0 true in

minIni

Version 1.5
January 2024

MINIINI is a programmer's library to read and write "INI" files in embedded systems. MINIINI takes little resources, has a deterministic memory footprint, and can be configured for various kinds of file I/O libraries.

The principal purpose for MINIINI is to be used on embedded systems that run on an RTOS (or even without any operating system). MINIINI requires that such a system provides a kind of storage and file I/O system, but it does not require that this file I/O system is compatible with the standard C/C++ library —indeed, the standard library is often too big and resource-hungry for embedded systems.

Contents

Trademarks

Copyright

The MININI library was derived in part from the article "Multiplatform .INI Files" by Joseph J. Graf in the March 1994 issue of Dr. Dobb's Journal.

Introduction

minIni is a library to read and write simple configuration files with a format compatible with "ini" files. The minIni library features a small code size footprint and it requires little resources (e.g. RAM). It is therefore suitable for use in (small) embedded systems.

The ini file format is best known from Microsoft Windows, with its functions `GetProfileString` and `WriteProfileString` to read from and write to ini files. The functions in minIni are modelled after the functions of the Windows SDK, but they are not fully compatible with them.

Although the main asset of minIni is that it is small and minimal, it has a few other features:

- minIni supports reading keys that are outside a section, and it thereby supports configuration files that do not use sections (but that are otherwise compatible with ini files).
- Section and key enumeration are supported.
- You may use a colon to separate key and value; the colon is equivalent to the equal sign. That is, the strings "`Name: Value`" and "`Name=Value`" have the same meaning.
- Trailing comments (i.e. behind a key/value pair on a line) are allowed. The hash character ("`#`") is an alternative for the semicolon to start a comment.
- When writing a value that contains a comment character ("`;`" or "`#`"), that value will automatically be put between double quotes; when reading the value, these quotes are removed. When a double-quote itself appears in the setting, these characters are escaped.
- Flexible rational number support, either fixed-point or floating-point.
- Since writing speed is *much* lower than reading speed in Flash memory (SD/MMC cards, USB memory sticks), minIni caches "file writes" to optimize performance, and it does so in a way that does not require extra memory.
- The memory footprint is deterministic. There is no dynamic memory allocation.

Limitations

minIni's design is aimed at being full-featured while using a small and *deterministic* memory footprint. It is not optimized for speed. On performance-sensitive code, I advice to read any values or settings that are needed *ahead* of the time and to store these in variables.

Specifically, minIni does *not* cache any key/value pairs that it reads from the ini file. It also does *not* keep the ini file open between calls; minIni closes the file after every read or write operation.

When writing to an ini file, minIni creates a temporary file into which it copies (with modifications) the original file. On success, it deletes the original file and renames the

temporary file back. If several settings must be changed, this cycle repeats for every setting.

There is no inherent file locking mechanism that protects against multiple applications (or threads, or tasks) accessing the same INI file. If an INI file is shared accross multiple programs/threads/tasks, see section Multi-tasking on page 9 for tips and options.

INI file syntax

An INI file has a simple syntax with name/value pairs in a plain text file. The name must be unique (per section) and the value must fit on a single line. An INI file is commonly separated into sections —in MINIINI, this is optional. A section is a name between square brackets, like "`[Network]`" in the example below.

LISTING: Example INI file

```
[Network]
hostname = My Computer
address = dhcp
dns = 192.168.1.1
```

In the API and in this documentation, the "name" for a setting is denoted as the *key* for the setting. The key and the value are separated by an equal sign ("`=`"). MINIINI supports the colon ("`:`") as an alternative to the equal sign for the key/value delimiter.

Section and key name comparisons are case insensitive in MINIINI (as is the case in the Microsoft Windows API). Therefore, in the INI file you may type "`DNS = 192.168.1.1`" just as well as "`dns = 192.168.1.1`".

Leading a trailing spaces around values or key names are removed. If you need to include leading and/or trailing spaces in a value, put the value between double quotes. The `ini_gets` function strips off the double quotes from the returned value. Function `ini_puts` adds double quotes if the value to write contains trailing white space (or special characters).

MINIINI ignores white space characters around the "`=`" or "`:`" delimiters, as well as spaces after the opening bracket "`[`" of a section and before the matching closing bracket "`]`". It does not remove spaces inside key or section names. Key names and section names may therefore have embedded spaces.

Comments in the INI must start with a semicolon ("`;`") or a hash character ("`#`"), and run to the end of the line. A comment can be a line of its own, or it may follow a key/value pair.

There is only a single hierarchy in an INI file: section and key. There have been a few proposals for deeper hierarchies, like adding sections "`[Network.eth0]`" and "`[Network.wifi0]`" (possibly in addition to "`[Network]`" for the general settings). The "`.`" would sometimes be a "/". In MINIINI, this syntax is allowed as well, for section names

as well as for key names. However, it is purely a naming convention: MINIni does not handle the dot (or slash) in a special way.

Using minIni

The first step in using MINIINI is making sure that it compiles. The library consists of only one C file and two header files, so the amount of configuration to do is minimal. If you cannot use the standard C/C++ library, there is, however, a configuration file (or "glue" file) that you must make or customize; this file is explained in the next section. The MINIINI distribution comes with a default configuration file that maps to the standard C library (specifically the file I/O functions from the "`stdio`" package) and example glue files for a few embedded file system libraries for embedded systems —see appendix A of this manual.

Once you have a good glue file, you can add the source file of MINIINI to your project and include the header file "`minIni.h`" in your source code files. In your source code, you can then use the functions in the MINIINI library to read text and values from INI files and to write text and values to an INI file. See the Function reference for details.

MINIINI uses string functions from the standard C/C++ library, including one function that is not in the ANSI C standard: `strnicmp`. On the Unix and Linux platforms, this function is usually called `strncasecmp`. If you are using a GNU GCC compiler, but you are not compiling for a Linux or "BSD" platform, you may need to define `strnicmp` as `strncasecmp` in the glue file (see below). If your compiler provides neither `strnicmp` nor `strncasecmp`, you can use a portable implementation in MINIINI by defining the macro `PORTABLE_STRNICMP` in the glue file (or on the compiler command line).

