Database Independent Abstraction Layer for C

libdbi Programmer's Guide

David A. Parker
Neon Goat Productions

david@neongoat.com

Markus Hoenicka

markus@mhoenicka.de

Database Independent Abstraction Layer for C: libdbi Programmer's Guide

by David A. Parker

by Markus Hoenicka

Document revision: $Id: programmers-guide.sgml,v 1.18 2013/02/03 23:04:32 mhoenicka Exp $ Edition
Published $Date: 2013/02/03 23:04:32 $

libdbi implements a database-independent abstraction layer in C, similar to the DBI/DBD layer in Perl. Writing one generic set of code, programmers can leverage the power of multiple databases and multiple simultaneous database connections by using this framework.

Table of Contents

List of Tables

Chapter 1. Introduction

1.1. Description

libdbi provides application developers with a database independent abstraction layer for C. It handles the database-specific implementations for each type of database, so that you can use the same exact code with any type of database server that libdbi supports. You can initiate and use multiple database connections simultaneously, regardless of the types of database servers you are connecting to. The driver architecture allows for new database drivers to be easily added dynamically.

1.2. libdbi Concepts and Terminology

In this guide, the terms “user” and “programmer” are used interchangably, since the target audience is the software developer using libdbi in his program. A star character (*) represents a wildcard matching any letters. For example, “dbi_conn_*” would represent all functions beginning with “dbi_conn_”.

Before doing anything useful, your program must initialize libdbi. This creates an “instance” of libdbi which is accessible through a handle. The libdbi architecture provides several “drivers”, one for each type of database server. All drivers are loaded into memory by each libdbi instance upon initialization and are made available to the programmer. Once a driver is *instantiated*, it represents a distinct database session and is called a “connection”.

1.3. Modifications and redistribution of libdbi

1.4. Contact Info

Please email us with any bugs, ideas, feature requests, or questions. The libdbi website has the latest version of this documentation and the libdbi software, as well as a central database of third-party drivers.

- http://libdbi.sourceforge.net
- David Parker <david@neongoat.com>
- Mark Tobenkin <mark@brentwoodradio.com>
- Markus Hoenicka <markus@mhoenicka.de>

Chapter 2. Building and Installing libdbi

libdbi uses the "autotools", i.e. automake, autoconf, and libtool to provide a simple and consistend build and install process on all supported platforms. The good news is that you usually don't have to fiddle that much with **./configure** as libdbi provides only a single option:

--disable-docs

> If you build libdbi from a tarball, the docs are already prebuilt and ready to install. In this case, you don't need this option. However, if you build from a cvs checkout, this is a simple way to switch off building the documentation. This is useful if you don't want to install the prerequisites for building the docs (openjade, the DocBook DSSSL stylesheets, TeX, and JadeTeX).

If you type **./configure --help**, you'll see a few more options. These are provided by the autotools, and you should consult their documentation to make best use of them.

Once you have figured out which options to use or not, you are ready to go ahead and run:

```
$ ./configure [your options]
$ make
$ sudo make install
```

Chapter 3. libdbi in a Nutshell (Quickstart Guide)

3.1. Quick Overview

libdbi is implemented as a library and is usually available as a shared object (aka dynamically linked library). If your program is supposed to use libdbi's capabilities, your program must be linked against libdbi which will be explained shortly. At runtime, the first step is always to create a libdbi instance by calling dbi_initialize_r. Programs that use threads or a plugin system may create more than one instance if needed and use them independently of each other. libdbi uses a plugin system that allows various databases to be supported simultaneously, and can dynamically load or unload drivers that are supplied by libdbi or a third party. Each instance loads the available database drivers independently. Programs can then use one or more of these drivers to create connections to databases, either by calling dbi_driver_open_r followed by dbi_conn_open, or in a single step by calling dbi_conn_new_r. Multiple connections may exist for a single driver, and all will function independently of each other. As not all database engine client libraries allow simultaneous access of a single connection by multiple threads, it is advisable to use separate connections per thread, even if you use a single libdbi instance for the entire process. Also, an instance may open connections using different drivers at the same time. Each libdbi instance can be independently shut down when no longer needed by calling dbi_shutdown_r which frees the memory occupied by the loaded drivers.

> **Note:** libdbi versions 0.8.x and earlier did not allow the use of separate instances per process. Instead, each process had exactly one instance which was managed by libdbi internally. This interface is still available for backwards compatibility but will eventually be dropped.

The connection's options (username, password, hostname, etc.) are set with dbi_conn_set_option and dbi_conn_set_option_numeric. Once all options are set, dbi_conn_connect will connect to the database, waiting to handle a dbi_conn_query. A query is a string containing a valid SQL statement. libdbi provides several functions to automatically quote any characters that might screw up the query string. The preferred functions are dbi_conn_quote_string and dbi_conn_quote_string_copy as they take into consideration the character encoding used by the current connection. The legacy functions dbi_driver_quote_string and dbi_driver_quote_string_copy are still supported but should be avoided in new code. After a successful query, you can retrieve rows with dbi_result_first_row, dbi_result_last_row, dbi_result_prev_row, dbi_result_next_row, and dbi_result_seek_row.

String data may be sent to and retrieved from a database using character encodings if they contain characters not covered by the ASCII character set. Most database engines support character encodings like ISO-8859-1, suitable for many European languages, or even the multibyte Unicode character sets like UTF-8. The character set used to store your data in your database is usually set by the **CREATE DATABASE** command, which you have to take care of yourself. libdbi uses the connection option "encoding" to select a particular character encoding for the current connection. If you set the value to "auto", libdbi will automatically use the database character encoding as the connection encoding. If you request a different character encoding, as defined by its IANA (http://www.iana.org) name, libdbi will convert the data on the fly.

There are two methods for fetching field data, and two ways to perform each method. You can either "pull" the data from DBI using the `dbi_result_get_*` family of functions, or have DBI automatically "push" the data into predefined variables with the `dbi_result_bind_*` family of functions. Both families of functions are as strongly typed as most SQL database engines are. That is, you must use the `dbi_result_get_*` or `dbi_result_bind_*` function that matches the type of the requested field. Table 3-1 shows an overview of these functions sorted by the field type they retrieve.

Pulling the data from the database can be done with one of the "get" functions such as dbi_result_get_long or dbi_result_get_string, which simply return the data in the field you asked for. You should run the function dbi_conn_error_flag immediately after each call to a "get" function to check for errors. You can also get more than one field at a time with dbi_result_get_fields, which uses a printf-like syntax.

If you want DBI to automatically fill your program's variables with field values whenever a new row is fetched, you can "bind" fields to your variables. Bindings are set up with dbi_result_bind_long, dbi_result_bind_string, and the rest of the bind family of functions. Like the associated "get" function, you can set up multiple bindings at once with the dbi_result_bind_fields function.

String data can be safely included into query strings by using the dbi_conn_quote_string and dbi_conn_quote_string_copy functions. Binary data can be included into query strings by using the dbi_conn_quote_binary_copy function. All of these functions return zero-terminated strings enclosed in the appropriate quoting characters. Binary strings are returned in their binary representation. That is, they may contain null bytes and other non-printable characters. It is mandatory to use the dbi_result_get_field_length or dbi_result_get_field_length_idx functions to determine the number of bytes contained in the binary string.

Caveats:

- For fields holding integers (not fractional numbers), DBI differentiates between signed and unsigned variables. By default, DBI returns signed values. If you want an unsigned value, prepend a "u" to the name of the target type. For example, dbi_result_bind_short becomes dbi_result_bind_ushort.
- You must set up any bindings *after* a successful query but *before* you fetch any rows. Even if you are using field bindings, you can still use the dbi_result_get_* functions as usual. (actually, I lied... setting up a binding should theoretically work at any time, but don't plan on this behavior in future versions)
- All string and binary data returned or bound from DBI is *read-only*. If you want your own local copy that can be modified at will, use dbi_result_get_string_copy, dbi_result_get_binary_copy, dbi_result_bind_string_copy, or dbi_result_bind_binary_copy. You will be responsible for freeing the memory allocated by these functions.

dbi_result_next_row and the other row-seeking functions will return zero when there are no more rows available. Before the program terminates, you must call dbi_result_free on every result set and the connection must be disconnected and unloaded with dbi_conn_close. Finally, libdbi must be unloaded with dbi_shutdown.

Table 3-1. get* and bind* functions sorted by field type

field type	get by name	get by field index	bind
signed char	dbi_result_get_char	dbi_result_get_char_idx	dbi_result_bind_char
unsigned char	dbi_result_get_uchar	dbi_result_get_uchar_idx	dbi_result_bind_uchar
short	dbi_result_get_short	dbi_result_get_short_idx	dbi_result_bind_short

field type	get by name	get by field index	bind
unsigned short	dbi_result_get_ushort	dbi_result_get_ushort_idx	dbi_result_bind_ushort
int	dbi_result_get_int	dbi_result_get_int_idx	dbi_result_bind_int
unsigned int	dbi_result_get_uint	dbi_result_get_uint_idx	dbi_result_bind_uint
long long	dbi_result_get_longlong	dbi_result_get_longlong_idx	dbi_result_bind_longlong
unsigned long long	dbi_result_get_ulonglong	dbi_result_get_ulonglong_idx	dbi_result_bind_ulonglong
float	dbi_result_get_float	dbi_result_get_float_idx	dbi_result_bind_float
double	dbi_result_get_double	dbi_result_get_double_idx	dbi_result_bind_double
character string	dbi_result_get_string, dbi_result_get_string_copy	dbi_result_get_string_idx, dbi_result_get_string_copy_idx	dbi_result_bind_string
binary string	dbi_result_get_binary, dbi_result_get_binary_copy	dbi_result_get_binary_idx, dbi_result_get_binary_copy_idx	dbi_result_bind_binary
date/time	dbi_result_get_datetime	dbi_result_get_datetime_idx	dbi_result_bind_datetime

3.2. Generic Example Program

The following listing shows how to establish a connection to a MySQL database server and retrieve the results of a SQL query. Only a small number of functions offered by libdbi are shown here. For a more extensive example check out the test program `tests/test_dbi.c` in the libdbi-drivers (http://libdbi-drivers.sourceforge.net) source tarball.

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

int main() {
    dbi_conn conn;
    dbi_result result;
    dbi_inst instance;

    double threshold = 4.333333;
    unsigned int idnumber;
    const char *fullname;

    dbi_initialize_r(NULL, &instance);
    conn = dbi_conn_new_r("mysql", instance);

    dbi_conn_set_option(conn, "host", "localhost");
    dbi_conn_set_option(conn, "username", "your_name");
    dbi_conn_set_option(conn, "password", "your_password");
```

```
    dbi_conn_set_option(conn, "dbname", "your_dbname");
    dbi_conn_set_option(conn, "encoding", "UTF-8");

    if (dbi_conn_connect(conn) < 0) {
      printf("Could not connect. Please check the option settings\n");
    }
    else {
      result = dbi_conn_queryf(conn, "SELECT id, name FROM coders "
                                     "WHERE hours_of_sleep > %0.2f", threshold);

      if (result) {
 while (dbi_result_next_row(result)) {
   idnumber = dbi_result_get_uint(result, "id");
   fullname = dbi_result_get_string(result, "name");
   printf("%i. %s\n", idnumber, fullname);
 }
 dbi_result_free(result);
      }
      dbi_conn_close(conn);
    }

    dbi_shutdown_r(instance);

    return 0;
}
```

Compile with: gcc -lm -ldl -ldbi -o foo foo.c

> **Note:** The `-ldl` option is not required on systems that implement the dynamic linking in their libc (like FreeBSD). You may also have to throw in something like `-I/usr/local/include` and `-L/usr/local/lib` to help gcc and ld find the libdbi headers and libraries.

Of course, a complete program should check for errors more thoroughly. This example keeps error-checking at a minimum for the sake of clarity. There are also other ways to retrieve data after a successful query. Keep reading on to see the rest.

3.3. Loading libdbi at runtime

The generic example shown in the previous section assumed that the program is linked against libdbi. This is in fact the recommended way to add libdbi functionality to your programs. However, there are situations where this approach will not work. Some programs are designed to load modules at runtime to extend their capabilities. A well-known example is the web server Apache, which uses loadable modules to custom-tailor its capabilities. If such a module were to use libdbi, we'd look at the following pattern:

Parent => dl_open(module) => dl_open(driver)

The dynamically loaded module is linked against libdbi whereas the parent application is not. For this pattern to work, the drivers are linked against libdbi by default. This avoids "undefined symbol" errors at runtime, but may cause problems under arcane conditions or on equally arcane operating systems. If you should ever encounter such problems, you can switch off linking the drivers against libdbi like this:

```
~/libdbi-drivers #./configure --disable-libdbi --with-mysql
```

3.4. Using libdbi in multithreaded applications

With some precautions, libdbi can be used in multithreaded applications. One way to do this is to initialize separate libdbi instances in each thread. If this is not an option and several threads have to share a single instance, you currently need to protect certain libdbi function calls with mutexes. These functions are dbi_conn_query, dbi_conn_ping, and dbi_conn_error.

3.5. Adding libdbi to your project

If your project uses autoconf to manage the build process on the target machine, you should add some tests to your `./configure` script to check for the presence and usability of libdbi. The following example shows how this can be done:

```
dnl check for dynamic linking functions
AC_CHECK_LIB(dl,dlopen)

dnl check for the libdbi library
AC_CHECK_LIB(dbi,dbi_initialize)

dnl to check for the availability and function of a particular
dnl driver we need a runtime check (since the driver is loaded
dnl dynamically). This example checks for the mysql driver
AC_MSG_CHECKING("for libdbi mysql driver (dynamic load)")
AC_RUN_IFELSE(
  [AC_LANG_PROGRAM(,
    [[dbi_initialize(0); return(dbi_conn_new("mysql") ? 0 : 1);]])],
  [AC_MSG_RESULT("yes")],
  [AC_MSG_FAILURE("mysql driver not installed?")])
```

The first two tests add the appropriate flags to the `LIBS` variable to link against the required libraries.

In addition, you have to make sure that both the directory which contains the libdbi header file directory (usually `/usr/include` or `/usr/local/include`) as well as the directory which contains the libdbi library (usually `/usr/lib` or `/usr/local/lib`) are accessible to the compiler and to the linker by using the `-I` and `-L` compiler flags, respectively.

Chapter 4. Error Handling

Applications should check all libdbi function calls for errors and respond appropriately to avoid entering an undefined status. libdbi uses two mechanisms to indicate errors:

Function return values

Essentially all libdbi functions return a value, a concept not unfamiliar to the seasoned C programmer. For example, the dbi_initialize function returns the number of loaded drivers, or -1 if an error occurred. In this case checking the return value is sufficient to detect an error condition. However, other functions like the family of "getters" cannot indicate error conditions with a return value. Consider e.g. the dbi_result_get_string function which is used to retrieve strings from a database. If there was an error in accessing the value, the function will return the string "ERROR". However, this string is a legal value of such a field (the problem is the same for any other conceivable return value, including the empty string and the NULL pointer). Therefore we need an additional mechanism to report errors.

Error numbers and error messages

Connections store the status of the most recent operation which can be queried by the accessor function `dbi_conn_error`. This is equivalent to the `errno` variable of the standard C library which is used by most system calls and can be printed in human-readable form by the `perror` system call. This mechanism implies that your program queries the status right after each operation, as the values will be overwritten by subsequent operations.

The return values of all libdbi functions are listed in the reference chapter below. The error numbers will be briefly discussed in the following sections.

4.1. Connection error numbers and messages

If your application has successfully opened a connection to a database, all operations on this connection may cause errors. There are two classes of errors:

Client library errors

libdbi retrieves and stores the error codes of the database engine client library if it provides error codes in a suitable format. If the client library does not support error numbers as positive integers, the value DBI_ERROR_CLIENT indicates an error status instead. In addition a string may be provided which describes or further elaborates the error status. The possible values of error number and error message depend on the database engine used for the connection.

libdbi errors

A number of errors may occur within the libdbi framework or within a database driver, e.g. if your program queries nonexistent columns or attempts to read past the last row of a result set. The possible error numbers and error messages are predefined by libdbi.

Use the function dbi_conn_error to access the error number and error message of the most recent connection operation.

4.2. Error numbers

dbi_conn_error returns DBI_ERROR_NONE (internally 0) if the last operation was successful, and a nonzero value if not. Client library errors use positive error numbers, whereas libdbi errors use negative error numbers as listed in the following table:

Table 4-1. libdbi error numbers

value	description
DBI_ERROR_USER	This indicates an error status set by the application itself, see dbi_conn_set_error
DBI_ERROR_BADOBJECT	invalid connection or result structure
DBI_ERROR_BADTYPE	the accessor function does not match the actual column type
DBI_ERROR_BADIDX	out-of-range index
DBI_ERROR_BADNAME	incorrect column or option name
DBI_ERROR_UNSUPPORTED	feature not supported by driver
DBI_ERROR_NOCONN	no valid connection
DBI_ERROR_NOMEM	out of memory
DBI_ERROR_BADPTR	invalid pointer
DBI_ERROR_NONE	no error occurred
DBI_ERROR_CLIENT	client library error

Chapter 5. Transactions and Savepoints

A variety of database engines support the concepts of transactions and savepoints. The common idea of both is to group database commands in a way such that the changes caused by the commands can be either written to the database or be rolled back as if the commands hadn't been issued. libdbi provides a consistent interface for both transactions and savepoints.

5.1. Transactions

To find out at runtime whether or not the driver associated with the current connection supports transactions, use the function dbi_conn_cap_get to query if the "transaction_support" capability is nonzero. Drivers should throw an error if transaction-related functions are called although the database engine does not support them. If a database engine supports both transaction-safe and non-transaction safe table types, the behaviour of the driver depends on its implementation.

Transactions are usually handled by a standard sequence of actions. First, the transaction is set up. Then one or more commands are executed. Based on the result of these commands, the transaction can either be aborted (rolled back) or be written to the database (committed). libdbi provides the following functions to handle these tasks:

Start a transaction

Some database engines start a transaction after a connection is established, and after each transaction is finished. These database engines do not strictly need a command to start a transaction. Others work in autocommit mode, i.e. each command is automatically committed. Some of the latter can be coerced to use transactions, while others simply do not provide explicit transaction management. As a database abstraction layer, libdbi has to cover all cases by providing an explicit command to start transactions, see dbi_conn_transaction_begin.

Commit a transaction

If no errors occurred during a transaction, the changes can actually be written to the database by calling dbi_conn_transaction_commit.

Rollback a transaction

If an error occurred during a transaction, the database can be reverted to the state when the transaction started by calling dbi_conn_transaction_rollback.

5.2. Savepoints

Most database engines which support transactions also support the concept of savepoints. To find out at runtime whether or not the driver associated with the current connection supports savepoints, use the function dbi_conn_cap_get to query if the "savepoint_support" capability is nonzero. Savepoints are essentially named markers within a transaction where you can jump back in case of an error. In this case, all commands that were issued after the savepoint are dismissed. The changes caused by the commands after a savepoint are committed as

soon as the entire transaction is committed. Usually several markers may be used at a time within the same transaction. libdbi provides the following commands to manage savepoints:

Set a savepoint

To set a named savepoint within a transaction, use the function dbi_conn_savepoint. You need to keep track of the name of savepoints in order to be able to revert the changes.

Rollback to a savepoint

To dismiss any changes during an open transaction which occurred after a particular savepoint, use the function dbi_conn_rollback_to_savepoint.

Release a savepoint

Some database engines allow to clear savepoints if they are no longer needed. This is important only within lengthy transactions as savepoints are automatically cleared when a transaction is committed or rolled back. Releasing a savepoint does not affect the commands that occurred after the savepoint was set. It merely makes it impossible to jump back to that savepoint and releases all system resources which were needed to maintain that savepoint. To release a savepoint, use the command dbi_conn_release_savepoint.

Chapter 6. Library and Interface Versions

Libraries are no static entities, just as any other piece of software. Some parts evolve, some optimizations are made, new features are requested, and old features may be too cumbersome to support if no one uses them anyway. Any of these developments may result in a new release. This chapter briefly discusses the issues related to library and driver versions.

The version issues with libdbi are a little more convoluted than with a run-of-the-mill library because we have to consider two application interfaces (APIs):

DBI API

This is, as with any other library, the data structures and the functions that you, as a programmer of a software linked against libdbi, need to know about.

DBD API

This is the interface that matters to the authors of database engine drivers.

Both interfaces may change independently, and both may affect the version number of the package as will be discussed shortly.

6.1. Package and library versions

libdbi utilizes libtool (http://www.gnu.org/software/libtool/) to manage the library in a platform-independent fashion. libtool uses an abstract library versioning scheme which consists of three numbers:

current

This is the number of the current interface. The first "incarnation" is usually 0 (zero), the next version that adds or subtracts something from this interface would be 1 (think of new API functions or functions with altered parameter lists).

revision

This number counts the released changes of the interface which do not alter the interface (think of internal optimizations, bugfixes and so on)

age

This is the number of previous interfaces that the current version is backwards-compatible with. That is, if `current` is "n", programs linked against "n", "n-1", and "n-2" will run with the current version.

The package version number (major.minor.patch) is computed from the above numbers using the formulas:

- major = current - age
- minor = age + 8

- patch = revision

The correction applied to `minor` is currently required to bring the package version in line with earlier releases which did not use proper libtool interface versioning. It will be dropped if `major` rises above zero.

6.2. libdbi and libdbi-drivers versions

The libdbi-drivers package is essentially developed independently of libdbi as most new versions are mandated by improvements of existing drivers and by the implementation of new drivers. However, both changes in the DBD API and in the DBI API (which is also accessible to the drivers) may also require a new libdbi-drivers version. In order to keep the confusion at a minimum, libdbi-drivers major and minor version numbers are chosen to match the libdbi versions they are compatible with. That is, any libdbi-drivers version 0.9.* is supposed to work with any libdbi version 0.9.*. Further compatibility information, e.g. whether a putative libdbi version 2.* is able to load libdbi-drivers 1.* are found in the release notes.

6.3. Determining the library version at runtime

The functions dbi_version and dbi_version_numeric provide access to the package version. The former returns the name and the version as a string (mainly for display purposes), whereas the latter returns an integer value for easy comparisons, defined as ((major*10000) + (minor * 100) + patch).

Chapter 7. libdbi API Reference

7.1. Instance Infrastructure

7.1.1. dbi_initialize_r

```
int dbi_initialize_r(const char *driverdir, dbi_inst *pInst);
```

Creates an instance of libdbi, locates all available database drivers and loads them into memory.

Arguments

`driverdir`: The directory to search for drivers. If NULL, DBI_DRIVER_DIR (defined at compile time) will be used instead.

`pInst`: A pointer to an instance handle. The function will fill in the new instance handle if successful, or set it to NULL if an error occurred.

Returns

The number of drivers successfully loaded, or -1 if there was an error. The latter may be due to an incorrect driver directory or to a lack of permissions.

Availability

0.9.0

7.1.2. dbi_initialize

```
int dbi_initialize(const char *driverdir);
```

Locates all available database drivers and loads them into memory.

Note: This function is deprecated. Use dbi_initialize_r instead. In contrast to that function, dbi_initialize allows only one (internally managed) libdbi instance per process.

Arguments

`driverdir`: The directory to search for drivers. If NULL, DBI_DRIVER_DIR (defined at compile time) will be used instead.

Returns

The number of drivers successfully loaded, or -1 if there was an error.

7.1.3. dbi_shutdown_r

```
void dbi_shutdown_r(dbi_inst Inst);
```

Frees all loaded drivers and terminates the libdbi instance. You should close each connection you opened before shutting down, but libdbi will clean up after you if you don't.

Arguments

`Inst`: The instance handle.

Availability

0.9.0

7.1.4. dbi_shutdown

```
void dbi_shutdown(void);
```

Frees all loaded drivers and terminates the DBI system. You should close each connection you opened before shutting down, but libdbi will clean up after you if you don't.

Note: This function is deprecated. Use dbi_shutdown_r instead.

7.1.5. dbi_set_verbosity_r

