libtmk

— Terminal Manipulation Kit – Documentation – Release 17.0.x —

https://github.com/skippyr/libtmk

C Programming Language

Author(s):

Sherman Rofeman ⟨skippyr.developer@icloud.com⟩

For the memory of all fellow dragons seeing this,
in the help to keep your flames alive.

Here Be Dragons!
Made With Love.

CONTENTS

CHAPTER 1

OVERVIEW

1.1 About

Libtmk (*aka* "Terminal Manipulation Kit") is a simple C99 terminal manipulation library that contains a modest set of features to manipulate terminal properties, styles, arguments and key readings primarily using UTF-8 encoding. It is available for Windows, macOS and Linux.

With its minimalistic and opinionated design, it encourages the development of software with great system compatibility, while leaving room to full-feature libraries to be built upon for more specific use cases.

1.2 Help

If you need any help related to this project, create a new issue in its issues page (`https://github.com/skippyr/libtmk/issues`) or send an e-mail (`mailto:skippyr.developer@icloud.com`) describing what is going on.

1.3 Contributing

This project is open to review and possibly accept contributions in the form of bug reports and suggestions. If you are interested, send your contribution to its pull requests page (`https://github.com/skippyr/libtmk/pulls`) or via e-mail (`mailto:skippyr.developer@icloud.com`).

1.4 License

CHAPTER 2

INSTALLATION

2.1 Dependencies

Due to its cross-platform nature, it is highly recommended to use this library with a setup that combines CMake and Git. For that, the following dependencies are required, varying depending of the operating system you are using:

- **Windows:**
  1. **Visual Studio 2022** or later **with the "Desktop Development with C++ Workload":** it contains most of the tools required to develop and build C projects, including a compiler and CMake.
  2. **Git:** it will be used to clone and maintain a version of the library's source code within your project.

  Tip: you can install these packages using Winget, the official Windows package manager, or Chocolatey, an alternative package manager.
- **macOS:**
  1. **Apple Command Line Tools:** it contains most of the tools required to build C projects, including a compiler and Git.
  2. **CMake:** it will be used to integrate and build the library within your project.

  Tip: you can install the Apple Command Line Tools using the command `xcode select -install`. For the rest, use HomeBrew (`https://brew.sh`).
- **Linux:**
  1. **GCC** or **Clang:** it will be used to compile the library.
  2. **CMake** and **Make**: it will be used to integrate and build the library within your project.
  3. **Git:** it will be used to clone and maintain a version of the library's source code within your project.

> **Tip:** you can install these packages using your distro's package manager. If it does not have one, use HomeBrew (`https://brew.sh`).

2.2 Procedures

On Windows, using the `Developer PowerShell for VS 2022` profile or, on any other operating systems, using any terminal, follow these instructions:

- In the root directory of your project, use Git to clone the library's repository as a submodule. This will download its latest version and allow you to update it easily by pulling new changes in the future:

```
git submodule add https://github.com/skippyr/libtmk libs/libtmk
```

Code 2.1: a command to clone the library's repository as a submodule.

- In your `CMakeLists.txt` configuration file, add the library's directory you have cloned as a subdirectory. That will make the library target `tmk` available for you to link it to your executable targets. Use the following example as a reference, adapting it to your project. It links the library to the executable `src/main.c`, named as target `main` in the project `tmk-example`:

```
cmake_minimum_required(VERSION 3.25)
set(CMAKE_C_STANDARD_REQUIRED ON)
set(CMAKE_C_STANDARD 99)
add_subdirectory("${CMAKE_SOURCE_DIR}/libs/libtmk"
                 "${CMAKE_BINARY_DIR}/libtmk")
project(tmk-example)
add_executable(main "${CMAKE_SOURCE_DIR}/src/main.c")
target_link_libraries(main tmk)
```

Code 2.2: a `CMakeLists.txt` example that links the library to an executable.

- Delete and recreate the CMake auto-generated build files and, at the next time you build your project, the library will be automatically linked to your executable.

CHAPTER 3

SPECIFICATION

3.1 Standard

The library uses features that require the C99 standard or later to be targetted. As shown during the installation, with CMake, that can be configured by setting the `CMAKE_C_STANDARD` and `CMAKE_C_STANDARD_REQUIRED` global variables.