A notable limitation of MINIINI is that there is a (fixed) maximum length of a line that can be read from an INI file. This maximum length is configurable (at compile-time, not at run-time) and it may be short on embedded systems —see page 6.

When running in an Unicode environment or when moving the INI file across platforms, there may be other considerations concerning the use of MINIINI —see the relevant sections in this chapter, specifically the section on Unicode on page 8.

The glue file

The MINIINI library must be configured for a platform with the help of a so-called "glue file". This glue file contains macros (and possibly "inline" functions) that map file reading and writing functions used by the MINIINI library to those provided by the operating system. The glue file must be called "`minGlue.h`".

One general configuration is whether internal error checking via "assertions" is active. The MINIINI library uses the `assert` macro to help catch errors in the MINIINI library and/or catch errors in how the application interfaces with the MINIINI library. To build a release version, one typically recompiles all source code with the `NDEBUG` macro set.

In the case that your (embedded) platform lacks an `assert.h` file, you may want to define `NDEBUG` in the `minGlue.h` file.

I/O functions

The MININI source code requires functions from a file I/O library to perform the actual reading and writing. This can be any library; MININI does not rely on the availability of a standard C library, because embedded operating systems may have limited support for file I/O. Even on full operating systems, separating the file I/O from the INI format parsing carries advantages, because it allows you to cache the INI file and thereby enhance performance.

The functions that you need to implement, or map to standard file I/O functions are:

LISTING: Functions to map in the "glue file"

```
int ini_openread(const char *filename, INI_FILETYPE *file)

int ini_openwrite(const char *filename, INI_FILETYPE *file)

int ini_close(INI_FILETYPE *file)

int ini_read(char *buffer, size_t size, INI_FILETYPE *file)

int ini_write(char *buffer, INI_FILETYPE *file)

int ini_rename(const char *source, const char *dest)

int ini_remove(const char *filename)

int ini_tell(INI_FILETYPE *file, INI_FILEPOS *pos)

int ini_seek(INI_FILETYPE *file, INI_FILEPOS *pos)
```

All functions should return zero on failure and a non-zero value on success. For examples of "implementations" for the above functions, see appendix A on page 23.

The `ini_remove` function is redundant if `ini_rename` overwrites an existing destination file. When using GCC and Glibc as the C library, you may use the behaviour of the `rename` function (from Glibc) to overwrite the destination if it exists. Removing the original file before renaming the new file to the original name is then redundant, and the definition of `ini_remove` is then likewise redundant. When not using GCC and Glibc, you should consult the documentation for your compiler and library; standard C defines the behaviour of `rename` in this condition as implementation-defined.

The `INI_FILETYPE` type used in the above "glue" functions, must also be defined in the glue file. If you are using the standard C/C^{++} file I/O library, this is the "`FILE*`" type of the standard C/C^{++} file I/O library. On embedded systems with a different I/O library, chances are that you need a different handle or "structure" to identify the storage. For example:

```
#define INI_FILETYPE    HANDLE
```

The MINIINI functions will declare variables of the `INI_FILETYPE` type and pass these variables to sub-functions (including the glue interface functions) by reference.

For read-only support of INI files, only the macros/functions `ini_openread`, `ini_close` and `ini_read` are needed (see also the next page). The other functions are only needed for writing support. The type that holds the "file position" (for functions `ini_tell` and `ini_seek`) must be declared as well. For applications that use the standard C/C++ file I/O library functions `fgetpos` and `fsetpos`, this is the `fpos_t` type.

```
#define INI_FILEPOS     fpos_t
```

Function `ini_openread` is for opening an existing file, and for opening it for reading only. Function `ini_openwrite` must create a new file, or delete and re-create an existing file. The definition of the function `ini_openrewrite` is optional; if available, it is used to open an existing file for writing, but without truncating the file (many libraries call this "read + write mode"). Function `ini_openrewrite` allows for an optimization in the special case that an update of a setting does not cause the file length to be changed.

On Microsoft Windows and DOS, files can be opened in either "text mode" or in "binary mode", and this relates mostly on the line termination translation. Despite INI files being text files, it is advised to open the INI file in binary mode.

See see appendix A on page 23 for examples of glue files for various file systems.

Buffer size (maximum line length)

Another item that needs to be configured is the buffer size. The functions in the MINIINI library allocate this buffer on the stack, so the buffer size is directly related to the stack usage. In addition, the buffer size determines the maximum line length that is supported in the INI file and the maximum path name length for the temporary file (for writing support). For example, `minGlue.h` could contain the definition:

```
#define INI_BUFFERSIZE      512
```

The above macro limits the line length of the INI files supported by MINIINI to 512 characters.

The buffer size declared here is also the size of the "write cache" that MINIINI uses to optimize performance on file writes.

Read-only support

In its default configuration, MINIINI supports both reading and writing INI files. If your application does not require write support, you can add a setting to the `minGlue.h` file to strip out the unneeded code.

```
#define INI_READONLY
```

When writing a setting to an INI, MINIINI writes it to a temporary file, copies the other sections and keys from the original INI file, and then deletes the original file and renames the temporary file to the name of the original file. This approach uses the least amount of memory. The disadvantage is that writes to an INI file are slow, especially on large INI files.

Furthermore, when writing to the temporary file, MINIINI repeatedly looks ahead in the source INI and jumps back to a position that it marked earlier. The goal of this design is to minimize the number of inidividual "write actions" to the file, because on Flash memory (and EEPROM memory), writing is an order of magnitude slower than reading.

The path name of the temporary file is the same as the input file, but with the last character set to a tilde ("~").

Browsing support

An effecient way to scan through the complete INI file and read all settings, is with `ini_browse`, see page 10. Browsing support may be excluded from the MINIINI library by defining the `INI_NOBROWSE` definition.