```
int dbi_set_verbosity_r(int verbosity, dbi_inst Inst);
```

Toggles internal error messages on or off in the given libdbi instance. This affects only those error messages which are directly sent to stderr, not those handled by the connection error API.

Arguments

`verbosity`: A nonzero value causes error messages to be printed on stderr. 0 (zero) suppresses error messages.

`Inst`: The instance handle.

Returns

The previous setting of *`verbosity`*.

Availability

0.9.0

7.1.6. dbi_set_verbosity

```
int dbi_set_verbosity(int verbosity);
```

Toggles internal error messages on or off. This affects only those error messages which are directly sent to stderr, not those handled by the connection error API.

Note: This function is deprecated. Use dbi_set_verbosity_r instead.

Arguments

`verbosity`: A nonzero value causes error messages to be printed on stderr. 0 (zero) suppresses error messages.

Returns

The previous setting of *`verbosity`*.

7.1.7. dbi_version

```
const char *dbi_version(void);
```

Requests the version of libdbi as a read-only string. The calling program must not attempt to free the returned string.

Returns

A string containing the library's name and version.

7.1.8. dbi_version_numeric

```
unsigned int dbi_version_numeric(void);
```

Requests the version of libdbi as an integer.

Returns

The version, computed as ((major * 10000) + (minor * 100) + patch).

Availability

0.9.0

7.2. Driver Infrastructure

7.2.1. dbi_driver_list_r

```
dbi_driver dbi_driver_list_r(dbi_driver Current, dbi_inst Inst);
```

Enumerates all loaded drivers of the given instance. If Current is NULL, the first available driver will be returned. If Current is a valid driver, the next available driver will be returned.

Arguments

`Current`: The current driver in the list of drivers, or NULL to retrieve the first one.

`Inst`: The instance handle.

Returns

The next available driver, or NULL if there is an error or no more are available.

Availability

0.9.0

7.2.2. dbi_driver_list

```
dbi_driver dbi_driver_list(dbi_driver Current);
```

Enumerates all loaded drivers. If Current is NULL, the first available driver will be returned. If Current is a valid driver, the next available driver will be returned.

Note: This function is deprecated. Use dbi_driver_list_r instead.

Arguments

`Current`: The current driver in the list of drivers, or NULL to retrieve the first one.

Returns

The next available driver, or NULL if there is an error or no more are available.

7.2.3. dbi_driver_open_r

```
dbi_driver dbi_driver_open_r(const char *name, dbi_inst Inst);
```

Locate the driver with the specified name.

Arguments

`name`: The name of the driver to open.

`Inst`: The instance handle.

Returns

The requested driver, or NULL if there is no such driver.

Availability

0.9.0

7.2.4. dbi_driver_open

```
dbi_driver dbi_driver_open(const char *name);
```

Locate the driver with the specified name.

Note: This function is deprecated. Use dbi_driver_open_r instead.

Arguments

`name`: The name of the driver to open.

Returns

The requested driver, or NULL if there is no such driver.

7.2.5. dbi_driver_get_instance

```
int dbi_driver_get_instance(dbi_driver Driver);
```

Retrieves the handle of the instance that loaded the driver.

Arguments

`Driver`: The target driver.

Returns

The instance handle, or NULL in case of an error.

Availability

0.9.0

7.2.6. dbi_driver_is_reserved_word

```
int dbi_driver_is_reserved_word(dbi_driver Driver, const char *word);
```

Looks for the specified word in the list of reserved words. The result of this function may vary between databases. Case does not matter.

Arguments

`Driver`: The target driver.

`word`: The word to check against the reserved word list.

Returns

-1 if an error occurs, 0 if the word is not reserved, 1 otherwise.

7.2.7. dbi_driver_specific_function

```
void *dbi_driver_specific_function(dbi_driver Driver, const char *name);
```

Returns a function pointer to the specifed custom function. This can be used to access database-specific functionality, but it will restrict your code to one particular database, lessening the benefits of using libdbi.

Arguments

`Driver`: The target driver.

`name`: The name of the custom function.

Returns

If the custom function is found, a pointer to that function. If not, returns NULL.

Availability

0.9.0

7.2.8. dbi_driver_quote_string

```
int dbi_driver_quote_string(dbi_driver Driver, char **orig);
```

Encloses the target string in the types of quotes that the database expects, and escapes any special characters. The original string will be freed and *orig* will point to a newly allocated one (which you still must free on your own). If an error occurs, the original string will not be freed.

Note: This function is deprecated. Use dbi_conn_quote_string instead.

Arguments

`Driver`: The target driver.

`orig`: A pointer to the string to quote and escape.

Returns

The new string's length in bytes, excluding the terminating zero byte, or 0 in case of an error. The length of a quoted empty string is 2 bytes.

7.2.9. dbi_driver_quote_string_copy

```
int dbi_driver_quote_string_copy(dbi_driver Driver, char **orig, char **newstr);
```

Encloses the target string in the types of quotes that the database expects, and escapes any special characters. The original string will be left alone, and *`newstr`* will point to a newly allocated string containing the quoted string (which you still must free on your own). In case of an error, *`newstr`* is an invalid pointer which you must not attempt to deallocate.

Note: This function is deprecated. Use dbi_conn_quote_string_copy instead.

Arguments

`Driver`: The target driver.

`orig`: A pointer to the string to quote and escape.

`newstr`: After the function returns, this pointer will point to the quoted and escaped string.

Returns

The quoted string's length in bytes, excluding the terminating zero byte, or 0 in case of an error. The length of a quoted empty string is 2 bytes.

7.2.10. dbi_driver_encoding_from_iana

```
const char *dbi_driver_encoding_from_iana(dbi_driver Driver, const char
*iana_encoding);
```

Requests the database engine specific name of the character encoding identified by its name as known to IANA (http://www.iana.org). Use this function to pass the database engine specific encoding name to SQL queries , e.g. as part of a **CREATE DATABASE** command.

Arguments

`Driver`: The target driver.

`iana_encoding`: The IANA name of the character encoding.

Returns

A string containing the database engine specific encoding name. If the encoding name cannot be translated, the IANA name is returned without translation.

Availability

0.8.0

7.2.11. dbi_driver_encoding_to_iana

```
const char *dbi_driver_encoding_to_iana(dbi_driver Driver, const char *db_encoding);
```

Requests the IANA (http://www.iana.org) name of the character encoding identified by its database engine specific name. Use this function to convert the database engine specific name returned by SQL queries to the corresponding common name.

Arguments

`Driver`: The target driver.

`db_encoding`: The database engine specific name of the character encoding.

Returns

A string containing the IANA encoding name. If the encoding name cannot be translated, the database engine specific name is returned without translation.

Availability

0.8.0

7.2.12. Driver Information

7.2.12.1. dbi_driver_get_name

```
const char *dbi_driver_get_name(dbi_driver Driver);
```

Requests the name of the specified driver. The calling program must not attempt to free the returned string.

Arguments

`Driver`: The target driver.

Returns

A string containing the driver's name.

7.2.12.2. dbi_driver_get_filename

```
const char *dbi_driver_get_filename(dbi_driver Driver);
```

Requests the filename of the specified driver. The calling program must not attempt to free the returned string.

Arguments

`Driver`: The target driver.

Returns

A string containing the driver's full path and file name.

7.2.12.3. dbi_driver_get_description

```
const char *dbi_driver_get_description(dbi_driver Driver);
```

Requests a description of the specified driver. The calling program must not attempt to free the returned string.

Arguments

`Driver`: The target driver.

Returns

A string containing the driver's description. It will be one or two short sentences with no newlines.

7.2.12.4. dbi_driver_get_maintainer

```
const char *dbi_driver_get_maintainer(dbi_driver Driver);
```

Requests the maintainer of the specified driver. The calling program must not attempt to free the returned string.

Arguments

`Driver`: The target driver.

Returns

A string containing the driver maintainer's full name and email address.

7.2.12.5. dbi_driver_get_url

```
const char *dbi_driver_get_url(dbi_driver Driver);
```

Requests the maintainer's URL for the specified driver. This is useful for drivers maintained by a third party. The calling program must not attempt to free the returned string.

Arguments

`Driver`: The target driver.

7.2.12.6. dbi_driver_get_version

```
const char *dbi_driver_get_version(dbi_driver Driver);
```

Requests the version of the specified driver. The calling program must not attempt to free the returned string.

Arguments

`Driver`: The target driver.

Returns

A string containing the driver's version.

7.2.12.7. dbi_driver_get_date_compiled

```
const char *dbi_driver_get_date_compiled(dbi_driver Driver);
```

Requests the compilation date of the specified driver. The calling program must not attempt to free the returned string.

Arguments

`Driver`: The target driver.

Returns

A string containing the date the driver was compiled.

7.2.12.8. dbi_driver_cap_get

```
int dbi_driver_cap_get(dbi_driver Driver, const char *capname);
```

Requests the value of the driver capability which is specified as an argument.

Arguments

`Driver`: The target driver.

`capname`: A pointer to a string containing the name of the driver capability to be queried.

Returns

The numeric value of the driver capability.

7.3. Connection Infrastructure

7.3.1. dbi_conn_new_r

```
dbi_conn dbi_conn_new_r(const char *name, dbi_inst Inst);
```

Creates a connection instance of the driver specified by "name" loaded by the given libdbi instance. This is a shortcut for calling dbi_driver_open_r and passing the result to dbi_conn_open.

Arguments

`name`: The name of the desired driver.

`Inst`: The instance handle.

Returns

A connection instance of the specified driver, or NULL if there was an error.

7.3.2. dbi_conn_new

```
dbi_conn dbi_conn_new(const char *name);
```

Creates a connection instance of the driver specified by "name". This is a shortcut for calling dbi_driver_open and passing the result to dbi_conn_open.

Note: This function is deprecated. Use dbi_conn_new_r instead.

Arguments

`name`: The name of the desired driver.

Returns

A connection instance of the specified driver, or NULL if there was an error.

7.3.3. dbi_conn_open

```
dbi_conn dbi_conn_open(dbi_driver Driver);
```

Creates a connection instance of the specified driver. This connection can be used to perform queries and set options.

Arguments

`Driver`: The target driver.

Returns

A connection instance of the specified driver, or NULL if there was an error.

7.3.4. dbi_conn_close

```
void dbi_conn_close(dbi_conn Conn);
```

Disconnects the specified connection connection from the database and cleans up the connection session.

Arguments

`Conn`: The target connection.

7.3.5. dbi_conn_get_driver

```
dbi_driver dbi_conn_get_driver(dbi_conn Conn);
```

Returns the driver type of the specified connection.

Arguments

`Conn`: The target connection.

Returns

The driver type of the target connection.

7.3.6. dbi_conn_set_option

```
int dbi_conn_set_option(dbi_conn Conn, const char *key, char *value);
```

Sets a specified connection option to a string value.

Arguments

`Conn`: The target connection.

`key`: The name of the target setting. Must only contain alphanumerics and the underscore character.

`value`: The string value of the target setting.

Returns

-1 on error, 0 on success. In case of an error, the error number is DBI_ERROR_NOMEM.

7.3.7. dbi_conn_set_option_numeric

```
int dbi_conn_set_option_numeric(dbi_conn Conn, const char *key, int value);
```

Sets a specified connection option to a numeric value.

Arguments

`Conn`: The target connection.

`key`: The name of the target setting. Must only contain alphanumerics and the underscore character.

`value`: The numeric value of the target setting.

Returns

-1 on error, 0 on success. In case of an error, the error number is DBI_ERROR_NOMEM.

7.3.8. dbi_conn_get_option

```
const char *dbi_conn_get_option(dbi_conn Conn, const char *key);
```

Retrieves the string value of the specified option set for a connection if available.

Arguments

`Conn`: The target connection.

`key`: The name of the target setting.

Returns

A read-only string with the setting, or NULL if it is not available. It is not considered an error if the setting is not available.

7.3.9. dbi_conn_require_option

```
const char *dbi_conn_require_option(dbi_conn Conn, const char *key);
```

Retrieves the string value of the specified option set for a connection and throws an error if the option was not set.

Arguments

`Conn`: The target connection.

`key`: The name of the target setting.

Returns

A read-only string with the setting, or NULL if it is not available. In the latter case, the error number is DBI_ERROR_BADNAME.

7.3.10. dbi_conn_get_option_numeric

