Gain: ");
26         if (s[i].gain == 0.)
27             printf("uncalibrated; assume %g", DEFGAIN);
28         else printf("%g", s[i].gain);
29         printf(" adu/%s\n", s[i].units ? s[i].units : "mV");
30         printf(" Initial value: %d\n", s[i].initval);
31         printf(" Storage format: %d\n", s[i].fmt);
32         printf(" I/O: ");
33         if (s[i].bsize == 0) printf("can be unbuffered\n");
34         else printf("%d-byte blocks\n", s[i].bsize);
35         printf(" ADC resolution: %d bits\n", s[i].adcres);
36         printf(" ADC zero: %d\n", s[i].adczero);
37         if (s[i].nsamp > 0L) {
38             printf(" Length: %s (%ld sample intervals)\n",
39                    timstr(s[i].nsamp), s[i].nsamp);
40             printf(" Checksum: %d\n", s[i].cksum);
41         }
42         else printf(" Length undefined\n");
43     }
44     exit(0);
45 }
```

Notes:

Line 8: There might be as many as `DB_MAXSIG` signals, so we allocate enough `DB_Siginfo` objects to accommodate them all.

Line 15: The command-line argument, `argv[1]`, is the record name. The number of readable signals is `nsig`. This program will give specifications for each of them, but not for any signals that are named in the '`header`' file but are not readable. If `nsig < 1`, `isigopen` will print an error message; in this case the program can't do anything useful, so it exits.

Line 18: 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 25–28:
Notice how a zero value for `gain` is interpreted.

Line 29: If the `units` field is NULL, the physical units are assumed to be millivolts ("mV").

Lines 32–34:
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 37–42:
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 90, for a more general approach.

```
#include <stdio.h>
#include <ecg/db.h>

main(argc, argv)
int argc;
char *argv[];
{
    static DB_Siginfo s[DB_MAXSIG];
    int i, nsig, nsamp = 1000;
    DB_Sample vin[DB_MAXSIG], vout[DB_MAXSIG];

    if (argc < 2) {
        fprintf(stderr, "usage: %s record\n", argv[0]); exit(1);
    }
    if ((nsig = isigopen(argv[1], s, DB_MAXSIG)) <= 0) exit(2);
```

```
16      if (osigopen("8l", s, nsig) <= 0) exit(3);
17      while (nsamp-- > 0 && getvec(vin) > 0) {
18          for (i = 0; i < nsig; i++)
19              vout[i] -= vin[i];
20          if (putvec(vout) < 0) break;
21          for (i = 0; i < nsig; i++)
22              vout[i] = vin[i];
23      }
24      (void)newheader("dif");
25      dbquit();
26      exit(0);
27  }
```

Notes:

Line 16: 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 17: 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 18–19:
For each signal, we compute the negated first difference by subtracting the new sample from the previous sample.

Line 20: One sample of each output signal is written here.

Lines 21–22:
The new input samples are copied into the output sample vector in preparation for the next iteration.

Line 24: 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 '`8l`', we can also read these files using record '`8l`'; one reason for making a new '`header`' file here is that the '`header`' file for '`8l`' may not necessarily indicate the proper sampling frequency for these signals.

Line 25: Since the program writes output signals, it must invoke `dbquit` to close the files properly.

Example 7: A General-Purpose FIR Filter

This program illustrates a useful technique for obtaining something close to 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.

```
#include <stdio.h>
#include <ecg/db.h>
#define BUFLN 512
int sample_ok = 1;

DB_Sample sample(s, t)
DB_Signal s;
DB_Time t;
{
    static DB_Sample sbuf[BUFLN][DB_MAXSIG];
    static DB_Time tt = -1L;

    if (t <= tt - BUFLN)
        fprintf(stderr, "sample: buffer too short\n");
    while (t > tt)
        if (getvec(sbuf[(++tt)&(BUFLN-1)]) < 0) sample_ok = 0;
    return (sbuf[t&(BUFLN-1)][s]);
}

main(argc, argv)
int argc;
char *argv[];
{
    double *c, one = 1.0, vv, atof();
    int i, j, nc = argc - 4, nsig, v[DB_MAXSIG];
    long nsamp, t;
    static DB_Siginfo s[DB_MAXSIG];

    if (argc < 4) {
        fprintf(stderr,
```

```
31             "usage: %s record start duration [ coefficients ... ]\n",
32                     argv[0]);
33         exit(1);
34     }
35     if (nc < 1) {
36         nc = 1; c = &one;
37     }
38     else if (nc >= BUFLN ||
39              (c = (double *)calloc(nc, sizeof(double))) == NULL) {
40         fprintf(stderr, "%s: too many coefficients\n", argv[0]);
41         exit(2);
42     }
43     for (i = 0; i < nc; i++)
44         c[i] = atof(argv[i+4]);
45     if ((nsig = isigopen(argv[1], s, DB_MAXSIG)) < 1)
46         exit(3);
47     if (isigsettime(strtim(argv[2])) < 0)
48         exit(4);
49     if ((nsamp = strtim(argv[3])) < 1) {
50         fprintf(stderr, "%s: inappropriate value for duration\n",
51                 argv[0]);
52         exit(5);
53     }
54     if (osigopen("16l", s, nsig) != nsig)
55         exit(6);
56
57     for (t = 0; t < nsamp && sample_ok; t++) {
58         for (j = 0; j < nsig; j++) {
59             for (i = 0, vv = 0.; i < nc; i++)
60                 if (c[i] != 0.) vv += c[i]*sample(j, t+i);
61             v[j] = (int)vv;
62         }
63         if (putvec(v) < 0) break;
64     }
65
66     (void)newheader("out");
67     dbquit();
68     exit(0);
69 }
```

Notes:

Line 4: `BUFLN` must be a power of 2 (why? see lines 16 and 17), and it should be larger than the length of the filter (i.e., the caller should not look back further than `BUFLN-1` samples into the past, relative to the most recent sample that has been read).

Lines 6–18:
This function supplies input samples to the `main` routine as needed, and frees the `main` routine of the need to read them in strict time order. The `sample` function returns the sample from signal `s` with adjusted sample number `t` (i.e., relative to the beginning of the segment to be processed), either by retrieving it from a circular buffer of samples recently read, or by reading it using `getvec`. The ugly-looking `getvec` argument is simply the next slot in the circular buffer; note that `tt` is an internal "clock" for `sample`, which records the adjusted sample number of the most recently read sample. In this program, the test in line 13 is redundant (why?) and might be removed for efficiency's sake.

Lines 35–37:
If no coefficients are provided on the command line, the program will simply copy the selected segment of the input signals.

Lines 38–44:
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 43 and 44, the ASCII strings that represent the coefficients are converted to `double` format and stored in the coefficient vector.

Lines 45–48:
The record name is `argv[1]`, and the start time is `argv[2]`; if the record can't be opened, or the start time is inappropriate, the program exits.

Lines 49–53:
The *duration* argument should be a time interval in *HH:MM:SS* format; `strtim` converts it to the appropriate number of samples.

Lines 54–55:
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 77). If we can't write as many output signals as there are input signals, the program exits.

Lines 57–64:
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 60, we retrieve an input sample, multiply it by a filter coefficient, and add it to a running sum. The sum is initialized to zero in line 59 before we begin, and is converted to an `int` in line 61 when we are finished. Once an entire sample vector has been filtered, it is written out in line 63. The entire process is repeated up to `nsamp` times, or until we run out of input samples.

Line 66: 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 '`header`' file for the new record. Details of data acquisition are hardware-dependent and are not shown here.