Rational number support

MINIINI can be configured to support reading and writing single-precision floating point values —see the functions `ini_getf` and `ini_putf`. Embedded processors may lack floating point hardware and software emulation of floating-point operations may be too costly in resources (memory). For these platforms, alternatives are to switch to a fixed-point representation or, when rational numbers are not relevant for the project, to disable the rational number support in MINIINI altogether.

To enable rational number support, a macro for the type and macros or interface functions for number-to-text conversions must be added to `minGlue.h`. For the standard C/C++ library, you can add the following definitions to the glue file:

```
#define INI_REAL                float
#define ini_ftoa(string,value)  sprintf((string),"%f",(value))
#define ini_atof(string)        (INI_REAL)strtod((string),NULL)
```

For a different representation of rational numbers, only the definitions in `minGlue.h` have to change. The following example is based on the "`fixedptc`" library by Ivan Voras.

```
#define INI_REAL                fixedpt
#define ini_ftoa(string,value)  fixedpt_str((value),(string))
#define ini_atof(string)        fixedpt_val((string))
```

To disable rational number support, remove the declaration for the `INI_REAL` type from the `minGlue.h` file.

Unicode (enable/disable)

minIni can be compiled with Unicode support, but it delegates storing the actual characters to the "glue" routines. Although you can use standard Unicode file reading and writing routines to create and query ini files in Unicode text format, it is advised to keep the ini file format as ascii, for best compatibility with other implementations. To store Unicode characters in the ascii file, convert the Unicode data to (and from) UTF-8 (the minIni library does not provide functions for this conversion).

It is advised to keep the section and key names as ascii or ansi Latin-1; only the "values" of each key should be encoded as UTF-8.

Currently, all distributions of Linux lack a header file called `tchar.h` which adds a portability layer for source code that can be compiled as ascii or as Unicode. minIni relies on `tchar.h` when compiling for Unicode. Therefore, when compiling a Unicode application under Linux, you have two options: create a minimal version of `tchar.h` yourself, or compile minIni for the 8-bit ansi character set, while the remainder of the application is Unicode. To force-compile minIni for ansi, add the definition `INI_ANSIONLY` in the glue file ("`minGlue.h`"). For example:

```
#define INI_ANSIONLY   /* ignore UNICODE or _UNICODE macros, compile as ASCII/ANSI */
```

Line termination

On Microsoft Windows and DOS, lines of text files are usually terminated by a CR-LF character pair ("`\r\n`" in C/C++ terminology). On Linux and Unix (and macOS), the line terminator is only the LF character.

The line termination convention is not important when reading from ini files, because minIni strips off all trailing white space (and control characters such as carriage-return and line-feed are considered white space). The line termination convention is also not important when the ini file is only accessed by minIni. Finally, if you use the standard C/C++ library as the back-end for reading and writing files, this standard C/C++ library may already handle the platform-dependent line termination for you.

However, if you wish to read and adjust the ini files with other applications, across platforms —e.g. edit the ini file with a simple text editor as Notepad on Microsoft Windows and then store it on an embedded device with a Linux-based operating system, then it may be advantageous to tell minIni the line termination characters to use. To do so, define the macro `INI_LINETERM` in the file "`minGlue.h`" and set it to the character or characters to use. For example:

```
#define INI_LINETERM    "\r\n"
```

Summary of configuration macros

`INI_ANSIONLY` If this macro is defined, INI files are forced to be written with 8-bit characters (ASCII or ANSI character sets), regardless of whether the remainder of the application is written as Unicode. See this page.

`INI_BUFFERSIZE` The maximum line length that is supported, as well as the maximum path length for temporary file (if write access is enabled). The default value is 512. See page 6.

`INI_FILEPOS` The type for a position in a file. This is a *required setting* if writing support is enabled.

`INI_FILETYPE` The type for a variable that represents a file. This is a *required setting*. See page 5.

`INI_LINETERM` This macro should be set to the line termination character (or characters). If left undefined, the default is a line-feed character. Note that the standard file I/O library may translate a line-feed character to a carriage-return/line-feed pair (this depends on the file I/O library). See the previous page

`INI_NOBROWSE` Exclude the `ini_browse` function from the MININI library.

`INI_READONLY` If this macro is defined, write access is disabled (and the code for writing INI files is stripped from the MININI library. See page 6

`INI_REAL` The type for a variable that represents a rational number. If left undefined, rational number support is disabled. See page 7.

`NDEBUG` If defined, the `assert` macro in the MININI source code is disabled. Typically developers build with assertions enabled during development and disable them for a release version. If your platform lacks an `assert` macro, you may want to define the `NDEBUG` macro in the `minGlue.h` file.

`PORTABLE_STRNICMP` If defined, MININI uses an internal, portable `strnicmp` function. This is required for platforms that lack this function —note that MININI already handles the case where this function is called `strncasecmp`. See page 4.

Multi-tasking

The MININI library does not have any global variables and it does not use any dynamically allocated memory. Yet, the library should not be considered "thread-safe" or reentrant, because it implicitly uses a particular shared resource: the file system.

Multiple tasks reading from an INI file do not pose a problem. However, when one task is writing to an INI file, no other tasks should access this INI file —neither for reading,

nor for writing. It might be easier, in the implementation, to serialize *all* accesses of the INI file.

The first advise in protecting resources from concurrent access in a multi-tasking environment is to avoid sharing resources between tasks. If only a single task uses a resource, no semaphore protection is necessary and no priority inversion or deadlock problems can occur. This advise also applies to the MINIINI library. If possible, make a single task the "owner" of the INI file and create a client/server architecture for other tasks to query and adjust settings.

If access to the INI file must be shared between tasks (and at least one of the tasks writes to the INI file), you need to write wrappers around the functions of the MINIINI library that block on a mutex or binary semaphore, or that use the file locking mechanism in the operating system. See the next sections for tips specific to an operating systems.

Linux

An option in Linux (and other Unix-like environments) is to use an advisory lock on the calls to open the INI file. In the snippet below, the glue function `ini_openread` sets a "shared" file lock (allowing others to also open the file for reading), but `ini_openwrite` sets an "exclusive" lock on the file.