```
int dbi_conn_get_option_numeric(dbi_conn Conn, const char *key);
```

Retrieves the integer value of the specified option set for a connection if available.

Arguments

`Conn`: The target connection.

`key`: The name of the target setting.

Returns

The value of the setting, or 0 (zero) if it is not available. It is not considered an error if the option has not been set.

7.3.11. dbi_conn_require_option_numeric

```
int dbi_conn_require_option_numeric(dbi_conn Conn, const char *key);
```

Retrieves the integer value of the specified option set for a connection and throws an error if it is not available.

Arguments

`Conn`: The target connection.

`key`: The name of the target setting.

Returns

The value of the setting, or -1 if it is not available. In the latter case, the error number is DBI_ERROR_BADNAME.

7.3.12. dbi_conn_get_option_list

```
const char *dbi_conn_get_option_list(dbi_conn Conn, const char *current);
```

Enumerates the list of available options for a connection. If current is NULL, the first available option will be returned. If current is a valid option name, the next available option will be returned.

Arguments

`Conn`: The target connection.

`current`: The key name of the target option.

Returns

The key name of the next option, or NULL if there are no more options or if there was an error. In the latter case the error number is set to DBI_ERROR_BADPTR.

7.3.13. dbi_conn_clear_option

```
void dbi_conn_clear_option(dbi_conn Conn, const char *key);
```

Removes the target option setting from a connection. It is not considered an error if the requested option was not set before.

Arguments

`Conn`: The target connection.

`key`: The name of the target setting.

7.3.14. dbi_conn_clear_options

```
void dbi_conn_clear_options(dbi_conn Conn);
```

Removes all option settings from a connection.

Arguments

`Conn`: The target connection.

7.3.15. dbi_conn_cap_get

```
int dbi_conn_cap_get(dbi_conn Conn, const char *capname);
```

Requests the value of a driver capability associated with the current connection. The name of the capability is specified as an argument.

Arguments

`Conn`: The target connection.

`capname`: A pointer to a string containing the name of the driver capability to be queried.

Returns

The numeric value of the driver capability.

7.3.16. dbi_conn_get_socket

```
int dbi_conn_get_socket(dbi_conn Conn);
```

Obtain the file descriptor number for the backend connection socket.

Arguments

`Conn`: The target connection

Returns

-1 on failure, the file descriptor number on success

7.3.17. dbi_conn_get_encoding

```
const char *dbi_conn_get_encoding(dbi_conn Conn);
```

Requests the character encoding used by the current connection. This may be different from the encoding requested when the connection was opened, most notably if the connection option was set to "auto".

Arguments

`Conn`: The current encoding.

Returns

A string containing the IANA (http://www.iana.org) name of the connection encoding. If the encoding option was set to "auto", the function returns the encoding the database was created with. In all other cases, the current connection encoding is returned, which may be different from the database encoding. Use the dbi_driver_encoding_from_iana function to translate the encoding name to that of the currently used database engine if necessary.

7.3.18. dbi_conn_get_engine_version_string

```
char *dbi_conn_get_engine_version_string(dbi_conn Conn, char *versionstring);
```

Requests the version of the database engine that serves the current connection as a string.

Arguments

`Conn`: The current connection.

`versionstring`: A string buffer that can hold at least VERSIONSTRING_LENGTH bytes.

Returns

A string representation of the version. This will be something like "4.1.10". The result is written to the buffer that *versionstring* points to. If successful, the function returns a pointer to that buffer. If the version cannot be determined, the function returns the string "0".

> **Note:** This string is useful to display the version to the user. In order to check for particular version requirements in your program, dbi_conn_get_engine_version is the better choice.

Availability

0.8.0

7.3.19. dbi_conn_get_engine_version

```
unsigned int dbi_conn_get_engine_version(dbi_conn Conn);
```

Requests the version of the database engine that serves the current connection in a numeric form.

Arguments

Conn: The current connection.

Returns

A numeric representation of the version. String representations of the version (e.g. "4.1.10") do not lend themselves to an easy comparison in order to find out whether a particular engine feature is already implemented. For example, a string comparison would claim that "4.1.9" is a later version than "4.1.10". Therefore libdbi computes a numeric representation of the version number [[[[A.]B.]C.]D.]E[.] according to the formula E + D*100 + C*10000 + B*1000000 + A*100000000. The resulting integers (40109 and 40110 in the example above) will be sorted correctly. Returns 0 if the version number cannot be retrieved.

Availability

0.8.0

7.3.20. Error Handling

7.3.20.1. dbi_conn_error

```
int dbi_conn_error(dbi_conn Conn, const char **errmsg_dest);
```

Returns a formatted message with the error number and description resulting from the previous database operation.

Arguments

`Conn`: The target connection.

`errmsg_dest`: The target string pointer, which will point to the error message. If NULL, no error message will be created, but the error number will still be returned. This string is managed by libdbi, so it must not be modified or freed. The pointer to the string is only valid until the next call to this function, so make a copy in time if you need to keep the error message.

Returns

The error number of the most recent database operation if it resulted in an error. If not, this will return DBI_ERROR_NONE. See Chapter 4 for further information.

7.3.20.2. dbi_conn_error_handler

```
void dbi_conn_error_handler(dbi_conn Conn, dbi_conn_error_handler_func function, void
*user_argument);
```

Registers an error handler callback to be triggered whenever the database encounters an error. The callback function should perform as little work as possible, since the state in which it is called can be uncertain. The actual function declaration must accept two parameters (and return nothing):

- dbi_conn Conn: the connection object that triggered the error, from which dbi_conn_error() can be called, and
- void *user_argument: a pointer to whatever data (if any) was registered along with the handler.

To remove the error handler callback, specify NULL as the function and user_argument.

Arguments

`Conn`: The target connection.

`function`: A pointer to the function to call when the error handler should be triggered.

`user_argument`: Any data to pass along to the function when it is triggered. Set to NULL if unused.

7.3.20.3. dbi_conn_error_flag

```
dbi_error_flag dbi_conn_error_flag(dbi_conn Conn);
```

The libdbi query functions set an error flag in order to distinguish e.g. the return value "0" from a "0" returned due to an error condition. Use this function after each query that may fail to read out the error status.

Arguments

`Conn`: The target connection.

Returns

0 means the previous query finished without errors. A value larger than zero means an error occurred.

Note: This function is deprecated. Use dbi_conn_error instead. Both functions return the same error code to maintain backwards compatibility.

7.3.20.4. dbi_conn_set_error

```
int dbi_conn_set_error(dbi_conn Conn, int errnum, const char *formatstr, ...);
```

Applications may set an error status and an error message which are accessible through the libdbi error API function dbi_conn_error.

Arguments

`Conn`: The target connection.

`errnum`: An application-defined error code. Applications should use only positive nonzero integers to indicate errors.

The remainder of the argument list is interpreted using a printf(3)-like syntax to define an error message.

Returns

The length of the error message in bytes, or 0 if there was an error.

7.3.21. Transactions and Savepoints

7.3.21.1. dbi_conn_transaction_begin

```
int dbi_conn_transaction_begin(dbi_conn Conn);
```

Starts a transaction.

Arguments

`Conn`: The target connection.

Returns

0 (zero) if successful, otherwise nonzero.

7.3.21.2. dbi_conn_transaction_commit

```
int dbi_conn_transaction_commit(dbi_conn Conn);
```

Commits a transaction, i.e. writes all changes since the transaction was started to the database.

Arguments

`Conn`: The target connection.

Returns

0 (zero) if successful, otherwise nonzero.

7.3.21.3. dbi_conn_transaction_rollback

```
int dbi_conn_transaction_rollback(dbi_conn Conn);
```

Rolls back a transaction, i.e. reverts all changes since the transaction started.

Arguments

`Conn`: The target connection.

Returns

0 (zero) if successful, otherwise nonzero.

7.3.21.4. dbi_conn_savepoint

```
int dbi_conn_savepoint(dbi_conn Conn, const char *savepoint);
```

Sets a savepoint named *savepoint* within the current transaction.

Arguments

`Conn`: The target connection.

`savepoint`: a pointer to a string containing the name of the savepoint.

Returns

0 (zero) if successful, otherwise nonzero.

7.3.21.5. dbi_conn_rollback_to_savepoint

```
int dbi_conn_rollback_to_savepoint(dbi_conn Conn, const char *savepoint);
```

Rolls back all changes since the savepoint named *savepoint* was established within the current transaction.

Arguments

`Conn`: The target connection.

`savepoint`: a pointer to a string containing the name of the savepoint.

Returns

0 (zero) if successful, otherwise nonzero.

7.3.21.6. dbi_conn_release_savepoint

```
int dbi_conn_release_savepoint(dbi_conn Conn, const char *savepoint);
```

Removes the savepoint named *savepoint* within the current transaction and releases all resources associated with it. Changes can no longer be rolled back to that particular savepoint. However, changes may still be rolled back to different savepoints, or to the beginning of the entire transaction.

Arguments

`Conn`: The target connection.

`savepoint`: a pointer to a string containing the name of the savepoint.

Returns

0 (zero) if successful, otherwise nonzero.

7.4. SQL and Database Infrastructure

7.4.1. dbi_conn_connect

```
int dbi_conn_connect(dbi_conn Conn);
```

Connects to the database using the options (host, username, password, port, (etc.) set with dbi_conn_set_option and related functions. See the documentation for each specific database driver for the options it recognizes and requires.

Arguments

`Conn`: The target connection.

Returns

0 (zero) on success, less than zero on failure. In the latter case, the error number may be DBI_ERROR_NOMEM, DBI_ERROR_NOCONN, or one of the driver-specific values.

7.4.2. dbi_conn_get_db_list

```
dbi_result dbi_conn_get_db_list(dbi_conn Conn, const char *pattern);
```

Queries the list of available databases on the server.

Arguments

`Conn`: The target connection.

`pattern`: A string pattern (SQL regular expression) that each name must match, or NULL to show all available databases.

Returns

A query result object, which will contain database names in the first field (for use with the by-index field functions). In case of an error, the function returns NULL and sets the error number to a database engine-specific nonzero value.

7.4.3. dbi_conn_get_table_list

```
dbi_result dbi_conn_get_table_list(dbi_conn Conn, const char *db, const char
*pattern);
```

Queries the list of available tables in a particular database.

Arguments

`Conn`: The target connection.

`db`: The target database name.

`pattern`: A string pattern (SQL regular expression) that each name must match, or NULL to show all available tables.

Returns

A query result object, which will contain table names in the first field (for use with the by-index field functions). In case of an error, the function returns NULL and sets the error number to a database engine-specific nonzero value.

7.4.4. dbi_conn_select_db

```
int dbi_conn_select_db(dbi_conn Conn, const char *db);
```

Switches to a different database on the server.

Arguments

`Conn`: The target connection.

`db`: The target database name.

Returns

-1 on failure, zero on success. In case of an error, the error number may be DBI_ERROR_UNSUPPORTED or a database engine-specific nonzero value.

7.5. Managing Queries

7.5.1. dbi_conn_query

```
dbi_result dbi_conn_query(dbi_conn Conn, const char *statement);
```

Execute the specified SQL query statement.

Arguments

`Conn`: The target connection.

`statement`: A string containing the SQL statement.

Returns

A query result object, or NULL if there was an error. In the latter case the error number is a database engine-specific nonzero value.

7.5.2. dbi_conn_queryf

```
dbi_result dbi_conn_queryf(dbi_conn Conn, const char *formatstr, ...);
```

Execute the specified SQL query statement.

Arguments

`Conn`: The target connection.

`formatstr`: The format string for the SQL statement. It uses the same format as printf().

ARG: (...) Any variables that correspond to the printf-like format string.

Returns

A query result object, or NULL if there was an error. In the latter case the error number is a database engine-specific nonzero value.

7.5.3. dbi_conn_query_null

```
dbi_result dbi_conn_query_null(dbi_conn Conn, const unsigned char *statement, unsigned
long st_length);
```

Execute the specified SQL query statement, which may contain valid NULL characters.

Note: This function is not implemented by all database drivers. For a portable way of including binary strings into SQL queries, see the function dbi_conn_quote_binary_copy.

Arguments

Conn: The target connection.

statement: The SQL statement, which may contain binary data.

st_length: The number of characters in the non-null-terminated statement string.

Returns

A query result object, or NULL if there was an error. In the latter case the error number is a database engine-specific nonzero value.

7.5.4. dbi_conn_sequence_last