```
#include <stdio.h>
#include <ecg/db.h>

main()
{
    char answer[32], record[8], directory[32];
    int i, nsig = 0, v[DB_MAXSIG];
    long nsamp, t;
    double freq = 0.;
    static char filename[DB_MAXSIG][32],
        description[DB_MAXSIG][32], units[DB_MAXISIG][22];
    static DB_Siginfo s[DB_MAXSIG];

    do {
        printf("Choose a record name [up to 6 characters]: ");
        fgets(record, 8, stdin); record[strlen(record)-1] = '\0';
    } while (newheader(record) < 0);
    do {
        printf("Number of signals to be recorded [1-%d]: ",
               DB_MAXSIG);
        fgets(answer, 32, stdin); sscanf(answer, "%d", &nsig);
    } while (nsig < 1 || nsig > DB_MAXSIG);
    do {
        printf("Sampling frequency [Hz per signal, > 0]: ");
        fgets(answer, 32, stdin); sscanf(answer, "%lf", &freq);
    } while (setsampfreq(freq) < 0);
    do {
        printf("Length of record (H:M:S): ");
        fgets(answer, 32, stdin);
    } while ((nsamp = strtim(answer)) < 1L);
    printf("Directory for signal files [up to 30 characters]: ");
    fgets(directory, 32, stdin);
    directory[strlen(directory)-1] = '\0';
    printf("Save signals in difference format? [y/n]: ");
    fgets(answer, 32, stdin);
    s[0].fmt = (answer[0] == 'y') ? 8 : 16;
    printf("Save all signals in one file? [y/n]: ");
    fgets(answer, 32, stdin);
    if (answer[0] == 'y') {
        sprintf(filename[0], "%s/d.%s", directory, record);
        for (i = 0; i < nsig; i++) {
```

```
            s[i].fname = filename[0];
            s[i].group = 0;
        }
    }
    else {
        for (i = 0; i < nsig; i++) {
            sprintf(filename[i], "%s/d%d.%s", directory,i,record);
            s[i].fname = filename[i];
            s[i].group = i;
        }
    }
    for (i = 0; i < nsig; i++) {
        s[i].fmt = s[0].fmt; s[i].bsize = 0;
        printf("Signal %d description [up to 30 characters]: ", i);
        fgets(description[i], 32, stdin);
        description[i][strlen(description[i])-1] = '\0';
        s[i].desc = description[i];
        printf("Signal %d units [up to 20 characters]: ", i);
        fgets(units[i], 22, stdin);
        units[i][strlen(units[i])-1] = '\0';
        s[i].units = (*units[i]) ? units[i] : "mV";
        do {
            printf(" Signal %d gain [adu/%s]: ", i, s[i].units);
            fgets(answer, 32, stdin);
            sscanf(answer, "%lf", &s[i].gain);
        } while (s[i].gain < 0.);
        do {
            printf(" Signal %d ADC resolution in bits [8-16]: ", i);
            fgets(answer, 32, stdin);
            sscanf(answer, "%d", &s[i].adcres);
        } while (s[i].adcres < 8 || s[i].adcres > 16);
        printf(" Signal %d ADC zero level [adu]: ", i);
        fgets(answer, 32, stdin);
        sscanf(answer, "%d", &s[i].adczero);
    }
    if (osigfopen(s, nsig) < nsig) exit(1);
    printf("To begin sampling, press RETURN;  to specify a\n");
    printf(" start time other than the current time, enter\n");
    printf(" it in H:M:S format before pressing RETURN: ");
    fgets(answer, 32, stdin); answer[strlen(answer)-1] = '\0';
    setbasetime(answer);

    initialize ADC here

    for (t = 0; t < nsamp; t++) {
        for (i = 0; i < nsig; i++)
            v[i] = sample from ADC channel i
        if (putvec(v) < 0) break;
```

```
90         }
91
92         stop ADC here
93
94         (void)newheader(record);
95         dbquit();
96         exit(0);
97     }
```

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 94.

Lines 39–45:
This code generates a file name and initializes the `fname` and `group` fields of the array of `DB_Siginfo` objects so that all signals will be saved in one file.

Lines 46–52:
This code generates unique file names and groups for each signal.

Lines 53–76:
Here, information specific to individual signals is gathered.

Line 77: 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 78–82:
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 84: 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 86–90:
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 92: 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 94: All of the information needed to generate the '`header`' file has been stored in DB library internal data structures by `osigfopen` and `putvec`; we call `newheader` here (before `dbquit`) to create the new '`header`' file.

Line 95: It is still necessary to use `dbquit` 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 `dbquit`.)

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).

```
#include <stdio.h>
#include <ecg/db.h>
#include <ecg/ecgmap.h>

main(argc, argv)
int argc;
char *argv[];
{
    int btype, i, j, nbeats = 0, nsig, hwindow, window;
    DB_Sample v[DB_MAXSIG], vb[DB_MAXSIG];
    long stoptime = 0L, *sum[DB_MAXSIG];
    DB_Anninfo a;
    DB_Annotation annot;
    static DB_Siginfo s[DB_MAXSIG];
    void *calloc();

    if (argc < 3) {
        fprintf(stderr,
                "usage: %s annotator record [beat-type from to]\n",
                argv[0]);
        exit(1);
    }
    a.name = argv[1]; a.stat = READ;
    if ((nsig = dbinit(argv[2], &a, 1, s, DB_MAXSIG)) < 1) exit(2);
```