The lock can only be set after opening the file, which is why `ini_openwrite` first attempts to open an existing file, and creates a new file if no existing file can be opened. If it had started by creating a new file, the call to `fopen` with the mode `"w"` would have truncated the file *before* aquiring the lock —thereby possibly truncating a locked file. If an existing file was indeed opened, and the lock aquired, it must now be explicitly truncated.

The `flock` function is blocking by default, meaning that it does not proceed if it cannot aquire the lock. It thereby implicitly functions as a kind of semaphore.

LISTING: Glue function using file locking

```
static inline int ini_openread(const char *filename, INI_FILETYPE *file) {
    if ((*file = fopen((filename),"r")) == NULL)
        return 0;
    return flock(fileno(*file), LOCK_SH) == 0;
}
static inline int ini_openwrite(const char *filename, INI_FILETYPE *file) {
    if ((*file = fopen((filename),"r+")) == NULL
        && (*file = fopen((filename),"w")) == NULL)
        return 0;
    if (flock(fileno(*file), LOCK_EX) < 0)
        return 0;
    return ftruncate(fileno(*file), 0) == 0;
}
```

Note that no "unlock" request is needed: the file is implicitly unlocked when the file is closed.

Browsing through the file contents

The "browse" function `ini_browse` processes the complete INI file and invokes a callback function for every setting that it reads from the file.

An alternative to browsing through the INI file is by enumerating the sections and the keys, see the next section. Browsing is more efficient when the whole INI file must be processed, enumeration allows you to scan only through specific sections.

Key and section enumeration

MINIINI can list all sections in an INI file and all keys in a section, but in a different way than the function `GetProfileString` from the Microsoft Windows API. To list all sections, call function `ini_getsection` with an incremental "index" number until it fails. Similarly, to list all keys in a section, call `ini_getkey` with an incremental "index" number (plus the name of the section) until it fails.

LISTING: Browsing through all keys and all sections in "config.ini"

```
int s, k;
char section[40], key[40];
for (s = 0; ini_getsection(s, section, sizeof section, "config.ini") > 0; s++) {
    printf("[%s]\n", section);
    for (k = 0; ini_getkey(section, k, key, sizeof key, "config.ini") > 0; k++)
        printf("\t%s\n", key);
} /* for */
```

Function reference

In addition to the functions in plain C, minIni comes with a C++ class. When creating a variable of the `minIni` class, you pass in the name of the INI file once, so that this name does not need to be passed to every other function. The class exists for the standard C++ string library and for wxWidgets, using the `wxString` type. The function reference only lists the methods with the `std::string` type, but these are replaced by versions that use `wxString` when compiling for wxWidgets.

minIni::minIni — class constructor

The `minIni` constructor creates an instance of the `minIni` class.

C++:
```
minIni(const std::string& filename)
```

`filename` The full file name of the INI file to use for all reads and writes, through this instance. The filename format and specifications, and whether or not this parameter may include a path, depends on the underlying file I/O library.

Example: Creating a class instance to read a setting (C++ only):

```
minIni ini("config.ini");
std::string username = ini.gets("Users", "admin");
```

ini_browse / minIni::browse — Browse through all settings

`ini_browse` runs through the file and invokes a callback on every setting.

C:
```
int ini_browse(INI_CALLBACK Callback, void *UserData,
               const char *Filename)
```

C++:
```
bool browse(INI_CALLBACK Callback, void *UserData)
```

`Callback` The function that is invoked on every setting, see the notes below.

`UserData` A general-purpose application-defined value that is passed to the callback function.

`Filename` The name of the INI file. The filename format and specifications, and whether or not this parameter may include a path, depends on the underlying file I/O library.

The C++ class uses the filename specified in the class constructor.

Returns: 1/`true` on success, 0/`false` on failure.

Notes: The callback function is defined as:

```
int Callback(const char *Section, const char *Key,
             const char *Value, void *UserData);
```

The `Section` string may an empty string, for the settings that are outside any section. The `UserData` parameter is the same as what is passed to the `ini_browse` function.

The callback function should return a non-zero value on success, or zero to abort further browsing through the INI file.

The callback function should not write into the INI file.

See also: `ini_getkey`, `ini_getsection`

---

minIni::del

Delete a section or a key

Delete a key or an entire section.

C++:
```
bool del(const std::string& Section,
         const std::string& Key)
```

C++:
```
bool del(const std::string& Section)
```

`Section` The name of the section.

`Key` The name of the key.

Returns: `true` on success, `false` on failure.

Notes: This method is the equivalent of `ini_puts` with the parameter `Key` and/or `Value` parameters to *NULL*.

This function is unavailable if MINIINI is configured as a read-only library (page 6).

See also: `ini_puts`

---

ini_getbool / minIni::getbool

Read a "truth" flag

`ini_getbool` returns zero for false or one for true, depending on the value that is found in the given section and at the given key.

C:
```
int ini_getbool(const char *Section, const char *Key,
                int DefValue, const char *Filename)
```

C++: `bool getbool(const std::string& Section,`
`const std::string& Key, bool DefValue=false)`

`Section` The name of the section. If this parameter is *NULL* or an empty string, the `Key` is searched outside any section.

`Key` The name of the key. This parameter may not be *NULL*.

`DefValue` The default value, which will be returned if the key is not present in the INI file. Even though it is declared as an "`int`" in the C interface, it should be either 0 (zero) or 1 (one).

`Filename` The name of the INI file. The filename format and specifications, and whether or not this parameter may include a path, depends on the underlying file I/O library.

The C++ class uses the filename specified in the class constructor.

Returns: The true/false flag as interpreted from the value read at the given key, or `DefValue` if the key is not present in the given section (or if it cannot be interpreted to either a "true" or a "false" flag).

Specifically, the return value depends on the first letter of the value read at the key. If that first character is:
◇ "Y", "T" or "1", the function returns `true` (or 1);
◇ "N", "F" or "0", the function returns `false` (or 0);
◇ anything else, the function returns parameter `DefValue`.

The comparison is *not* case-sensitive.