```
unsigned long long dbi_conn_sequence_last(dbi_conn Conn, const char *name);
```

Requests the row ID generated by the last **INSERT** command. The row ID is most commonly generated by an auto-incrementing column in the table. Use the return value to address the dataset that was last inserted.

Arguments

`Conn`: The current database connection.

`name`: The name of the sequence, or NULL if the database engine does not use explicit sequences.

> **Note:** You may have noted that this function does not sufficiently encapsulate the peculiarities of the underlying database engines. You must keep track of sequence names yourself if your target database engine does use sequences.

Returns

An integer value corresponding to the ID that was created by the last **INSERT** command. If the database engine does not support sequences, the function returns 0 (zero) and sets the error number to DBI_ERROR_UNSUPPORTED.

7.5.5. dbi_conn_sequence_next

```
unsigned long long dbi_conn_sequence_next(dbi_conn Conn, const char *name);
```

Requests the row ID that would be generated by the next **INSERT** command. The row ID is most commonly generated by an auto-incrementing column in the table.

> **Note:** Not all database engines support this feature. Portable code should use dbi_conn_sequence_last instead.

Arguments

`Conn`: The current database connection.

`name`: The name of the sequence, or NULL if the database engine does not use explicit sequences.

> **Note:** You may have noted that this function does not sufficiently encapsulate the peculiarities of the underlying database engines. You must keep track of sequence names yourself if your target database engine does use sequences.

Returns

An integer value corresponding to the ID that was created by the last **INSERT** command, or 0 if the database engine does not support this feature. In the latter case, the error number is DBI_ERROR_UNSUPPORTED

7.5.6. dbi_conn_ping

```
int dbi_conn_ping(dbi_conn Conn);
```

Checks whether the current connection is still alive. Use this function to decide whether you must reconnect before running a query if your program is designed to keep connections open over prolonged periods of time.

Arguments

`Conn`: The current database connection.

Returns

1 if the connection is alive. Otherwise the function returns 0.

> **Note:** Database drivers may attempt to reconnect automatically if this function is called. If the reconnect is successful, this function will also return 1, as if the connection never had gone down.

7.5.7. dbi_conn_quote_string

```
size_t dbi_conn_quote_string(dbi_conn Conn, char **orig);
```

Escapes any special characters in a string and places the string itself in quotes so the string can be sent to the database engine as a query string, using either dbi_conn_query or dbi_conn_queryf. The original string will be freed and *`orig`* will point to a newly allocated one (which you still must free on your own). If an error occurs, the original string will be left alone. This function is preferred over dbi_driver_quote_string because it takes the character encoding of the current connection into account when performing the escaping.

Arguments

`Conn`: The current database connection.

`orig`: A pointer to the string to quote and escape.

Returns

The new string's length in bytes, excluding the terminating zero byte, or 0 in case of an error. The length of a quoted empty string is 2 bytes. In case of an error the error number is DBI_ERROR_BADPTR or DBI_ERROR_NOMEM.

Availability

0.8.0

7.5.8. dbi_conn_quote_string_copy

```
size_t dbi_conn_quote_string_copy(dbi_conn Conn, char *orig, char **newstr);
```

Escapes any special characters in a string and places the string itself in quotes so the string can be sent to the database engine as a query string, using either dbi_conn_query or dbi_conn_queryf. The original string will be left alone, and *newstr* will point to a newly allocated string containing the quoted string (which you still must free on your own). If the function fails, *newstr* is an invalid pointer that must not be freed. This function is preferred over dbi_driver_quote_string_copy because it takes the character encoding of the current connection into account when performing the escaping.

Arguments

`Conn`: The current database connection.

`orig`: A pointer to the string to quote and escape.

`newstr`: After the function returns, this pointer will point to the quoted and escaped string.

Returns

The new string's length in bytes, excluding the terminating zero byte, or 0 in case of an error. Possible error numbers are DBI_ERROR_BADPTR and DBI_ERROR_NOMEM.

Availability

0.8.0

7.5.9. dbi_conn_quote_binary_copy

```
size_t dbi_conn_quote_binary_copy(dbi_conn Conn, char *orig, size_t from_length, char
**newstr);
```

Escapes any special characters, including null bytes, in a binary string and places the resulting string in quotes so it can be used in an SQL query. The original string will be left alone, and *newstr* will point to a newly allocated string containing the quoted string (which you still must free on your own). If an error occurs, *newstr* is an invalid pointer which must not be freed.

Arguments

`Conn`: The current database connection.

`orig`: A pointer to the string to quote and escape.

`from_length`: The length of the binary string in bytes.

`newstr`: After the function returns, this pointer will point to the quoted and escaped string.

Returns

The new string's length in bytes, excluding the terminating zero byte, or 0 in case of an error. Possible error numbers are DBI_ERROR_BADPTR and DBI_ERROR_NOMEM.

Availability

0.8.0

7.5.10. dbi_conn_escape_string

```
size_t dbi_conn_escape_string(dbi_conn Conn, char **orig);
```

Works like dbi_conn_quote_string but does not surround the resulting string with quotes.

Arguments

`Conn`: The current database connection.

`orig`: A pointer to the string to quote and escape.

Returns

The new string's length in bytes, excluding the terminating zero byte, or 0 in case of an error. The length of a quoted empty string is 2 bytes. In case of an error the error number is DBI_ERROR_BADPTR or DBI_ERROR_NOMEM.

Availability

0.8.3

7.5.11. dbi_conn_escape_string_copy

```
size_t dbi_conn_escape_string_copy(dbi_conn Conn, char *orig, char **newstr);
```

Works like dbi_conn_quote_string_copy but does not surround the resulting string with quotes.

Arguments

`Conn`: The current database connection.

`orig`: A pointer to the string to quote and escape.

`newstr`: After the function returns, this pointer will point to the quoted and escaped string.

Returns

The new string's length in bytes, excluding the terminating zero byte, or 0 in case of an error. Possible error numbers are DBI_ERROR_BADPTR and DBI_ERROR_NOMEM.

Availability

0.8.3

7.5.12. dbi_conn_escape_binary_copy

```
size_t dbi_conn_escape_binary_copy(dbi_conn Conn, char *orig, size_t from_length, char
**newstr);
```

Works like dbi_conn_quote_binary_copy but does not surround the resulting string with quotes.

Arguments

`Conn`: The current database connection.

`orig`: A pointer to the string to quote and escape.

`from_length`: The length of the binary string in bytes.

`newstr`: After the function returns, this pointer will point to the quoted and escaped string.

Returns

The new string's length in bytes, excluding the terminating zero byte, or 0 in case of an error. Possible error numbers are DBI_ERROR_BADPTR and DBI_ERROR_NOMEM.

Availability

0.8.3

7.6. Managing Results

7.6.1. dbi_result_get_conn

```
dbi_conn dbi_result_get_conn(dbi_result Result);
```

Returns the connection belonging to the specified result object.

Arguments

`Result`: The target query result.

Returns

The connection belonging to the target query result.If an error occurs, the return value is NULL, and the error number is set to DBI_ERROR_BADPTR.

7.6.2. dbi_result_free

```
int dbi_result_free(dbi_result Result);
```

Frees the result's query, disables all stored field bindings, and releases internally stored variables.

Arguments

`Result`: The target query result.

Returns

-1 on failure, zero on success. If a failure was caused by the database client library, the error number is set to the database engine-specific nonzero error code.

7.6.3. dbi_result_seek_row

```
int dbi_result_seek_row(dbi_result Result, unsigned long long rowidx);
```

Jump to a specific row in a result set.

Arguments

`Result`: The target query result.

`rowidx`: The ordinal number of the row to seek to. The first row is at position 1, not zero.

Returns

1 if successful, or 0 if there was an error. In the latter case, the error number is one of DBI_ERROR_BADPTR, DBI_ERROR_BADIDX, or a database engine-specific nonzero value.

7.6.4. dbi_result_first_row

```
int dbi_result_first_row(dbi_result Result);
```

Jump to the first row in a result set.

Arguments

`Result`: The target query result.

Returns

1 if successful, or 0 if there was an error. In the latter case, the error number is one of DBI_ERROR_BADPTR, DBI_ERROR_BADIDX, or a database engine-specific nonzero value.

7.6.5. dbi_result_last_row

```
int dbi_result_last_row(dbi_result Result);
```

Jump to the last row in a result set.

Arguments

`Result`: The target query result.

Returns

1 if successful, or 0 if there was an error. In the latter case, the error number is one of DBI_ERROR_BADPTR, DBI_ERROR_BADIDX, or a database engine-specific nonzero value.

7.6.6. dbi_result_prev_row

```
int dbi_result_prev_row(dbi_result Result);
```

Jump to the previous row in a result set.

Arguments

`Result`: The target query result.

Returns

1 if successful, or 0 if there is an error. In the latter case, the error number is one of DBI_ERROR_BADPTR, DBI_ERROR_BADIDX, or a database engine-specific nonzero value.

7.6.7. dbi_result_next_row

```
int dbi_result_next_row(dbi_result Result);
```

Jump to the next row in a result set.

Arguments

`Result`: The target query result.

Returns

1 if successful, or 0 if there was an error. In the latter case, the error number is one of DBI_ERROR_BADPTR, DBI_ERROR_BADIDX, or a database engine-specific nonzero value.

7.6.8. dbi_result_get_currow

```
unsigned long long dbi_result_get_currow(dbi_result Result);
```

Returns the ordinal number of the current row in the specified result set.

Arguments

`Result`: The target query result.

Returns

The ordinal number of the row, or 0 if there was an error. The first row has the number 1. In case of an error, the error number is DBI_ERROR_BADPTR.

7.6.9. dbi_result_get_numrows

```
unsigned long long dbi_result_get_numrows(dbi_result Result);
```

Returns the number of rows in the specified result set.

Arguments

`Result`: The target query result.

Returns

The number of rows in the result set, which may be 0 if the query did not return any datasets, or DBI_ROW_ERROR in case of an error. In that case, the error number is DBI_ERROR_BADPTR.

7.6.10. dbi_result_get_numrows_affected

```
unsigned long long dbi_result_get_numrows_affected(dbi_result Result);
```

Returns the number of rows in the specified result set that were actually modified. Note that not all database servers support this, in which case it will always be zero. See the documentation for each specific driver for details.

Arguments

`Result`: The target query result.

Returns

The number of modified rows in the result set which may be 0 if no row was affected by the previous query. Also returns 0 if the database engine does not support this feature. The return value will be DBI_ROW_ERROR in case of an error. In case of an error, the error number is DBI_ERROR_BADPTR.

7.7. Retrieving Field Meta-data

7.7.1. dbi_result_get_field_length

```
size_t dbi_result_get_field_length(dbi_result Result, const char *const char
*fieldname);
```

Returns the length of the value stored in the specified field which contains a string or a binary string.

Arguments

`Result`: The target query result.

`fieldname`: The name of the target field.

Returns

The length in bytes of the target field data, excluding the terminating zero byte, or DBI_LENGTH_ERROR in case of an error. The return value is 0 for field types other than string or binary string. In case of an error the error number is one of DBI_ERROR_BADPTR, DBI_ERROR_BADOBJECT, or DBI_ERROR_BADIDX.

7.7.2. dbi_result_get_field_length_idx

```
size_t dbi_result_get_field_length_idx(dbi_result Result, unsigned int idx);
```

Returns the length of the value stored in the specified field which contains a string or a binary string.

Arguments

`Result`: The target query result.

`idx`: The index of the target field (starting at 1).

Returns

The length in bytes of the target field data, excluding the terminating zero byte, or DBI_LENGTH_ERROR in case of an error. The return value is 0 for field types other than string or binary string. In case of an error the error number is one of DBI_ERROR_BADPTR, DBI_ERROR_BADOBJECT, or DBI_ERROR_BADIDX.

7.7.3. dbi_result_get_field_size

```
size_t dbi_result_get_field_size(dbi_result Result, const char *fieldname);
```

Returns the size in bytes of the value stored in the specified field.

Note: This function is deprecated. Use dbi_result_get_field_length instead.

7.7.4. dbi_result_get_field_size_idx