```
25     hwindow = strtim(".05"); window = 2*hwindow + 1;
26     btype = (argc > 3) ? strann(argv[3]) : NORMAL;
27     if (argc > 4) iannsettime(strtim(argv[4]));
28     if (argc > 5) {
29         if ((stoptime = strtim(argv[5])) < 0L)
30             stoptime = -stoptime;
31         if (s[0].nsamp > 0L && stoptime > s[0].nsamp)
32             stoptime = s[0].nsamp;
33     }
34     else stoptime = s[0].nsamp;
35     if (stoptime > 0L) stoptime -= hwindow;
36     for (i = 0; i < nsig; i++)
37         if ((sum[i]=(long *)calloc(window,sizeof(long))) == NULL) {
38             fprintf(stderr, "%s: insufficient memory\n", argv[0]);
39             exit(3);
40         }
41     while (getann(0, &annot) == 0 && annot.time < hwindow)
42         ;
43     do {
44         if (annot.anntyp != btype) continue;
45         isigsettime(annot.time - hwindow - 1);
46         (void)getvec(vb);
47         for (j = 0; j < window && getvec(v) > 0; j++)
48             for (i = 0; i < nsig; i++)
49                 sum[i][j] += v[i] - vb[i];
50         nbeats++;
51     } while (getann(0, &annot) == 0 &&
52               (stoptime == 0L || annot.time < stoptime));
53     if (nbeats < 1) {
54         fprintf(stderr, "%s: no ‘%s’ beats found\n",
55                 argv[0], annstr(btype));
56         exit(4);
57     }
58     printf("Average of %d ‘%s’ beats:\n", nbeats, annstr(btype));
59     for (j = 0; j < window; j++)
60         for (i = 0; i < nsig; i++)
61             printf("%g%c", (double)sum[i][j]/nbeats,
62                    (i == nsig-1) ? ’\n’ : ’\t’);
63     exit(0);
64 }
```

Notes:

Line 25: 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).

Line 26: 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 27: 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 28–35:
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 '`header`' file for the record, `stoptime` will have a positive value in line 35, 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 36–40:
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`.

Lines 41-42:
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 43–52:
Here we read annotations (the first is already in `annot` when we enter the loop, and subsequent annotations are read in line 51); select the desired ones (line 44); skip to the correct spot in the signals (line 45; the sample selected there is the one just before the beginning of the window); read a sample from each signal (line 46) into the `vb` vector, which will be used as a crude baseline estimate; read `window` samples from each signal (line 47), subtracting the baseline from each and adding the result into the running totals; update a beat counter (line 50); and check for loop termination conditions (line 52).

Lines 53–62:
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 61 (necessary to preserve precision), and the trick used in line 62 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).

```
#include <stdio.h>
#include <ecg/db.h>
#include <ecg/ecgcodes.h>

#define abs(A) ((A) >= 0 ? (A) : -(A))

main(argc, argv)
int argc;
char *argv[];
{
    int filter, time=0, slopecrit, sign, maxslope=0, nslope=0,
        qtime, maxtime, t0, t1, t2, t3, t4, t5, t6, t7, t8, t9,
        ms160, ms200, s2, scmax, scmin = 0, v[DB_MAXSIG];
    DB_Anninfo a;
    DB_Annotation annot;
    static DB_Siginfo s[DB_MAXSIG];

    if (argc < 2) {
        fprintf(stderr, "usage: %s record [threshold]\n", argv[0]);
        exit(1);
    }
    a.name = argv[0]; a.stat = WRITE;
    if (dbinit(argv[1], &a, 1, s, DB_MAXSIG) < 1) exit(2);
    if (sampfreq(NULL) != 250.)
        fprintf(stderr, "warning: %s is designed for 250 Hz input\n",
                argv[0]);
    if (argc > 2) scmin = muvadu(0, atoi(argv[2]));
    if (scmin < 1) scmin = muvadu(0, 1000);
    slopecrit = scmax = 10 * scmin;
    ms160 = strtim("0.16"); ms200 = strtim("0.2"); s2 = strtim("2");
    annot.subtyp = annot.chan = annot.num = 0; annot.aux = NULL;
    (void)getvec(v);
    t9 = t8 = t7 = t6 = t5 = t4 = t3 = t2 = t1 = v[0];

    do {
        filter = (t0 = v[0]) + 4*t1 + 6*t2 + 4*t3 + t4
                - t5         - 4*t6 - 6*t7 - 4*t8 - t9;
        if (time % s2 == 0) {
            if (nslope == 0) {
                slopecrit -= slopecrit >> 4;
```

```
41                      if (slopecrit < scmin) slopecrit = scmin;
42                  }
43                  else if (nslope >= 5) {
44                      slopecrit += slopecrit >> 4;
45                      if (slopecrit > scmax) slopecrit = scmax;
46                  }
47              }
48              if (nslope == 0 && abs(filter) > slopecrit) {
49                  nslope = 1; maxtime = ms160;
50                  sign = (filter > 0) ? 1 : -1;
51                  qtime = time;
52              }
53              if (nslope != 0) {
54                  if (filter * sign < -slopecrit) {
55                      sign = -sign;
56                      maxtime = (++nslope > 4) ? ms200 : ms160;
57                  }
58                  else if (filter * sign > slopecrit &&
59                           abs(filter) > maxslope)
60                      maxslope = abs(filter);
61                  if (maxtime-- < 0) {
62                      if (2 <= nslope && nslope <= 4) {
63                          slopecrit += ((maxslope>>2) - slopecrit) >> 3;
64                          if (slopecrit < scmin) slopecrit = scmin;
65                          else if (slopecrit > scmax) slopecrit = scmax;
66                          annot.time = strtim("i") - (time - qtime) - 4;
67                          annot.anntyp = NORMAL; (void)putann(0, &annot);
68                          time = 0;
69                      }
70                      else if (nslope >= 5) {
71                          annot.time = strtim("i") - (time - qtime) - 4;
72                          annot.anntyp = ARFCT; (void)putann(0, &annot);
73                      }
74                      nslope = 0;
75                  }
76              }
77              t9 = t8; t8 = t7; t7 = t6; t6 = t5; t5 = t4;
78              t4 = t3; t3 = t2; t2 = t1; t1 = t0; time++;
79          } while (getvec(v) > 0);
80
81          dbquit();
82          exit(0);
83      }
```

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 24–26:
Most of this program is independent of sampling frequency, but the filter (lines 36–37) 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). The program will work for MIT DB records sampled at 360 Hz, but better results can be obtained if the filter is optimized for the correct sampling frequency.

Lines 27–29:
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 32–33:
Here we read the first sample and copy it into the variables that will be used to store the ten most recent samples.

Lines 36–37:
This FIR filter differentiates and low-pass filters the input signal.

Lines 38–47:
Here we adjust the threshold if more than two seconds have elapsed since a QRS was detected. In line 40, `slopecrit` is set to 15/16 of its previous value if no slopes have been found; in line 45, it is set to 17/16 of its previous value if 5 or more slopes were found (suggesting the presence of noise).

Lines 48–52:
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 53–76:
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 54–57), which must occur within a specified time. We record the maximum absolute value of the filter in `maxslope` (lines 58–60) for eventual use in updating the threshold (lines 63–65). Once a sufficient interval has elapsed following the last threshold crossing (line 61), if there were between 2 and 4 slopes, we have (apparently) found a QRS complex, and the program records a `NORMAL` annotation (lines 66–67). If there were 5 or more slopes, the program records an artifact annotation (lines 71–72). If only 1 slope was found, it is assumed to represent a baseline shift and no output is produced.

Lines 77–79:

At the end of the loop, the samples are shifted through the `t`*n* 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. Remember to set and export the environment variable `DB` (see Section 1.3 [DB path], page 13). It is a good idea to include this step in your '`.profile`', '`.cshrc`', or '`autoexec.bat`'. As input, try record '`100s`', input annotator '`atruth`', and output annotator '`normal`'. The program should finish in five seconds or less. The annotations will have been written into a file called '`normal.100s`' (under MS-DOS, '`100s.nor`') in the current directory. Now type "`rdann atruth 100s`" and observe the output for a few seconds, then try "`rdann normal 100s`" 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. Wesley Q. Phortran, IV, wrote this program to print beat times (in minutes and seconds) and R-R intervals for the reference annotation file of record '`100`'. Why doesn't it work?