See also: `ini_getl`, `ini_putbool`

---

ini_getf / minIni::getf — Read a rational number

`ini_getf` returns the numeric value that is found in the given section and at the given key. The value may have a fractional part (i.e. rational numbers).

C: `INI_REAL ini_getf(const char *Section, const char *Key,`
`INI_REAL DefValue, const char *Filename)`

C++: `INI_REAL getf(const std::string& Section,`
`const std::string& Key, INI_REAL DefValue=0)`

`Section` The name of the section. If this parameter is *NULL* or an empty string, the `Key` is searched outside any section.

`Key` The name of the key. This parameter may not be *NULL*.

`DefValue` The default value, which will be returned if the key is not present in the INI file.

`Filename` The name of the INI file. The filename format and specifications, and whether or not this parameter may include a path, depends on the underlying file I/O library.

The C++ class uses the filename specified in the class constructor.

Returns: The value read at the given key, or `DefValue` if the key is not present in the given section.

Notes: Rational number support must have been *enabled* to use the function —see page 7. The type for the rational numbers (the `INI_REAL` type), depends on the configuration of MININI.

See also: `ini_getl`, `ini_putf`

---

ini_getkey / minIni::getkey — Enumerate keys

Read the name of an indexed key inside a given section.

C:
```
int ini_getkey(const char *Section, int Index,
               char *Buffer, int BufferSize,
               const char *Filename)
```

C++:
```
str::string getkey(const std::string& Section, int Index)
```

`Section` The name of the section. If this parameter is *NULL* or an empty string, the keys outside any section are enumerated.

`Index` The zero-based index of the key to return.

`Buffer` The buffer into which this function will store the key name.

`BufferSize` The size of the buffer in parameter `Buffer`. This is the maximum number of characters that will be read and stored.

`Filename` The name of the INI file. The filename format and specifications, and whether or not this parameter may include a path, depends on the underlying file I/O library.

The C++ class uses the filename specified in the class constructor.

Returns: The C function returns the number of characters that were read, or zero if no (more) keys are present in the specified section. The C++ method returns the name of the key in a string.

Example: Enumerating keys in section “Devices”:

```
int k;
char name[20];
for (k = 0; ini_getkey("Devices", k, name, 20, "config.ini") > 0; k++)
    printf("%s\n", name);
```

See also: ini_browse, ini_getsection, ini_haskey

ini_getl / minIni::getl — Read a numeric value

`ini_getl` returns the integer value (a “whole number”) that is found in the given section and at the given key.

C:
```
long ini_getl(const char *Section, const char *Key,
              long DefValue, const char *Filename)
```

C++:
```
long getl(const std::string& Section,
          const std::string& Key, long DefValue=0)
```

C++:
```
int geti(const std::string& Section,
         const std::string& Key, int DefValue=0)
```

`Section` The name of the section. If this parameter is *NULL* or an empty string, the `Key` is searched outside any section.

`Key` The name of the key. This parameter may not be *NULL*.

`DefValue` The default value, which will be returned if the key is not present in the INI file.

`Filename` The name of the INI file. The filename format and specifications, and whether or not this parameter may include a path, depends on the underlying file I/O library.

The C++ class uses the filename specified in the class constructor.

Returns: The value read at the given key, or `DefValue` if the key is not present in the given section.

If the key is present, but it does not represent a decimal number, this function may return zero or an incorrect value.

Notes: The number must be in decimal or in hexadecimal format. For hexadecimal values, the value must be preceded with “`0x`”; for example, `0x1234` stands for the decimal value 4660.

See also: ini_getf, ini_gets, ini_putl

ini_gets / minIni::gets — Read a string

`ini_gets` reads the textual value that is found in the given section and at the given key.

C:
```
int ini_gets(const char *Section, const char *Key,
             const char *DefValue, char *Buffer,
             int BufferSize, const char *Filename)
```

C++:
```
std::string gets(const std::string& Section,
                 const std::string& Key,
                 const std::string& DefValue="")
```

`Section` The name of the section. If this parameter is *NULL* or an empty string, the `Key` is searched outside any section.

`Key` The name of the key. This parameter may not be *NULL*.

`DefValue` The default value, which will be returned (in parameter `Buffer`) if the key is not present in the INI file.

`Buffer` The buffer into which this function will store the data read.

`BufferSize` The size of the buffer in parameter `Buffer`. This is the maximum number of characters that will be read and stored.

`Filename` The name of the INI file. The filename format and specifications, and whether or not this parameter may include a path, depends on the underlying file I/O library.

The C++ class uses the filename specified in the class constructor.

Returns: The C function returns the number of characters that were read. The C++ method returns the string read at the given key, or `DefValue` if the key is not present in the given section.

See also: `ini_getl`, `ini_puts`

ini_getsection / minIni::getsection — Enumerate sections

`ini_getsection` reads the name of an indexed section.

C:
```
int ini_getsection(int Index, char *Buffer, int BufferSize,
                   const char *Filename)
```

C++:
```
std::string getsection(int Index)
```

`Index` The zero-based index of the section to return.

`Buffer` The buffer into which this function will store the section name.

`BufferSize` The size of the buffer in parameter `Buffer`. This is the maximum number of characters that will be read and stored.

`Filename` The name of the INI file. The filename format and specifications, and whether or not this parameter may include a path, depends on the underlying file I/O library.

The C++ class uses the filename specified in the class constructor.

Returns: The C function returns the number of characters that were read, or zero if no (more) sections are present in the INI file. The C++ method returns the name of the section in a string.

Example: Enumerating all sections in file "config.ini":

```
int s;
char name[20];
for (s = 0; ini_getsection(s, name, 20, "config.ini") > 0; s++)
    printf("%s\n", name);
```

See also: `ini_browse`, `ini_getkey`, `ini_hassection`

ini_haskey / minIni::haskey — Check whether a key exists

`ini_haskey` checks whether a key exists in a section(without returning its value).

C:
```
int ini_haskey(const char *Section, const char *Key,
               const char *Filename)
```

C++:
```
bool haskey(const std::string& Section,
            const std::string& Key)
```

`Section` The name of the section. If this parameter is *NULL* or an empty string, the `Key` is searched outside any section.