```
size_t dbi_result_get_field_size_idx(dbi_result Result, unsigned long idx);
```

Returns the size in bytes of the value stored in the specified field.

Note: This function is deprecated. Use dbi_result_get_field_length_idx instead.

7.7.5. dbi_result_get_field_idx

```
unsigned int dbi_result_get_field_idx(dbi_result Result, const char *fieldname);
```

Given a field's name, return that field's numeric index.

Arguments

`Result`: The target query result.

`fieldname`: The name of the target field.

Returns

The index (starting at 1) of the target field, or 0 in case of an error. In that case, the error number is DBI_ERROR_BADPTR.

7.7.6. dbi_result_get_field_name

```
const char *dbi_result_get_field_name(dbi_result Result, unsigned int idx);
```

Given a field's numeric index, return that field's name.

Arguments

`Result`: The target query result.

`idx`: The index of the target field (starting at 1).

Returns

The target field's name, or NULL in case of an error. In that case the error number is one of DBI_ERROR_BADPTR, DBI_ERROR_BADOBJECT, or DBI_ERROR_BADIDX.

7.7.7. dbi_result_get_numfields

```
unsigned int dbi_result_get_numfields(dbi_result Result);
```

Returns the number of fields in the query result.

Arguments

`Result`: The target query result.

Returns

The number of fields in the query result, or DBI_FIELD_ERROR in case of an error.

7.7.8. dbi_result_get_field_type

```
unsigned short dbi_result_get_field_type(dbi_result Result, const char *fieldname);
```

Returns the target field's data type. The constants returned by this function are defined in dbi.h with the prefix "DBI_TYPE_".

Arguments

`Result`: The target query result.

`fieldname`: The target field's name.

Returns

The target field's data type, or DBI_TYPE_ERROR in case of an error. In the latter case the error number is DBI_ERROR_BADPTR, DBI_ERROR_BADOBJECT, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME.

7.7.9. dbi_result_get_field_type_idx

```
unsigned short dbi_result_get_field_type_idx(dbi_result Result, unsigned int idx);
```

Returns the target field's data type. The constants returned by this function are defined in dbi.h with the prefix "DBI_TYPE_".

Arguments

`Result`: The target query result.

`idx`: The index of the target field (starting at 1).

Returns

The target field's data type, or DBI_TYPE_ERROR in case of an error. In the latter case the error number is DBI_ERROR_BADPTR, DBI_ERROR_BADOBJECT, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME.

7.7.10. dbi_result_get_field_attrib

```
unsigned int dbi_result_get_field_attrib(dbi_result Result, const char *fieldname,
unsigned int attribmin, unsigned int attribmax);
```

Returns the target field's data type attributes in the specified range. The constants returned by this function are defined in `include/dbi/dbi.h` with the prefix "DBI_", followed by the name of the field's datatype.

Arguments

`Result`: The target query result.

`fieldname`: The target field's name.

`attribmin`: The first attribute value in the range of attributes to extract.

`attribmax`: The last attribute value in the range of attributes to extract. This may be the same as attribmin if you are only trying to extract a single attribute value.

Returns

The target field's requested attribute range, or DBI_ATTRIBUTE_ERROR in case of an error. In the latter case the error number is DBI_ERROR_BADPTR, DBI_ERROR_BADOBJECT, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME.

7.7.11. dbi_result_get_field_attrib_idx

```
unsigned int dbi_result_get_field_attrib_idx(dbi_result Result, unsigned int idx,
unsigned int attribmin, unsigned int attribmax);
```

Returns the target field's data type attributes in the specified range. The constants returned by this function are defined in dbi.h with the prefix "DBI_", followed by the name of the field's datatype.

Arguments

`Result`: The target query result.

`idx`: The index of the target field (starting at 1).

`attribmin`: The first attribute value in the range of attributes to extract.

`attribmax`: The last attribute value in the range of attributes to extract. This may be the same as attribmin if you are only trying to extract a single attribute value.

Returns

The target field's requested attribute range, or DBI_ATTRIBUTE_ERROR in case of an error. In the latter case the error number is DBI_ERROR_BADPTR, DBI_ERROR_BADOBJECT, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME.

7.7.12. dbi_result_get_field_attribs

```
unsigned int dbi_result_get_field_attribs(dbi_result Result, const char *fieldname);
```

Returns the target field's data type attributes. The constants returned by this function are defined in dbi.h with the prefix "DBI_", followed by the name of the field's datatype.

Arguments

`Result`: The target query result.

`fieldname`: The target field's name.

Returns

The target field's attributes, or DBI_ATTRIBUTE_ERROR in case of an error. In case of an error the error number is DBI_ERROR_BADPTR, DBI_ERROR_BADOBJECT, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME.

7.7.13. dbi_result_get_field_attribs_idx

```
unsigned int dbi_result_get_field_attribs_idx(dbi_result Result, unsigned int
fieldidx);
```

Returns the target field's data type attributes. The constants returned by this function are defined in dbi.h with the prefix "DBI_", followed by the name of the field's datatype.

Arguments

`Result`: The target query result.

`fieldidx`: The index of the target field (starting at 1).

Returns

The target field's attributes, or DBI_ATTRIBUTE_ERROR in case of an error. In that case the error number is DBI_ERROR_BADPTR, DBI_ERROR_BADOBJECT, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME.

7.7.14. dbi_result_field_is_null

```
int dbi_result_field_is_null(dbi_result Result, const char *fieldname);
```

Determines whether the indicated field contains a NULL value.

Arguments

`Result`: The target query result.

`fieldname`: The name of the target field.

Returns

1 if the field contains a NULL value, otherwise 0, or DBI_FIELD_FLAG_ERROR in case of an error. In the latter case the error number is DBI_ERROR_BADPTR, DBI_ERROR_BADOBJECT, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME.

7.7.15. dbi_result_field_is_null_idx

```
int dbi_result_field_is_null_idx(dbi_result Result, unsigned int fieldidx);
```

Determines whether the indicated field contains a NULL value.

Arguments

`Result`: The target query result.

`fieldidx`: The index of the target field (starting at 1).

Returns

1 if the field contains a NULL value, otherwise 0, or DBI_FIELD_FLAG_ERROR in case of an error. In the latter case the error number is DBI_ERROR_BADPTR, DBI_ERROR_BADOBJECT, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME.

7.8. Retrieving Field Data by Name

7.8.1. dbi_result_get_fields

```
unsigned int dbi_result_get_fields(dbi_result Result, const char *format, ...);
```

Fetch multiple fields from the current result set, using a printf-like syntax. The formatter string specified field names and types, and each field's associated destination variable is passed as an argument following the format string. Fields in the formatter string are separated by spaces, and follow the format `"a.%b"`, where "a" is the name of the field, and "b" is the field type specifier. Make sure you pass the destination variables' memory addresses by prepending the & operator to each variable's name.

Field type specifiers:

- `%c / %uc`: A signed/unsigned character (1-byte)
- `%h / %uh`: A signed/unsigned short integer (2-byte)
- `%l / %ul`: A signed/unsigned integer (4-byte)
- `%i / %ui`: A signed/unsigned integer (4-byte)
- `%L / %uL`: A signed/unsigned long long integer (8-byte)
- `%f`: A floating point number
- `%d`: A double-precision number
- `%s`: A read-only string
- `%S`: A local copy of a string (must be freed by program)
- `%b`: A read-only pointer to binary data
- `%B`: A local copy of binary data (must be freed by program)
- `%m`: A time_t value representing a DATE and/or TIME

Example usage: `dbi_result_get_fields(result, "idnum.%ul lastname.%s", &id_number, &name)`

Arguments

`Result`: The target query result.

`format`: The field format string as described above.

`ARG`: (...) Pointers to the destination variables corresponding with each field in the format string.

Returns

The number of fields fetched, or DBI_FIELD_ERROR if there was an error. If an invalid field name was specified it will not raise an error, and the other fetched fields will work as usual.

7.8.2. dbi_result_bind_fields

```
unsigned int dbi_result_bind_fields(dbi_result Result, const char *format, ...);
```

Bind multiple fields in the current result set, using a printf-like syntax. See dbi_result_get_fields for a detailed explanation of the syntax.

Arguments

`Result`: The target query result.

`format`: The field format string as described above.

`ARG`: (...) Pointers to the destination variables corresponding with each field in the format string.

Returns

The number of field binding set up, or DBI_FIELD_ERROR if there was an error.

7.8.3. dbi_result_get_char

```
signed char dbi_result_get_char(dbi_result Result, const char *fieldname);
```

Fetch the data stored in the speficied field, which contains a character (a 1-byte signed integer). This is the default for the "char" type on the x86 platform, as well as on Mac OS X.

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to fetch.

Returns

The data stored in the specified field, or 0 (zero) if an error occurs. In the latter case the error number is DBI_ERROR_BADTYPE, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME.

7.8.4. dbi_result_get_uchar

```
unsigned char dbi_result_get_uchar(dbi_result Result, const char *fieldname);
```

Fetch the data stored in the speficied field, which contains an unsigned character (1-byte unsigned integer). This is the default for the "char" type on Linux for PowerPC.

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to fetch.

Returns

The data stored in the specified field, or 0 (zero) if an error occurs. In the latter case the error number is DBI_ERROR_BADTYPE, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME.

7.8.5. dbi_result_get_short

```
short dbi_result_get_short(dbi_result Result, const char *const char *fieldname);
```

Fetch the data stored in the speficied field, which contains a short integer (2-byte signed integer).

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to fetch.

Returns

The data stored in the specified field, or 0 (zero) if an error occurs. In the latter case the error number is DBI_ERROR_BADTYPE, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME..

7.8.6. dbi_result_get_ushort

```
unsigned short dbi_result_get_ushort(dbi_result Result, const char *fieldname);
```

Fetch the data stored in the speficied field, which contains an unsigned short integer (2-byte unsigned integer).

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to fetch.

Returns

The data stored in the specified field, or 0 (zero) if an error occurs. In the latter case the error number is DBI_ERROR_BADTYPE, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME..

7.8.7. dbi_result_get_int

```
int dbi_result_get_int(dbi_result Result, const char *fieldname);
```

Fetch the data stored in the speficied field, which contains an integer (4-byte signed integer).

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to fetch.

Returns

The data stored in the specified field, or 0 (zero) if an error occurs. In the latter case the error number is DBI_ERROR_BADTYPE, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME..

Availability

0.8.0

7.8.8. dbi_result_get_uint

```
unsigned int dbi_result_get_uint(dbi_result Result, const char *fieldname);
```

Fetch the data stored in the speficied field, which contains an unsigned integer (4-byte unsigned integer).

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to fetch.

Returns

The data stored in the specified field, or 0 (zero) if an error occurs. In the latter case the error number is DBI_ERROR_BADTYPE, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME..

Availability

0.8.0

7.8.9. dbi_result_get_long

```
int dbi_result_get_long(dbi_result Result, const char *fieldname);
```

This is the same as dbi_result_get_int. The use of this function is deprecated as the name implies the wrong return type on 64-bit platforms.

7.8.10. dbi_result_get_ulong

```
unsigned int dbi_result_get_ulong(dbi_result Result, const char *fieldname);
```

This is the same as dbi_result_get_uint. The use of this function is deprecated as the name implies the wrong return type on 64-bit platforms.

7.8.11. dbi_result_get_longlong

```
long long dbi_result_get_longlong(dbi_result Result, const char *fieldname);
```

Fetch the data stored in the speficied field, which contains a long long integer (8-byte signed integer).

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to fetch.

Returns

The data stored in the specified field, or 0 (zero) if an error occurs. In the latter case the error number is DBI_ERROR_BADTYPE, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME..

7.8.12. dbi_result_get_ulonglong

```
unsigned long long dbi_result_get_ulonglong(dbi_result Result, const char *fieldname);
```

Fetch the data stored in the speficied field, which contains an unsigned long long integer (8-byte unsigned integer).

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to fetch.

Returns

The data stored in the specified field, or 0 (zero) if an error occurs. In the latter case the error number is DBI_ERROR_BADTYPE, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME..

7.8.13. dbi_result_get_float