```
#include <ecg/db.h>

main()
{
    DB_Annotation *annot;
    DB_Anninfo ai;
    int t;

    ai.name = "atruth";
    ai.stat = READ;
    if (annopen(100, ai, 1)) {
        while (getann(1, annot)) {
            printf("%s\t(%d)\t%s\n", timstr(annot.time),
                   annot.time, mstimstr(annot.time - t));
            t = annot.time;
        }
    }
}
```

    Extra credit: Without actually trying it out, what *does* it produce on the standard output?

12. 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?

13. This exercise assumes that you have access to the MIT-BIH Arrhythmia Database. Since the 360 Hz sampling frequency used in that database is an integer multiple of 60 Hz, it is quite easy to design a 60 Hz notch filter that can be applied to the database. Write a program that filters two input signals and writes out the filtered data using `putvec` (see [Example 7], page 90, for a model program). Try it out on MIT-BIH record '`122`'. Caution: signal files are quite large. In order to avoid using too much disk space for this exercise, limit the program to writing 10 seconds' worth of samples. Use your programs from the previous exercises to display your output.
14. If you used Example 7 as a model in the previous exercise, you may have noticed that it is quite slow. Make it faster by arranging for `sample` to return a pointer to a vector of samples from all signals (thereby reducing the number of function calls). Speed it up further by defining a macro that calls the function only if the proper sample vector is not already in the circular buffer; otherwise the macro should evaluate to a pointer to the correct sample vector.
15. Prof. Nottin Ventedhier says, "Real programmers don't use inefficient library I/O routines — they write their own, in assembly language." Implement a version of the QRS detector in Example 10 *without* using the DB library. (To keep it simple, assume that only one input format — of your choice — needs to be supported.) How much faster than the original is your version?
16. (Non-trivial) Write a QRS detector that is independent of sampling frequency. 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. If you have access to the MIT-BIH Arrhythmia Database, run your detector on record '`200`'. 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 '`atruth`'. How does your detector compare to Example 10?
17. If the previous exercise was too easy, modify your detector so that the annotations it generates match those in the '`atruth`' file. Copying the '`atruth`' file is not permitted. You may find this rather difficult. Good luck!

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 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 MIT format (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` (a constant defined in '`<ecg/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. On writable UNIX and Macintosh file systems, the annotation file name is constructed from the annotator name by appending a '`.`' and the record name. On CD-ROMs and MS-DOS file systems, the annotator name is restricted to three characters, and the annotation file name is constructed from the record name by appending a '`.`' and the annotator name. UNIX and Macintosh

versions of the DB library can locate and read annotation files named using either convention.

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. Since at most `DB_MAXANN` (defined in '`<ecg/db.h>`') input or output annotation files may be open at once, the largest valid annotator number is `DB_MAXANN - 1`.

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

'`atruth`' The annotator name for the reference annotation files. On CD-ROMs and MS-DOS file systems, '`atruth`' is truncated within the names of the reference annotation files to '`atr`'; `annopen` accepts either '`atruth`' or '`atr`' as the annotator name for these files.

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 `DB_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 `DB_Calinfo` objects (see Section 3.2 [Calibration Information Structures], page 62). Each such structure specifies the size and type of the calibration pulse, and the customary plotting scale, for a particular type of signal.

CD-ROM A read-only medium used for distribution of the MIT-BIH and ESC databases, among others. CD-ROMs are physically identical in appearance to audio compact disks. CD-ROM readers are available for most microcomputers and for many larger systems.

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 DB 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 DB library.

Database path
The names of the directories in which '`header`', annotation, and calibration files are kept. The environment variable `DB` must be set by the user and exported accordingly.

DB library
A set of functions (subroutines), able to read and write database files, callable by C and C++ programs, and described in this guide.

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.

'`header`' *file*
: A file accessible via the DB library that describes the signal files associated with a given database record. On writable UNIX and Macintosh file systems, '`header`' files have names of the form '`header.`*record*', where *record* is the record name (q.v.). On CD-ROMs and MS-DOS file systems, '`header`' files have names of the form '*record*`.hea`'. UNIX and Macintosh versions of the DB library can locate and read '`header`' files named using either convention.

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.

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
: The standard format for storage of DB records on CD-ROMs, used on the MIT, ESC, and MGH DB CD-ROMs, among others.

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 `dbinit`) 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 `dbinit`) 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 `dbinit`. (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 DB 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 DB 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 DB Software Package includes `dbcollate` (under MS-DOS, `dbcoll8`), an application that can create multi-segment records from sets of single-segment records. Version 9.1 and later versions of the DB 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.

9-track tape

A medium used for archival storage of DB 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.

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.

Prolog
: Extraneous bytes at the beginning of a signal file that are not to be read as samples. Signal files created using the DB library do not contain prologs, but signal files created using other means may contain prologs. To read such a signal file using the DB library, provided that the sample data is 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 *ECG Database Applications Guide*, or see [dbsetstart], page 57.

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 optical 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 `DB_MAXRNL` (defined in '`<ecg/db.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 '`atruth`' 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 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 31), 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. Since at most `DB_MAXSIG` (defined in '`<ecg/db.h>`') input or output signals may be open at once, the largest valid signal number is `DB_MAXSIG - 1`.

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(1)* (in the *ECG Database Applications Guide*), or see [dbsetskew], page 57. 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 35).

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".

Appendix B Installing the DB Software Package

How to obtain the DB Software Package

This appendix briefly describes how to install the DB Software Package on a new system. The package includes C-language sources for the DB library and for a variety of applications (see Appendix C [DB Applications], page 119), TeX source for this manual, `troff` source for UNIX `man` pages for the package, and a one-minute sample record (‘`100s`’). Except for UNIX-specific and MS-DOS-specific visualization programs and an MS-DOS-specific digitization/replay program, all of the software is portable between UNIX and MS-DOS, and is usable under VMS and on the Macintosh with only minor modifications.

The package is included on the second edition of the MIT-BIH Arrhythmia Database CD-ROM (which also includes a version of *WAVE* for Sun SPARCstations), and is also available separately on 1.44 Mb 3.5" floppy diskettes. For information on obtaining the latest version of the DB Software Package, see Appendix E [Sources], page 129.

The DB library is provided in precompiled form for MS-DOS users, together with the ‘include’ (‘`.h`’) files needed to use it, on several other CD-ROMs that contain databases of ECGs and other signals, including the first edition of the MIT-BIH Arrhythmia Database CD-ROM, both editions of the European ST-T Database CD-ROM and the MIT-BIH Polysomnographic Database CD-ROM, and on the MGH/Marquette Foundation Waveform Database CD-ROMs (see Appendix E [Sources], page 129). These CD-ROMs do *not* include the DB applications included in the DB Software Package. To use the precompiled DB library, you must compile and link your own programs with it using the Microsoft C compiler (or another compiler/linker that is compatible with Microsoft ‘`.lib`’ libraries). If you have only a CD-ROM with MS-DOS binaries, and have not obtained the DB Software Package, see the files ‘`readme.doc`’ (in the root directory of the CD-ROM) and ‘`lib.doc`’ (in the ‘`lib`’ directory) for information about using the precompiled DB library, and check ‘`lib\ecg\db.h`’ to see which DB library version is on your CD-ROM.

On more recent (1995 and later) CD-ROMs, the DB Software Package is provided in source form within ‘`software/db`’, in precompiled binary form for MS-DOS within ‘`software/msdos`’, and as `tar` archives (named `software/`*xxx*`.tar`) containing precompiled binaries for several versions of UNIX. The DB Browser for MS Windows (‘`wview`’) is provided on these CD-ROMs in binary form only, within `software/msdos`. *WAVE* is also provided on these CD-ROMs in binary form only, within the `tar` archives for those versions of UNIX under which *WAVE* runs (as of mid-1995, these included SunOS 4.x, SPARC Solaris 2.x, and Linux). If you are installing the DB Software Package from one of these CD-ROMs, first review the general notes in 'software/README.TXT', then follow the specific instructions for your operating system found in:

- 'software/MAC.TXT', for the Macintosh
- 'software/MSDOS.TXT', for MS-DOS and MS Windows
- 'software/UNIX.TXT', for all versions of UNIX (Linux users, also see 'software/LINUX.TXT')

The remainder of this appendix refers only to the diskette and the August 1992 CD-ROM distributions.

UNIX

Select an existing directory in a writable file system for the installation; '/usr/local/src' is a good choice in most cases. Make sure that at least 6 megabytes are available (most of this space can be recovered after the installation is complete).

If you are installing the software from a CD-ROM, copy the contents of the 'src/db' directory to your writable file system. One way to do this is to change to the 'src' directory on the CD-ROM and then to type:

```
tar cfv - db | ( cd /usr/local/src; tar xfv - )
```

To save space, you may prefer to make symbolic links (if your system supports them) from the CD-ROM files to '/usr/local/src'.

If you have obtained the DB Software Package on a diskette, copy the file 'db-tar.z' from the diskette to your writable file system. (Note that 'db-tar.z' is a binary file; do not remove carriage return characters from it as you would normally do if copying an MS-DOS text file.) Now type:

```
uncompress <db-tar.z | tar xfv -
```

This command unpacks the archive into a directory called 'db'. You may remove 'db-tar.z' from your hard disk now if you wish. (If you don't have 'uncompress' or 'tar', unpack the archive on a PC using the MS-DOS installation procedure below; you will be given an opportunity to stop once the archive has been unpacked. Copy the entire contents of the 'db' directory to your UNIX file system, treating all files as binary files.)

Change to the 'db' directory on your hard disk:

```
cd db
```

Type the following command to rename some of the files for UNIX:

```
sh setunix
```

Follow the instructions in file 'UNIX.TXT', within the 'db' directory, to complete the installation. (In most cases, this requires only reading 'Makefile', changing the site-dependent variables defined there as appropriate for your system, then typing 'make'. Depending on the speed of your system and of your C compiler, 'make' will generally require between 5 and 15 minutes.)

MS-DOS

Install your C compiler if you have not already done so, and make sure that your hard disk has at least 6 megabytes of free space remaining. (Most of this space can be reclaimed after the installation is complete.)

If you have Microsoft or Turbo C or C++, and a Microstar Laboratories DAP 1200- or 2400-series analog interface board, you can instruct the 'install' procedure to compile 'sample' (a program for creating database records from analog signals, and for replaying them in analog form). To do so successfully, you must first have installed the Microstar '#include' files and DAP interface library on your system. Specifically, files 'c_lib.c', 'clock.h', and 'ioutil.h' must be installed in your 'include' directory, and file 'cdapl.lib' must be installed in a directory in which libraries are found by your linker. (If you are using Microsoft C, copy 'cdapl5.lib' from the Microstar distribution diskettes into your library directory, and rename it 'cdapl.lib'.)

If you are installing the software from a CD-ROM, load the CD-ROM and type:

d:\src\install

(replacing *d* by the drive letter of your CD-ROM drive).

If you are installing the software from a diskette, load the diskette into the a drive and type:

```
a:install
```

(or `b:install` if you must use the `b` drive).

Follow the instructions that are presented on-screen to complete the installation. Depending on the speed of your system and of your C compiler, the automatic portion of the installation will generally require from 5 minutes to an hour once you have selected the configuration.

Other systems

If you are installing the software from a CD-ROM, copy the contents of the '`src/db`' directory to your hard disk. If you are installing the software from a diskette, and your system does not have UNIX-compatible '`tar`' and '`compress`' utilities, unpack the archive on a PC and copy the entire contents of the '`db`' directory to your system. Follow the MS-DOS installation procedure as above; you will be given an opportunity to stop once the archive has been unpacked. Note that the unpacked text files are in UNIX format (i.e., lines are terminated by ASCII line-feed characters only). If your system expects text files in MS-DOS format (with both a carriage return and a line-feed at the end of each line; VMS is one such system), let the MS-DOS installation procedure continue one step further, until it announces that it has reformatted the text files for MS-DOS. If your system is a Macintosh (which expects that lines are terminated by carriage returns only), you will have to reformat the text files yourself, which may be done on the PC if you have third-party software to do so, or on the Macintosh otherwise. Additional notes for Macintosh users may be found in file '`MAC`' (within the '`db`' directory).

The DB Software Package is written in highly portable C, and (with the exception of a few MS-DOS or UNIX-specific display or data-acquisition programs) should be easy to compile with any K&R or ANSI C compiler. The UNIX and MS-DOS '`make`' description files ('`makefile.unx`' and '`makefile.dos`' in '`db`' and in each of its subdirectories) should get you started.

Appendix C DB Application Programs

This appendix briefly describes the application programs that are included with the DB Software Package. Except where noted otherwise, these applications are usable on all systems for which the DB library is available. For details on using these programs, refer to the *ECG Database 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 DB 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 1.3 [DB path], page 13). 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 35). 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 DB 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

```
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 ...]*

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 is 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, a standard (designated EC38) for ambulatory electrocardiographs. EC38 specifies 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 specifies 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 included on the MIT-BIH Arrhythmia Database CD-ROM), 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 Standard for Ambulatory Electrocardiographs* (ANSI/AAMI EC38–1994) and *Testing and Reporting Performance Results of Ventricular Arrhythmia Detection Algorithms* (AAMI ECAR–1987; see Appendix E [Sources], page 129).

To evaluate an arrhythmia detector using this software, obtain for each DB 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 ECAR). 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 *ECG Database 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 ECAR (section 4.3) and in EC38 (section 4.2.14.2.2). 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 ECAR (section 5.3) and in EC38 (section 4.2.14.2.4). '`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 4.2.14.2.5 and 4.2.14.2.6). 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 ECAR (sections 4.6.1 and 5.6) and EC38 (section 3.2.14.3) 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 EC38 (section A.3.2.14.2.6.1).

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 *ECG Database 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 DB format. '`nst`' was used to generate the graded series of noisy ECG records in the '`nstdb`' directory of the MIT-BIH Arrhythmia Database CD-ROM. These records are among those specified as standard test material by EC38 (section 3.2.14.2).

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 ...]*
`calibrate -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 DB 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 '`header`' 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 DB records, similar to the one discussed in chapter 6 (see [Example 7], page 90).

'`sigamp`' measures signal amplitudes (either baseline-corrected RMS amplitudes or peak-to-peak amplitudes); it may be useful for calibrating signals (together with '`calibrate`') 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 99). 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 129) 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 93) and to the source for '`sample`' as models.

If you create your own database records using '`sample`' or other means, program '`calibrate`' may be useful for determining signal gains and offsets if your signals include standard calibration pulses or identifiable signal levels. '`calibrate`' incorporates two independent algorithms for measuring calibration pulses; it rewrites '`header`' files based on its measurements.

Graphical Applications

`wave -r` *record [* `-a` *annotator]*
`dbtool -r` *record [* `-a` *annotator]*

`dbplot -t`*record* [`-a`*annotator options* ...]
`view` *record annotator*
`wview` *record annotator*
`pschart` [[*options* ...] *script* ...]
`psfd` [[*options* ...] *script* ...]

'`wave`' is an X Window System client application for viewing and editing DB records. ('`wave`' is not included in the DB software package, but is available separately; see Appendix E [Sources], page 129.) '`wave`' can be run on UNIX systems, and can be accessed remotely using networked PCs or other systems for which X11 servers are available. Run '`wave`' without any arguments to obtain instructions for printing its on-line manual.

'`dbtool`' is a SunView application for viewing the specified record. '`dbtool`' can be run on Sun workstations only. It is very similar to '`wave`', but '`dbtool`' lacks annotation editing capabilities.

If your system supports the UNIX '`plot`' utility, you can pipe the output of '`dbplot`' into '`plot`' in order to view DB records with annotations interactively on a graphics terminal. '`dbplot`' has many options, which can be given as arguments or interactively.

'`view`', included on the MIT-BIH Arrhythmia Database and European ST-T Database CD-ROMs, among others, is an MS-DOS application for viewing DB records on CGA, EGA, VGA, SVGA, XGA, or Hercules graphics-capable PCs. See '`bin.doc`' in the '`bin`' directory of the CD-ROM for more information.

'`wview`', included on recent (1995 and later) CD-ROMs and also available separately (see Appendix E [Sources], page 129; note that '`wview`' is not part of the DB Software Package), is an MS-Windows application for viewing DB records. It has most of the display capabilities of '`wave`', but lacks support for annotation editing.

'`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*.

Appendix D Extensions

This section may be helpful if you wish to extend the capabilities of the DB 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 DB library sources (see Appendix E [Sources], page 129). The sources are distributed among four 'include' ('`.h`') files and five '`.c`' files:

`db.h`	Constant and structure definitions, and function prototypes
`ecgcodes.h`	Annotation codes
`ecgmap.h`	Annotation code mapping macros
`dblib.h`	External definitions for private DB library functions
`dbinit.c`	Functions `dbinit`, `dbquit`, and `dbflush`
`signal.c`	Functions for signals
`calib.c`	Functions for signal calibration
`annot.c`	Functions for annotations
`dbio.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 <ecg/`*file*`.h>`' statements. When modifying the DB 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 DB library.

If you wish to increase the number of simultaneously accessible signals or annotators, simply change `DB_MAXSIG` or `DB_MAXANN` in '`db.h`', and recompile the DB library and your applications. Your operating system's limit on the number of simultaneously open files may require you to use multiplexed signal files if you increase `DB_MAXSIG` substantially.

The cleanest mechanism for adding additional fields to '`header`' files is to include them in 'info' strings (see [`getinfo`], page 54), rather than by modifying the code that reads and writes '`header`' 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 DB 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 '`db.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 '`Sample`' in '`<ecg/db.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 DB 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. 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 DB 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 '`db.h`'. In '`annot.c`', add additional `case` statements and code to `annopen`, `getann`, `putann`, and `db_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 MIT 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 `dbinit`). 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 DB applications. Another solution is to modify the DB 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 DB 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 DB 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 DB library modules. For signal processing applications that do not involve ECGs, the entire '`annot.c`' package may be removed (with trivial modifications to the functions in '`dbinit.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 '`dbio.c`' for reading and writing them in a machine-independent format.