`Key` The name of the key. This parameter may not be *NULL*.

`Filename` The name of the INI file. The filename format and specifications, and whether or not this parameter may include a path, depends on the underlying file I/O library.

The C++ class uses the filename specified in the class constructor.

Returns: 1/`true` if the key is present, 0/`false` otherwise.

See also: `ini_getkey`, `ini_hassection`

ini_hassection / minIni::hassection — Check whether a section exists

`ini_hassection` checks whether a key exists in a section(without returning its value).

C:
```
int ini_hassection(const char *Section,
                   const char *Filename)
```

C++:
```
bool hassection(const std::string& Section)
```

`Section` The name of the section. This parameter may not be *NULL*.

`Filename` The name of the INI file. The filename format and specifications, and whether or not this parameter may include a path, depends on the underlying file I/O library.

The C++ class uses the filename specified in the class constructor.

Returns: 1/`true` if the section is present, 0/`false` otherwise.

See also: `ini_getsection`, `ini_haskey`

ini_putbool / minIni::put — Store a true/false flag

`ini_putbool` stores the value as a true/false flag. Any non-zero value is stored as "true", a zero value is stored as "false".

C:
```
int ini_putbool(const char *Section, const char *Key,
                int Value, const char *Filename)
```

C++:
```
bool put(const std::string& Section,
         const std::string& Key, bool Value)
```

`Section` The name of the section. If this parameter is *NULL* or an empty string, the `Key` is stored outside any section (i.e. above the first section, if the INI file has any sections).

`Key` The name of the key. This parameter may not be *NULL*.

`Value` The value to write at the key and the section.

`Filename` The name of the INI file. The filename format and specifications, and whether or not this parameter may include a path, depends on the underlying file I/O library.

The C++ class uses the filename specified in the class constructor.

Returns: 1/`true` on success, 0/`false` on failure.

Notes: This function is unavailable if MINIINI is configured as a read-only library (page 6).

See also: `ini_getbool`, `ini_putl`

---

ini_putf / minIni::put — Store a rational number

`ini_putf` stores the numeric value that in the given section and at the given key. The numeric value is written as a rational number, with a "whole part" and a fractional part.

C:
```
int ini_putf(const char *Section, const char *Key,
             INI_REAL Value, const char *Filename)
```

C++:
```
bool put(const std::string& Section,
         const std::string& Key, INI_REAL Value)
```

`Section` The name of the section. If this parameter is *NULL* or an empty string, the `Key` is stored outside any section (i.e. above the first section, if the INI file has any sections).

`Key` The name of the key. This parameter may not be *NULL*.

`Value` The value to write at the key and the section.

`Filename` The name of the INI file. The filename format and specifications, and whether or not this parameter may include a path, depends on the underlying file I/O library.

The C++ class uses the filename specified in the class constructor.

Returns: 1/`true` on success, 0/`false` on failure.

Notes: This function is unavailable if MINIINI is configured as a read-only library (page 6). It is also unavailable if rational number support has *not* been enabled (page 7).

The type for the rational numbers, `INI_REAL`, depends on the configuration of MINIINI.

See also: `ini_getf`, `ini_putl`

ini_putl / minIni::put

Store a numeric value

`ini_putl` stores the numeric value that in the given section and at the given key.

C:
```
int ini_putl(const char *Section, const char *Key,
             long Value, const char *Filename)
```

C++:
```
bool put(const std::string& Section,
         const std::string& Key, long Value)
```

`Section` The name of the section. If this parameter is *NULL* or an empty string, the `Key` is stored outside any section (i.e. above the first section, if the INI file has any sections).

`Key` The name of the key. This parameter may not be *NULL*.

`Value` The value to write at the key and the section.

`Filename` The name of the INI file. The filename format and specifications, and whether or not this parameter may include a path, depends on the underlying file I/O library.

The C++ class uses the filename specified in the class constructor.

Returns: 1/`true` on success, 0/`false` on failure.

Notes: This function is unavailable if MINIINI is configured as a read-only library (page 6).

See also: `ini_getl`, `ini_puts`

ini_puts / minIni::put

Store a string

`ini_puts` stores the text parameter that in the given section and at the given key.

C:
```
int ini_puts(const char *Section, const char *Key,
             const char *Value, const char *Filename)
```

C++:
```
bool put(const std::string& Section,
         const std::string& Key,
         const std::string& Value)
```

C++:
```
bool put(const std::string& Section,
         const std::string& Key, const char* Value)
```

`Section` The name of the section. If this parameter is *NULL* or an empty string, the `Key` is stored outside any section (i.e. above the first section, if the INI file has any sections).

`Key` The name of the key. If this parameter is *NULL*, the function erases all keys (and their associated values) from the section.

`Value` The text to write at the key and the section. This string should not contain carriage-return or line-feed characters.
If this parameter is *NULL*, the function erases the key/value pair.

`Filename` The name of the INI file. The filename format and specifications, and whether or not this parameter may include a path, depends on the underlying file I/O library.

The C++ class uses the filename specified in the class constructor.

Returns: 1/`true` on success, 0/`false` on failure.

Notes: This function can also be used to delete entries or sections, by setting the `Key` or `Value` parameters to *NULL*.

This function is unavailable if MINIINI is configured as a read-only library (page 6).