```
float dbi_result_get_float(dbi_result Result, const char *fieldname);
```

Fetch the data stored in the speficied field, which contains a floating-point number.

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to fetch.

Returns

The data stored in the specified field, which contains a fractional number, or 0 (zero) if an error occurs. In the latter case the error number is DBI_ERROR_BADTYPE, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME..

7.8.14. dbi_result_get_double

```
double dbi_result_get_double(dbi_result Result, const char *fieldname);
```

Fetch the data stored in the speficied field, which contains a double-precision fractional number.

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to fetch.

Returns

The data stored in the specified field, or 0 (zero) if an error occurs. In the latter case the error number is DBI_ERROR_BADTYPE, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME..

7.8.15. dbi_result_get_string

```
const char *dbi_result_get_string(dbi_result Result, const char *fieldname);
```

Fetch the data stored in the speficied field, which contains a zero-terminated string.

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to fetch.

Returns

The data stored in the specified field, which is a zero-terminated string. If the field contains a NULL value, the function returns a NULL pointer. The string may not be modified, and may not necessarily persist between row fetches. In case of an error, this function returns the string "ERROR". In the latter case the error number is DBI_ERROR_BADTYPE, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME.

7.8.16. dbi_result_get_string_copy

```
char *dbi_result_get_string_copy(dbi_result Result, const char *fieldname);
```

Fetch the data stored in the speficied field, which contains a zero-terminated string, and return it in an allocated buffer.

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to fetch.

Returns

The data stored in the specified field as a zero-terminated allocated string. If the field contains a NULL value, the function returns a NULL pointer, and no memory is allocated. The newly allocated string may be modified by the host program, but the program is responsible for freeing the string. In case of an error, this function returns an allocated string reading "ERROR". In that case the error number is DBI_ERROR_BADTYPE, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME.

7.8.17. dbi_result_get_binary

```
const unsigned char *dbi_result_get_binary(dbi_result Result, const char *fieldname);
```

Fetch the data stored in the speficied field, which contains binary data.

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to fetch.

Returns

The data stored in the specified field. The binary data may contain zero bytes and non-printable characters. Use dbi_result_get_field_length or dbi_result_get_field_length_idx to determine the number of bytes contained in the resulting binary string. The data may not be modified, and may not necessarily persist between row fetches. If the field contains a NULL value, the function returns a NULL pointer. In case of an error, this function returns the string "ERROR". In that case the error number is DBI_ERROR_BADTYPE, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME.

7.8.18. dbi_result_get_binary_copy

```
unsigned char *dbi_result_get_binary_copy(dbi_result Result, const char *fieldname);
```

Fetch the data stored in the speficied field, which contains binary data, and return it in an allocated buffer.

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to fetch.

Returns

The data stored in the specified field. The binary data may contain zero bytes and non-printable characters. Use dbi_result_get_field_length or dbi_result_get_field_length_idx to determine the number of bytes contained in the resulting binary string. The newly allocated memory may be modified by the host program, but the program is responsible for freeing the data. If the field contains a NULL value, the function returns a NULL pointer. In case of an error, this function returns the string "ERROR". In that case the error number is DBI_ERROR_BADTYPE, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME.

7.8.19. dbi_result_get_datetime

```
time_t dbi_result_get_datetime(dbi_result Result, const char *fieldname);
```

Fetch the data stored in the specified field, which contains a DATE and/or TIME value.

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to fetch.

Returns

The data stored in the specified field as a time_t value. To convert this into human-readable dates or times, use the C library functions gmtime (3) and localtime (3). In case of an error this function returns 0 (zero) which resolves to the Unix epoch when converted. In case of an error the error number is DBI_ERROR_BADTYPE, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME.

7.8.20. dbi_result_get_as_longlong

```
long long dbi_result_get_as_longlong(dbi_result Result, const char *fieldname);
```

Fetch the data stored in the specified field. Return the contents as a long long integer value, using appropriate casts or conversions if applicable.

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to fetch.

Returns

The data stored in the specified field as a long long integer. Integer and floating point data as well as datetime data are cast to long long. Strings are converted using strtoll(). Empty strings, strings that do not translate into an integer, and binary strings are returned as 0 (zero) without raising an error. In case of an error this function returns 0 (zero) and the error number is DBI_ERROR_BADTYPE, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME.

Availability

0.9.0

7.8.21. dbi_result_get_as_string_copy

```
char *dbi_result_get_as_string_copy(dbi_result Result, const char *fieldname);
```

Fetch the data stored in the specified field. Return the contents as an allocated string, using appropriate conversions if applicable. The caller is responsible for freeing the returned buffer when done.

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to fetch.

Returns

A string representation of the data stored in the specified field. Integer, floating point and datetime data are pretty-printed using snprintf(). Strings are returned as such. Empty strings and binary strings are returned as empty strings without raising an error. In case of an error this function returns the string "ERROR" and the error number is DBI_ERROR_NOMEM, DBI_ERROR_BADTYPE, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME.

Availability

0.9.0

7.8.22. dbi_result_bind_char

```
int dbi_result_bind_char(dbi_result Result, const char *fieldname, char *bindto);
```

Bind the specified variable to the specified field, which holds a character (a 1-byte signed integer). This is the default for the "char" type on the x86 platform, as well as on Mac OS X.

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to bind to.

`bindto`: A pointer to the variable that will be updated with the specified field's value.

Returns

0 upon success, DBI_BIND_ERROR if there was an error. Possible error numbers are DBI_ERROR_BADPTR, DBI_ERROR_NOMEM, and DBI_ERROR_BADNAME.

7.8.23. dbi_result_bind_uchar

```
int dbi_result_bind_uchar(dbi_result Result, const char *fieldname, unsigned char
*bindto);
```

Bind the specified variable to the specified field, which holds an unsigned character (1-byte unsigned integer). This is the default for the "char" type on Linux for PowerPC.

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to bind to.

`bindto`: A pointer to the variable that will be updated with the specified field's value.

Returns

0 upon success, DBI_BIND_ERROR if there was an error. Possible error numbers are DBI_ERROR_BADPTR, DBI_ERROR_NOMEM, and DBI_ERROR_BADNAME.

7.8.24. dbi_result_bind_short

```
int dbi_result_bind_short(dbi_result Result, const char *fieldname, short *bindto);
```

Bind the specified variable to the specified field, which holds a short integer (2-byte signed integer).

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to bind to.

`bindto`: A pointer to the variable that will be updated with the specified field's value.

Returns

0 upon success, DBI_BIND_ERROR if there was an error. Possible error numbers are DBI_ERROR_BADPTR, DBI_ERROR_NOMEM, and DBI_ERROR_BADNAME.

7.8.25. dbi_result_bind_ushort

```
int dbi_result_bind_ushort(dbi_result Result, const char *fieldname, unsigned short
*bindto);
```

Bind the specified variable to the specified field, which holds an unsigned short integer (2-byte unsigned integer).

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to bind to.

`bindto`: A pointer to the variable that will be updated with the specified field's value.

Returns

0 upon success, DBI_BIND_ERROR if there was an error. Possible error numbers are DBI_ERROR_BADPTR, DBI_ERROR_NOMEM, and DBI_ERROR_BADNAME.

7.8.26. dbi_result_bind_int

```
int dbi_result_bind_int(dbi_result Result, const char *fieldname, long *bindto);
```

Bind the specified variable to the specified field, which holds an integer (4-byte signed integer).

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to bind to.

`bindto`: A pointer to the variable that will be updated with the specified field's value.

Returns

0 upon success, DBI_BIND_ERROR if there was an error. Possible error numbers are DBI_ERROR_BADPTR, DBI_ERROR_NOMEM, and DBI_ERROR_BADNAME.

Availability

0.8.0

7.8.27. dbi_result_bind_uint

```
int dbi_result_bind_uint(dbi_result Result, const char *fieldname, unsigned long
*bindto);
```

Bind the specified variable to the specified field, which holds an unsigned long integer (4-byte unsigned integer).

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to bind to.

`bindto`: A pointer to the variable that will be updated with the specified field's value.

Returns

0 upon success, DBI_BIND_ERROR if there was an error. Possible error numbers are DBI_ERROR_BADPTR, DBI_ERROR_NOMEM, and DBI_ERROR_BADNAME.

Availability

0.8.0

7.8.28. dbi_result_bind_long

```
int dbi_result_bind_long(dbi_result Result, const char *fieldname, long *bindto);
```

The same as dbi_result_bind_int. The use of this function is deprecated as the name implies the wrong return type on 64-bit platforms.

7.8.29. dbi_result_bind_ulong

```
int dbi_result_bind_ulong(dbi_result Result, const char *fieldname, unsigned long
*bindto);
```

The same as dbi_result_bind_uint. The use of this function is deprecated as the name implies the wrong return type on 64-bit platforms.

7.8.30. dbi_result_bind_longlong

```
int dbi_result_bind_longlong(dbi_result Result, const char *fieldname, long long
*bindto);
```

Bind the specified variable to the specified field, which holds a long long integer (8-byte signed integer).

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to bind to.

`bindto`: A pointer to the variable that will be updated with the specified field's value.

Returns

0 upon success, DBI_BIND_ERROR if there was an error. Possible error numbers are DBI_ERROR_BADPTR, DBI_ERROR_NOMEM, and DBI_ERROR_BADNAME.

7.8.31. dbi_result_bind_ulonglong

```
int dbi_result_bind_ulonglong(dbi_result Result, const char *fieldname, unsigned long
long *bindto);
```

Bind the specified variable to the specified field, which holds an unsigned long long integer (8-byte unsigned integer).

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to bind to.

`bindto`: A pointer to the variable that will be updated with the specified field's value.

Returns

0 upon success, DBI_BIND_ERROR if there was an error. Possible error numbers are DBI_ERROR_BADPTR, DBI_ERROR_NOMEM, and DBI_ERROR_BADNAME.

7.8.32. dbi_result_bind_float

```
int dbi_result_bind_float(dbi_result Result, const char *fieldname, float *bindto);
```

Bind the specified variable to the specified field, which holds a floating-point number.

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to bind to.

`bindto`: A pointer to the variable that will be updated with the specified field's value.

Returns

0 upon success, DBI_BIND_ERROR if there was an error. Possible error numbers are DBI_ERROR_BADPTR, DBI_ERROR_NOMEM, and DBI_ERROR_BADNAME.

7.8.33. dbi_result_bind_double

```
int dbi_result_bind_double(dbi_result Result, const char *fieldname, double *bindto);
```

Bind the specified variable to the specified field, which holds a double-precision fractional number.

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to bind to.

`bindto`: A pointer to the variable that will be updated with the specified field's value.

Returns

0 upon success, DBI_BIND_ERROR if there was an error. Possible error numbers are DBI_ERROR_BADPTR, DBI_ERROR_NOMEM, and DBI_ERROR_BADNAME.

7.8.34. dbi_result_bind_string

```
int dbi_result_bind_string(dbi_result Result, const char *fieldname, const char
**bindto);
```

Bind the specified variable to the specified field, which holds a string. The string must not be modified.

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to bind to.

`bindto`: A pointer to the variable that will be updated with the specified field's value.

Returns

0 upon success, DBI_BIND_ERROR if there was an error. Possible error numbers are DBI_ERROR_BADPTR, DBI_ERROR_NOMEM, and DBI_ERROR_BADNAME.

7.8.35. dbi_result_bind_binary

```
int dbi_result_bind_binary(dbi_result Result, const char *fieldname, const unsigned
char **bindto);
```

Bind the specified variable to the specified field, which holds binary BLOB data. The data must not be modified.

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to bind to.

`bindto`: A pointer to the variable that will be updated with the specified field's value.

Returns

0 upon success, DBI_BIND_ERROR if there was an error. Possible error numbers are DBI_ERROR_BADPTR, DBI_ERROR_NOMEM, and DBI_ERROR_BADNAME.

7.8.36. dbi_result_bind_string_copy

```
int dbi_result_bind_string_copy(dbi_result Result, const char *fieldname, char
**bindto);
```

Bind the specified variable to the specified field, which holds a string. The newly allocated string may be modified by the host program, but the program is responsible for freeing the string.

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to bind to.

`bindto`: A pointer to the variable that will be updated with the specified field's value.

Returns

0 upon success, DBI_BIND_ERROR if there was an error. Possible error numbers are DBI_ERROR_BADPTR, DBI_ERROR_NOMEM, and DBI_ERROR_BADNAME.

7.8.37. dbi_result_bind_binary_copy