Porting the DB 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 `db_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, `getdb` can be modified to read the DB path from a file. You may wish to modify the private function `db_error` (which is responsible for all error reporting from DB library functions) if the 'standard error output' is unavailable or inadequate for use in the target environment. All of these functions are contained within '`dbio.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 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.

Appendix E Sources

This section is a compendium of sources for databases and related materials that may be useful to readers of this guide. If you have installed the DB Software Package, you may find an updated version of this list in '`SOURCES.TXT`', in the '`db`' directory that was created during the installation. Please send any corrections to the author (first address below).

ECG Database Programmer's Guide (this guide)
ECG Database Applications Guide
WAVE User's Guide
MIT-BIH Arrhythmia Database CD-ROM
MIT-BIH Arrhythmia Database Directory
MIT-BIH Polysomnographic Database CD-ROM
MIT-BIH Polysomnographic Database Directory
Samples of Physiologic Databases CD-ROM
Software for Physiologic Databases with Samples CD-ROM
MIMIC Database CD-ROMs

MIT-BIH Database Distribution
MIT Room 20A-113
77 Massachusetts Ave.
Cambridge, MA 02139 USA

e-mail: `dbdist@hstbme.mit.edu`, `george@hstbme.mit.edu`
WWW: http://ecg.mit.edu
telephone: +1 617 253 7424
telefax: +1 617 253 2514

European ST-T Database CD-ROM
European ST-T Database Directory
VALE Database Directory

National Research Council (CNR) Institute of Clinical Physiology
Computer Laboratory
via Trieste, 41
56100 PISA, Italy

e-mail: `taddei@anemone.ifc.pi.cnr.it`
telephone: +39 50 502771
telefax: +39 50 589038

AHA Database for Evaluation of Ventricular Arrhythmia Detectors

ECRI
5200 Butler Pike
Plymouth Meeting, PA 19462 USA

e-mail: `ecri@shrsys.hslc.org`
telephone: +1 215 825 6000

MGH/Marquette Foundation Waveform Database CD-ROMs
Anaesthesia/Bioengineering Unit
Massachusetts General Hospital
Fruit St.
Boston, MA 02114 USA

e-mail: `cooper@etherdome.mgh.harvard.edu`
telephone: +1 617 726 8824

This is a large database of multi-channel recordings (3 ECG leads, radial arterial, pulmonary arterial, and central venous pressure, respiration, and CO2), which has been issued on 10 CD-ROMs.

American National Standard for Ambulatory Electrocardiographs (ANSI/AAMI EC38–1994)
AAMI Recommended Practice for Testing and Reporting Performance Results of Ventricular Arrhythmia Detection Algorithms (AAMI ECAR–1987)
Association for the Advancement of Medical Instrumentation
3330 Washington Boulevard, Suite 400
Arlington, VA 22201 USA

telephone: +1 703 525 4890
telefax: +1 703 276 0793

Computers in Cardiology
WWW: `http://www.cinc.org/`

CIC 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. CIC conferences have taken place annually since 1974, usually in September; in even-numbered years, they are convened in North America, and in Europe in odd-numbered years. The deadline for submission of abstracts is 1 May each year. Proceedings of the conferences are published by the IEEE Computer Society Press, and usually appear about 3 months after the date of the conference. CIC will be in Lund (Sweden) in 1997.

Proceedings of Computers in Cardiology (ISSN 0276-6574)
IEEE Customer Service
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331 USA

e-mail: `customer.service@ieee.org`
WWW: `http://www.ieee.org/bkstr.html`
telephone: 1 800 678 IEEE (USA and Canada) or +1 908 981 0060
telefax: +1 908 981 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
675 Massachusetts Ave.
Cambridge, MA 02139 USA

e-mail: `gnu@prep.ai.mit.edu`
WWW: `http://www.gnu.ai.mit.edu`
telephone: +1 617 876 3296

GNU software is available on CD-ROM or tape from the address above, and is also freely available by anonymous FTP from:

prep.ai.mit.edu
wuarchive.wustl.edu
ftp.cs.columbia.edu
gatekeeper.dec.com
labrea.stanford.edu
ftp.uu.net
ftp.cs.ubc.ca
ftp.unicamp.br
archie.au
ftp.technion.ac.il
ftp.sun.ac.za
ftp.informatik.tu-muenchen.de
ftp.sunet.se
hp4nl.nluug.nl
ftp.funet.fi
ftp.denet.dk
ugle.unit.no
ftp.eunet.ch
irisa.irisa.fr
ftp.ieunet.ie
archive.eu.net
cair.kaist.ac.kr
ftp.nectec.or.th
utsun.s.u-tokyo.ac.jp
ftp.cs.titech.ac.jp

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.shsu.edu, ftp.tex.ac.uk, and ftp.uni-stuttgart.de. 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://jasper.ora.com/ctan.html`').

The UNIX TEX distribution is also distributed on CD-ROM and in other tape formats by the Free Software Foundation (address above) and others. It is also included with most Linux distributions (see below).

Several commercial implementations of TEX for MS-DOS are widely available.

General information on TEX

TeX Users Group
1850 Union St., #1637
San Francisco, CA 94123 USA

WWW: `http://www.tug.org`
e-mail: `tug@tug.org`
telephone: +1 415 982 8449
telefax: +1 415 982 8559

X11R6 (the X Window System, Version 11, Release 6)
XView

The Open Group
11 Cambridge Center
Cambridge, MA 02142 USA

WWW: `http://www.x.org/`

These packages are freely available by anonymous FTP from ftp.x.org and other archive sites. X11R6 sources are also available on tape from the Free Software Foundation (address above).

Linux

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 Linux, visit the web site of the Linux Documentation Project:

`WWW: http://sunsite.unc.edu/linux/`

Linux is freely available by anonymous FTP in source and binary form from many sites, including:

tsx-11.mit.edu
sunsite.unc.edu
ftp.funet.fi
net.tamu.edu
ftp.mcc.ac.uk

src.doc.ic.ac.uk
fgb1.fgb.mw.tu-muenchen.de
ftp.informatik.tu-muenchen.de
ftp.dfv.rwth-aachen.de
ftp.informatik.rwth-aachen.de
ftp.Germany.EU.net
ftp.ibp.fr
kirk.bond.edu.au
ftp.uu.net
wuarchive.wustl.edu
ftp.win.tue.nl
ftp.stack.urc.tue.nl
srawgw.sra.co.jp
cair.kaist.ac.kr
ftp.denet.dk
NCTUCCCA.edu.tw
nic.switch.ch
monu1.monash.edu.au
cnuce_arch.cnr.it