See also: `ini_gets`, `ini_putl`

APPENDIX A

Example glue files

stdio (standard C/C++ library)

On Microsoft Windows or DOS, it is advised to open the INI file in binary mode, despite INI files being *text* files. If text mode is unavailable on yor platform, change `"rb"` and `"wb"` to `"r"` and `"w"` respectively.

```
/* map required file I/O types and functions to the standard C library */
#include <stdio.h>

#define INI_FILETYPE                    FILE*
#define ini_openread(filename,file)     ((*(file) = fopen((filename),"rb")) != NULL)
#define ini_openwrite(filename,file)    ((*(file) = fopen((filename),"wb")) != NULL)
#define ini_openrewrite(filename,file)  ((*(file) = fopen((filename),"r+b")) != NULL)
#define ini_close(file)                 (fclose(*(file)) == 0)
#define ini_read(buffer,size,file)      (fgets((buffer),(size),*(file)) != NULL)
#define ini_write(buffer,file)          (fputs((buffer),*(file)) >= 0)
#define ini_rename(source,dest)         (rename((source), (dest)) == 0)
#define ini_remove(filename)            (remove(filename) == 0)

#define INI_FILEPOS                     long int
#define ini_tell(file,pos)              (*(pos) = ftell(*(file)))
#define ini_seek(file,pos)              (fseek(*(file), *(pos), SEEK_SET) == 0)

/* for floating-point support, define additional types and functions */
#define INI_REAL                        float
#define ini_ftoa(string,value)          sprintf((string),"%f",(value))
#define ini_atof(string)                (INI_REAL)strtod((string),NULL)
```

CCS FAT library (http://www.ccsinfo.com)

```
#define INI_BUFFERSIZE  256       /* maximum line length, maximum path length */

#ifndef FAT_PIC_C
  #error FAT library must be included before this module
#endif
#define const                     /* keyword not supported by CCS */

#define INI_FILETYPE                    FILE
#define ini_openread(filename,file)     (fatopen((filename), "r", (file)) == GOODEC)
#define ini_openwrite(filename,file)    (fatopen((filename), "w", (file)) == GOODEC)
#define ini_close(file)                 (fatclose((file)) == 0)
#define ini_read(buffer,size,file)      (fatgets((buffer), (size), (file)) != NULL)
#define ini_write(buffer,file)          (fatputs((buffer), (file)) == GOODEC)
#define ini_remove(filename)            (rm_file((filename)) == 0)

#define INI_FILEPOS                     fatpos_t
#define ini_tell(file,pos)              (fatgetpos((file), (pos)) == 0)
#define ini_seek(file,pos)              (fatsetpos((file), (pos)) == 0)

#ifndef INI_READONLY
/* CCS FAT library lacks a rename function, so instead we copy the file to the
 * new name and delete the old file
```

```
 */
static int ini_rename(char *source, char *dest)
{
  FILE fr, fw;
  int n;

  if (fatopen(source, "r", &fr) != GOODEC)
    return 0;
  if (rm_file(dest) != 0)
    return 0;
  if (fatopen(dest, "w", &fw) != GOODEC)
    return 0;

  /* With some "insider knowledge", we can save some memory: the "source"
   * parameter holds a filename that was built from the "dest" parameter. It
   * was built in a local buffer with the size INI_BUFFERSIZE. We can reuse
   * this buffer for copying the file.
   */
  while (n=fatread(source, 1, INI_BUFFERSIZE, &fr))
    fatwrite(source, 1, n, &fw);

  fatclose(&fr);
  fatclose(&fw);

  /* Now we need to delete the source file. However, we have garbled the buffer
   * that held the filename of the source. So we need to build it again.
   */
  ini_tempname(source, dest, INI_BUFFERSIZE);
  return rm_file(source) == 0;
}
#endif
```

EFSL (http://www.efsl.be/)

```
#define INI_BUFFERSIZE  256       /* maximum line length, maximum path length */
#define INI_LINETERM    "\r\n"    /* set line termination explicitly */

#include "efs.h"
extern EmbeddedFileSystem g_efs;

#define INI_FILETYPE                    EmbeddedFile
#define ini_openread(filename,file)     (file_fopen((file), &g_efs.myFs, \
                                                    (char*)(filename), 'r') == 0)
#define ini_openwrite(filename,file)    (file_fopen((file), &g_efs.myFs, \
                                                    (char*)(filename), 'w') == 0)
#define ini_close(file)                 file_fclose(file)
#define ini_read(buffer,size,file)      (file_read((file), (size), (buffer)) > 0)
#define ini_write(buffer,file)          (file_write((file), strlen(buffer), \
                                                    (char*)(buffer)) > 0)
#define ini_remove(filename)            rmfile(&g_efs.myFs, (char*)(filename))

#define INI_FILEPOS                     euint32
#define ini_tell(file,pos)              (*(pos) = (file)->FilePtr))
#define ini_seek(file,pos)              file_setpos((file), (*pos))

#if ! defined INI_READONLY
/* EFSL lacks a rename function, so instead we copy the file to the new name
 * and delete the old file
 */
static int ini_rename(char *source, const char *dest)
```

```
{
  EmbeddedFile fr, fw;
  int n;

  if (file_fopen(&fr, &g_efs.myFs, source, 'r') != 0)
    return 0;
  if (rmfile(&g_efs.myFs, (char*)dest) != 0)
    return 0;
  if (file_fopen(&fw, &g_efs.myFs, (char*)dest, 'w') != 0)
    return 0;

  /* With some "insider knowledge", we can save some memory: the "source"
   * parameter holds a filename that was built from the "dest" parameter. It
   * was built in buffer and this buffer has the size INI_BUFFERSIZE. We can
   * reuse this buffer for copying the file.
   */
  while (n=file_read(&fr, INI_BUFFERSIZE, source))
    file_write(&fw, n, source);

  file_fclose(&fr);
  file_fclose(&fw);

  /* Now we need to delete the source file. However, we have garbled the buffer
   * that held the filename of the source. So we need to build it again.
   */
  ini_tempname(source, dest, INI_BUFFERSIZE);
  return rmfile(&g_efs.myFs, source) == 0;
}
#endif
```

FAT Filing System (http://www.embedded-code.com/)

```
#define INI_BUFFERSIZE  256       /* maximum line length, maximum path length */
#include <mem-ffs.h>

#define INI_FILETYPE                  FFS_FILE*
#define ini_openread(filename,file)   ((*(file) = ffs_fopen((filename),"r")) != NULL)
#define ini_openwrite(filename,file)  ((*(file) = ffs_fopen((filename),"w")) != NULL)
#define ini_close(file)               (ffs_fclose(*(file)) == 0)
#define ini_read(buffer,size,file)    (ffs_fgets((buffer),(size),*(file)) != NULL)
#define ini_write(buffer,file)        (ffs_fputs((buffer),*(file)) >= 0)
#define ini_rename(source,dest)       (ffs_rename((source), (dest)) == 0)
#define ini_remove(filename)          (ffs_remove(filename) == 0)

#define INI_FILEPOS                   long
#define ini_tell(file,pos)            (ffs_fgetpos(*(file), (pos)) == 0)
#define ini_seek(file,pos)            (ffs_fsetpos(*(file), (pos)) == 0)
```