```
int dbi_result_bind_binary_copy(dbi_result Result, const char *fieldname, unsigned
char **bindto);
```

Bind the specified variable to the specified field, which holds binary BLOB data. The newly allocated data may be modified by the host program, but the program is responsible for freeing the data.

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to bind to.

`bindto`: A pointer to the variable that will be updated with the specified field's value.

Returns

0 upon success, DBI_BIND_ERROR if there was an error. Possible error numbers are DBI_ERROR_BADPTR, DBI_ERROR_NOMEM, and DBI_ERROR_BADNAME.

7.8.38. dbi_result_bind_datetime

```
int dbi_result_bind_datetime(dbi_result Result, const char *fieldname, time_t
*bindto);
```

Bind the specified variable to the specified field, which holds a DATE and/or TIME value.

Arguments

`Result`: The target query result.

`fieldname`: The name of the field to bind to.

`bindto`: A pointer to the variable that will be updated with the specified field's value.

Returns

0 upon success, DBI_BIND_ERROR if there was an error. Possible error numbers are DBI_ERROR_BADPTR, DBI_ERROR_NOMEM, and DBI_ERROR_BADNAME.

7.9. Retrieving Field Data by Index

7.9.1. dbi_result_get_char_idx

```
signed char dbi_result_get_char_idx(dbi_result Result, unsigned int fieldidx);
```

Fetch the data stored in the speficied field, which contains a character (a 1-byte signed integer). This is the default for the "char" type on the x86 platform, as well as on Mac OS X.

Arguments

`Result`: The target query result.

`fieldidx`: The index of the target field (starting at 1).

Returns

The data stored in the specified field, or 0 (zero) if an error occurs. In the latter case the error number is DBI_ERROR_BADTYPE or DBI_ERROR_BADIDX.

7.9.2. dbi_result_get_uchar_idx

```
unsigned char dbi_result_get_uchar_idx(dbi_result Result, unsigned int fieldidx);
```

Fetch the data stored in the speficied field, which contains an unsigned character (1-byte unsigned integer). This is the default for the "char" type on Linux for PowerPC.

Arguments

`Result`: The target query result.

`fieldidx`: The index of the target field (starting at 1).

Returns

The data stored in the specified field, or 0 (zero) if an error occurs. In the latter case the error number is DBI_ERROR_BADTYPE or DBI_ERROR_BADIDX.

7.9.3. dbi_result_get_short_idx

```
short dbi_result_get_short_idx(dbi_result Result, unsigned int fieldidx);
```

Fetch the data stored in the speficied field, which contains a short integer (2-byte signed integer).

Arguments

`Result`: The target query result.

`fieldidx`: The index of the target field (starting at 1).

Returns

The data stored in the specified field, or 0 (zero) if an error occurs. In the latter case the error number is DBI_ERROR_BADTYPE or DBI_ERROR_BADIDX.

7.9.4. dbi_result_get_ushort_idx

```
unsigned short dbi_result_get_ushort_idx(dbi_result Result, unsigned int fieldidx);
```

Fetch the data stored in the speficied field, which contains an unsigned short integer (2-byte unsigned integer).

Arguments

`Result`: The target query result.

`fieldidx`: The index of the target field (starting at 1).

Returns

The data stored in the specified field, or 0 (zero) if an error occurs. In the latter case the error number is DBI_ERROR_BADTYPE or DBI_ERROR_BADIDX.

7.9.5. dbi_result_get_int_idx

```
int dbi_result_get_int_idx(dbi_result Result, unsigned int fieldidx);
```

Fetch the data stored in the speficied field, which contains an integer (4-byte signed integer).

Arguments

`Result`: The target query result.

`fieldidx`: The index of the target field (starting at 1).

Returns

The data stored in the specified field, or 0 (zero) if an error occurs. In the latter case the error number is DBI_ERROR_BADTYPE or DBI_ERROR_BADIDX.

Availability

0.8.0

7.9.6. dbi_result_get_uint_idx

```
unsigned int dbi_result_get_uint_idx(dbi_result Result, unsigned int fieldidx);
```

Fetch the data stored in the speficied field, which contains an unsigned integer (4-byte signed integer).

Arguments

`Result`: The target query result.

`fieldidx`: The index of the target field (starting at 1).

Returns

The data stored in the specified field, or 0 (zero) if an error occurs. In the latter case the error number is DBI_ERROR_BADTYPE or DBI_ERROR_BADIDX.

Availability

0.8.0

7.9.7. dbi_result_get_long_idx

```
int dbi_result_get_long_idx(dbi_result Result, unsigned int fieldidx);
```

Same as dbi_result_get_int_idx. This function is deprecated as the name implies the wrong return type on 64bit platforms.

7.9.8. dbi_result_get_ulong_idx

```
unsigned int dbi_result_get_ulong_idx(dbi_result Result, unsigned int fieldidx);
```

Same as dbi_result_get_uint_idx. This function is deprecated as the name implies the wrong return type on 64bit platforms.

7.9.9. dbi_result_get_longlong_idx

```
long long dbi_result_get_longlong_idx(dbi_result Result, unsigned int fieldidx);
```

Fetch the data stored in the speficied field, which contains a long long integer (8-byte signed integer).

Arguments

`Result`: The target query result.

`fieldidx`: The index of the target field (starting at 1).

Returns

The data stored in the specified field, or 0 (zero) if an error occurs. In the latter case the error number is DBI_ERROR_BADTYPE or DBI_ERROR_BADIDX.

7.9.10. dbi_result_get_ulonglong_idx

```
unsigned long long dbi_result_get_ulonglong_idx(dbi_result Result, unsigned int fieldidx);
```

Fetch the data stored in the speficied field, which contains an unsigned long long integer (8-byte unsigned integer).

Arguments

`Result`: The target query result.

`fieldidx`: The index of the target field (starting at 1).

Returns

The data stored in the specified field, or 0 (zero) if an error occurs. In the latter case the error number is DBI_ERROR_BADTYPE or DBI_ERROR_BADIDX.

7.9.11. dbi_result_get_float_idx

```
float dbi_result_get_float_idx(dbi_result Result, unsigned int fieldidx);
```

Fetch the data stored in the speficied field, which contains a floating-point number.

Arguments

`Result`: The target query result.

`fieldidx`: The index of the target field (starting at 1).

Returns

The data stored in the specified field, or 0 (zero) if an error occurs. In the latter case the error number is DBI_ERROR_BADTYPE or DBI_ERROR_BADIDX.

7.9.12. dbi_result_get_double_idx

```
double dbi_result_get_double_idx(dbi_result Result, unsigned int fieldidx);
```

Fetch the data stored in the speficied field, which contains a double-precision fractional number.

Arguments

`Result`: The target query result.

`fieldidx`: The index of the target field (starting at 1).

Returns

The data stored in the specified field, or 0 (zero) if an error occurs. In the latter case the error number is DBI_ERROR_BADTYPE or DBI_ERROR_BADIDX.

7.9.13. dbi_result_get_string_idx

```
const char *dbi_result_get_string_idx(dbi_result Result, unsigned int fieldidx);
```

Fetch the data stored in the speficied field, which contains a zero-terminated string.

Arguments

`Result`: The target query result.

`fieldidx`: The index of the target field (starting at 1).

Returns

The data stored in the specified field. If the field contains a NULL value, the function returns a NULL pointer. The string may not be modified, and may not necessarily persist between row fetches. In case of an error, this function returns the string "ERROR". In the latter case the error number is DBI_ERROR_BADTYPE, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME.

7.9.14. dbi_result_get_string_copy_idx

```
char *dbi_result_get_string_copy_idx(dbi_result Result, unsigned int fieldidx);
```

Fetch the data stored in the speficied field, which contains a zero-terminated string, and return it in an allocated buffer.

Arguments

`Result`: The target query result.

`fieldidx`: The index of the target field (starting at 1).

Returns

The data stored in the specified field. If the field contains a NULL value, the function returns a NULL pointer, and no memory is allocated. The newly allocated string may be modified by the host program, but the program is responsible for freeing the string. In case of an error, this function returns an allocated string reading "ERROR". In the latter case the error number is DBI_ERROR_BADTYPE, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME.

7.9.15. dbi_result_get_binary_idx

```
const unsigned char *dbi_result_get_binary_idx(dbi_result Result, unsigned int
fieldidx);
```

Fetch the data stored in the speficied field, which contains binary data. The data may not be modified, and may not necessarily persist between row fetches. If the field contains a NULL value, the function returns a NULL pointer. In case of an error, this function returns the string "ERROR".

Arguments

`Result`: The target query result.

`fieldidx`: The index of the target field (starting at 1).

Returns

The data stored in the specified field. The binary data may contain zero bytes and non-printable characters. Use dbi_result_get_field_length or dbi_result_get_field_length_idx to determine the number of bytes contained in the resulting binary string. The data may not be modified, and may not necessarily persist between row fetches. If the field contains a NULL value, the function returns a NULL pointer. In case of an error, this function returns the string "ERROR". In the latter case the error number is DBI_ERROR_BADTYPE, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME.

7.9.16. dbi_result_get_binary_copy_idx

```
unsigned char *dbi_result_get_binary_copy_idx(dbi_result Result, unsigned int
fieldidx);
```

Fetch the data stored in the speficied field, which contains binary data, and return it in an allocated buffer. The newly allocated memory may be modified by the host program, but the program is responsible for freeing the data. If the field contains a NULL value, the function returns a NULL pointer. In case of an error, this function returns the string "ERROR".

Arguments

`Result`: The target query result.

`fieldidx`: The index of the target field (starting at 1).

Returns

The data stored in the specified field. The newly allocated memory may be modified by the host program, but the program is responsible for freeing the data. If the field contains a NULL value, the function returns a NULL pointer. In case of an error, this function returns the string "ERROR". In the latter case the error number is DBI_ERROR_BADTYPE, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME.

7.9.17. dbi_result_get_datetime_idx

```
time_t dbi_result_get_datetime_idx(dbi_result Result, unsigned int fieldidx);
```

Fetch the data stored in the specified field, which contains a DATE and/or TIME value.

Arguments

`Result`: The target query result.

`fieldidx`: The index of the target field (starting at 1).

Returns

The data stored in the specified field as a time_t value. To convert this into human-readable dates or times, use the C library functions gmtime (3) and localtime (3). In case of an error this function returns 0 (zero) which resolves to the Unix epoch when converted. In case of an error the error number is DBI_ERROR_BADTYPE, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME.

7.9.18. dbi_result_get_as_longlong_idx

```
long long dbi_result_get_as_longlong_idx(dbi_result Result, unsigned int fieldidx);
```

Fetch the data stored in the specified field. Return the contents as a long long integer value, using appropriate casts or conversions if applicable.

Arguments

`Result`: The target query result.

`fieldidx`: The index of the target field (starting at 1).

Returns

The data stored in the specified field as a long long integer. Integer and floating point data as well as datetime data are cast to long long. Strings are converted using strtoll(). Empty strings, strings that do not translate into an integer, and binary strings are returned as 0 (zero) without raising an error. In case of an error this function returns 0 (zero) and the error number is DBI_ERROR_BADTYPE, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME.

Availability

0.9.0

7.9.19. dbi_result_get_as_string_copy_idx

```
char *dbi_result_get_as_string_copy_idx(dbi_result Result, unsigned int fieldidx);
```

Fetch the data stored in the specified field. Return the contents as an allocated string, using appropriate conversions if applicable. The caller is responsible for freeing the returned buffer when done.

Arguments

`Result`: The target query result.

`fieldidx`: The index of the target field (starting at 1).

Returns

A string representation of the data stored in the specified field. Integer, floating point and datetime data are pretty-printed using snprintf(). Strings are returned as such. Empty strings and binary strings are returned as empty strings without raising an error. In case of an error this function returns the string "ERROR" and the error

number is DBI_ERROR_NOMEM, DBI_ERROR_BADTYPE, DBI_ERROR_BADIDX, or DBI_ERROR_BADNAME.

Availability

0.9.0

Appendix A. GNU Free Documentation License

Version 1.1, March 2000

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has

been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document’s Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.