There are also many low-cost (typically US$10 to US$30) distributions of Linux on CD-ROMs widely available. Among the more popular are:

Red Hat Linux
3203 Yorktown Ave., Suite 123
Durham, NC 27713 USA

e-mail: `sales@redhat.com`
WWW: `http://www.redhat.com/`
telephone: 1 800 546 7274 (USA and Canada) or +1 919 572 6500
telefax: +1 919 572 6726

InfoMagic
11950 N. Hwy 89
Flagstaff, AZ 86004 USA

e-mail: `questions@infomagic.com`
WWW: `http://www.infomagic.com/`
telephone: 1 800 800 6613 (USA and Canada) or +1 520 526 9565
telefax: +1 520 526 9573

Walnut Creek CD-ROM
4041 Pike Lane, Suite E
Concord, CA 94520 USA

e-mail: `info@cdrom.com`
WWW: `http://www.cdrom.com/`
telephone: 1 800 786 9907 (USA and Canada) or +1 510 674 0783

telefax: +1 510 674 0821

Caldera, Inc.
633 South 550 East
Provo, UT 84606 USA

e-mail: `info@caldera.com`
WWW: `http://www.caldera.com/`
telephone: +1 801 377 7687
telefax: +1 801 377 8752

S.u.S.E. GmbH
e-mail: `kfr@suse.de`
WWW: `http://www.suse.de/`

See ads in Byte, Dr. Dobb's Journal, and similar magazines for other sources.

Borland C++
Turbo C/C++

Borland International, Inc.
1800 Green Hills Road
P.O. Box 660001
Scotts Valley, CA 95067 USA

WWW: `http://www.borland.com/`
telephone: +1 408 438 5300

Microsoft C/C++
MS-DOS with Microsoft CD-ROM extensions

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052 USA

WWW: `http://www.microsoft.com/`
telephone: +1 206 936 8661

Versions of MS-DOS earlier than 6.0 did not include 'Microsoft CD-ROM extensions' ('`mscdex.exe`', an MS-DOS TSR that intercepts read requests directed to a CD-ROM drive, thus making an ISO 9660 or High Sierra file system appear to be a read-only MS-DOS file system). If you have an earlier version of MS-DOS, either upgrade to MS-DOS 6.x or later, or obtain '`mscdex.exe`' from your CD-ROM drive vendor. Be sure to specify which version of MS-DOS you are using, since newer versions of MS-DOS are not compatible with older versions of '`mscdex.exe`' (MS-DOS 3.x requires '`mscdex`' 2.00 or later, MS-DOS 4.x requires '`mscdex`' 2.10 or later, and MS-DOS 5.0 requires '`mscdex`' 2.20 or later). Microsoft Windows 95 (Chicago) does not require '`mscdex`', since it incorporates the functions of '`mscdex`' into the operating system.

CD-ROM drives

These are now available from most PC and UNIX workstation vendors. For the CD-ROMs listed above, the only requirement is that the drive and the software supplied with it must be compatible with your operating system and with ISO 9660 format CD-ROMs. All currently available drives for PCs and UNIX workstations meet this requirement. High-speed drives (4x, 6x, and faster) are now cheaper than single-speed drives were three years ago; for use with signal databases, these high-speed drives are well worth the small incremental cost.

For users of UNIX versions that date from 1993 and earlier, it may be worthwhile to upgrade to newer software that supports Rock Ridge extensions to ISO 9660 format. Our recent (1995 and later) CD-ROMs are written in this format, which appears identical to ISO 9660 format when read on a PC or an older UNIX system, but supports full UNIX file naming semantics on Rock Ridge aware systems.

Microstar DAP 1200 and 2400 analog interface boards for PCs

Microstar Laboratories
2265 116th Avenue N.E.
Bellevue, WA 98004 USA

telephone: +1 206 453 2345
telefax: +1 206 453 3199
WWW: http://www.mstarlabs.com/mstarlabs/

Web browsers

The most popular Web browsers may be downloaded by anonymous FTP.

Netscape
FTP: `ftp.netscape.com`
WWW: `http://www.netscape.com`

Mosaic
FTP: `ftp.ncsa.uiuc.edu`
WWW: `http://www.ncsa.uiuc.edu/SDG/Software`

NCSA Mosaic is available in both source and precompiled form, and is freely usable by anyone. Netscape is available in precompiled form only, and is free for academic use or for limited-period evaluations; information about commercial and other non-academic licensing is available on Netscape's Web site.

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 −2048 to +2047 adu, or −174.8 to +234.7 mmHg.

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. On a UNIX system that supports the '`graph`' and '`plot`' commands, 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 `| graph | plot`

11. Line 5 allocates storage for a pointer; in line 12, `getann` needs storage for the annotation structure. '`t`' is declared as an `int` in line 7, which works on a 32-bit machine, but probably not on a 16-bit machine. Line 11 contains three errors (did you find all of them?): the first argument to `annopen` should be a character string, not an integer; the second argument should be a *pointer* to a `DB_Anninfo` object, not the object itself; and (as written) the '`if`' condition is satisfied only if `annopen` fails (it returns zero if successful). Line 12 also contains three errors: the first argument to `getann` is the annotator number, but the first (in this case, the only) input annotator is `0`, not `1`; the second argument to `getann` should be, but is not, a pointer to an allocated `DB_Annotation` structure; and the '`while`' loop terminates only if `getann` *succeeds*. There are two errors in lines 13 and 14: '`annot.time`' is a long integer (unless `long` is equivalent to `int` on your machine, '`%d`' is an incorrect specification for printing it); and the functions `timstr` and `mstimstr` return pointers to static storage that is overwritten by each call. If the other errors are fixed, the `printf` statement will print the same string twice (which one depends on the order of evaluation of function arguments, which may vary between compilers). Having fixed all of these errors, the output is still incorrect, since `getann` returns rhythm and signal quality annotations as well as beat labels (only the latter should be used for calculating R-R intervals), and '`t`' is not initialized, which makes the first interval wrong in any case. As for the

extra credit question, the program probably produces nothing at all on its standard output! If, by some miracle, `annopen` succeeds, it returns zero, and the body of the '`if`' is never executed. If `annopen` simply fails, perhaps because the input annotation file can't be opened, `getann` also fails, and the program probably dumps core with an illegal memory reference in the `printf` statement, since `annot` hasn't been initialized. More likely, the program will dump core in `annopen`, attempting to reference memory location 100.

Concept Index

C

D

E

F

G

H

I

J

L

M

Function and Macro Index

For a number of entries below, the function name is followed by the version number of the DB library in which the function first appeared. Functions for which no such number appears have been present in all numbered versions of the DB library.

P

S

T

U

Table of Contents