FatFs (http://elm-chan.org/)

```
#define INI_BUFFERSIZE  256       /* maximum line length, maximum path length */

/* You must set FF_USE_STRFUNC to 1 or 2 in the include file ff.h (or tff.h)
 * to enable the "string functions" fgets() and fputs().
 */
#include "ff.h"                   /* include tff.h for Tiny-FatFs */

/* When setting FF_USE_STRFUNC to 2 (for LF to CR/LF translation), INI_LINETERM
 * should be defined to "\n" (otherwise "\r\n" will be translated by FatFS to
 * "\r\r\n").
*/
#if defined FF_USE_STRFUNC && FF_USE_STRFUNC == 2 && !defined INI_LINETERM
  #define INI_LINETERM  "\n"
#endif

#define INI_FILETYPE    FIL
#define ini_openread(filename,file)   (f_open((file), (filename), \
                                      FA_READ+FA_OPEN_EXISTING) == FR_OK)
#define ini_openwrite(filename,file)  (f_open((file), (filename), \
                                      FA_WRITE+FA_CREATE_ALWAYS) == FR_OK)
#define ini_close(file)               (f_close(file) == FR_OK)
#define ini_read(buffer,size,file)    f_gets((buffer), (size),(file))
#define ini_write(buffer,file)        f_puts((buffer), (file))
#define ini_remove(filename)          (f_unlink(filename) == FR_OK)

#define INI_FILEPOS                   DWORD
#define ini_tell(file,pos)            (*(pos) = f_tell((file)))
#define ini_seek(file,pos)            (f_lseek((file), *(pos)) == FR_OK)

static int ini_rename(TCHAR *source, const TCHAR *dest)
{
  /* Function f_rename() does not allow drive letters in the destination file */
  char *drive = strchr(dest, ':');
  drive = (drive == NULL) ? dest : drive + 1;
  return (f_rename(source, drive) == FR_OK);
}
```

“Memory Disk Drive” file system (Microchip)

```
#define INI_BUFFERSIZE  256       /* maximum line length, maximum path length */

#include "MDD File System\fsio.h"
#include <string.h>

#define INI_FILETYPE                  FSFILE*
#define ini_openread(filename,file)   ((*(file) = FSfopen((filename),FS_READ)) != NULL)
#define ini_openwrite(filename,file)  ((*(file) = FSfopen((filename),FS_WRITE)) != NULL)
#define ini_openrewrite(filename,file) ((*(file) = fopen((filename),FS_READPLUS)) != NULL)
#define ini_close(file)               (FSfclose(*(file)) == 0)
#define ini_write(buffer,file)        (FSfwrite((buffer), 1, strlen(buffer), (*file)) > 0)
#define ini_remove(filename)          (FSremove((filename)) == 0)

#define INI_FILEPOS                   long int
#define ini_tell(file,pos)            (*(pos) = FSftell(*(file)))
#define ini_seek(file,pos)            (FSfseek(*(file), *(pos), SEEK_SET) == 0)

/* Since the Memory Disk Drive file system library reads only blocks of files,
 * the function to read a text line does so by "over-reading" a block of the
```

```
 * of the maximum size and truncating it behind the end-of-line.
 */
static int ini_read(char *buffer, int size, INI_FILETYPE *file)
{
  size_t numread = size;
  char *eol;

  if ((numread = FSfread(buffer, 1, size, *file)) == 0)
    return 0;                   /* at EOF */
  if ((eol = strchr(buffer, '\n')) == NULL)
    eol = strchr(buffer, '\r');
  if (eol != NULL) {
    /* terminate the buffer */
    *++eol = '\0';
    /* "unread" the data that was read too much */
    FSfseek(*file, - (int)(numread - (size_t)(eol - buffer)), SEEK_CUR);
  } /* if */
  return 1;
}

#ifndef INI_READONLY
static int ini_rename(const char *source, const char *dest)
{
  FSFILE* ftmp = FSfopen((source), FS_READ);
  FSrename((dest), ftmp);
  return FSfclose(ftmp) == 0;
}
#endif
```

APPENDIX B

License

The software product "MINIINI" is copyright © 2008–2020 by CompuPhase. MINIINI is distributed under the "Apache License" version 2.0, which is reproduced below, plus an exception clause to explicitly allow static linking.

EXCEPTION TO THE APACHE 2.0 LICENSE

As a special exception to the Apache License 2.0 (and referring to the definitions in Section 1 of this license), you may link, statically or dynamically, the "Work" to other modules to produce an executable file containing portions of the "Work", and distribute that executable file in "Object" form under the terms of your choice, without any of the additional requirements listed in Section 4 of the Apache License 2.0. This exception applies only to redistributions in "Object" form (not "Source" form) and only if no modifications have been made to the "Work".

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. **Definitions**

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work.

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. **Grant of Copyright License.** Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sub-license, and distribute the Work and such Derivative Works in Source or Object form.
3. **Grant of Patent License.** Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counter-claim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.
4. **Redistribution.** You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:
   a. You must give any other recipients of the Work or Derivative Works a copy of this License; and
   b. You must cause any modified files to carry prominent notices stating that You changed the files; and
   c. You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
   d. If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

   You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.
5. **Submission of Contributions.** Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.
6. **Trademarks.** This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.
7. **Disclaimer of Warranty.** Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.
8. **Limitation of Liability.** In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work

stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. **Accepting Warranty or Additional Liability.** While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

Index

◇ Names of persons or companies (not products) are in *italics*.
◇ Function names, constants and compiler reserved words are in `typewriter` font.