3.2 Platform

Due to the constrain of supporting UTF-8 encoding, it requires to be run in computer systems that are x32 or x64 bits. This condition is always met for Windows and macOS, but, as Linux is very versatile, this turn the library less portable, depending on where you need to use it.

3.3 Headers

The library code is prototyped within the header file `tmk.h`. When included, as depends on some standard headers, the following ones will be forwarded and becoming available to your software:

```
1 #include <stdarg.h>
2 #include <stddef.h>
```

Code 3.1: the headers forwarded by the library.

3.4 Naming

The library reserves all of its artifacts by prefixing their names with `tmk_` (for public members) and `_tmk_` (for private members).

3.5 Limitations

The library has some limitations over its usage. To avoid conflicts and malfunctions, you must obey to these rules:

- It uses byte oriented functions from the standard library for UTF-8 encoding support. All standard streams must be kept unoriented or under that same orientation.
- It does not fully reset terminal properties. Avoid manipulating them manually using functions from the other libraries.
- It is not prepared for multithreading. Always use only one thread to manipulate the terminal and avoid resources racing.

CHAPTER 4

USAGE

4.1 History

The terminal was a hardware device that allowed people to interact with mainframes with the use of a keyboard, screen and text based applications. As computers have become smaller and portable, vendors started to offer a software to emulate its features called the "virtual terminal", "terminal emulator" or simply "terminal".

Though graphical related software have become popular among desktop users, the terminal still relevant specially considering that all software running on a computer is always connected to a terminal related layer.

The terminal can be manipulated using operating system APIs or, if it supports, with the use of ANSI escape sequences, a serie of standardized codes. Along this chapter, you will learn about some terminal components and how to use the library functions to interact with them.

4.2 Standard Streams

The terminal is always composed by three standard data streams that serve as essential communication channels between the operating system and the softwares running within it. Those are declared as enumerators within the `tmk_Std` enum:

- `tmk_StdIn` (*aka* "standard input", "stdin" or "stream 0"): an input stream that is connected to a buffer that receives bytes describing events or text readings.
- `tmk_StdOut` (*aka* "standard output", "stdout" or "stream 1"): an output stream where regular tense messages are intended to be output to. It is connected to a buffer, which holds bytes to be written until it suffers a "flush", which can happen naturally or manually.
- `tmk_StdErr` (*aka* "standard error", "stderr" or "stream 2"): an output stream where error tense messages are intended to be output to. It does not have a buffer.

4.2.1 Buffers

The input buffer can be cleared and the output buffer can be manually flushed using the `tmk_clearin` and `tmk_flushout` functions, respectively, ensuring precise control over data processing.

```
1  void tmk_clearin(void);
2  void tmk_flushout(void);
```

Code 4.1: the declarations of the `tmk_clearin` and `tmk_flushout` functions.

4.2.2 Writing & Clearing

You can write to the output and error streams using the `tmk_write` and `tmk_ewrite` functions, respectively. If you need to use a variadic list of arguments, prefer the versions prefixed with "v", `tmk_vwrite` and `tmk_vewrite`, as they accept a `va_list` parameter.

All of these writing functions are wrappers to the `printf` functions family from the standard library, making them behave alike, even accepting the same format specifiers. They differ by ensuring the use of UTF-8 encoding on Windows, and, when writing to the error stream, they flush the output buffer to sync possible cached ANSI escape sequences.

Using them, you can write data and draw text user interfaces, but to avoid clashing up with existing contents in a line, the current one can always be cleared using the `tmk_clearln` function.

```
1  #include <tmk.h>
2
3  #define DRAGON "\xf0\x9f\x90\x89"
4
5  int main(void) {
6    tmk_write("Hello, world!");
7    tmk_clearln();
8    tmk_write("Here Be Dragons! %s\n", DRAGON);
9    return 0;
10 }
```

Code 4.2: an example using the writing functions.

In the example above, note that the first greeting written will be replaced by the second one as the line gets cleared. As UTF-8 encoding is ensured, the dragon emoji will also be formatted and correctly displayed in all operating systems.

```
1  void tmk_vwrite(const char *fmt, va_list v);
2  void tmk_write(const char *fmt, ...);
3  void tmk_vewrite(const char *fmt, va_list v);
4  void tmk_ewrite(const char *fmt, ...);
5  void tmk_clearln(void);
```

Code 4.3: the declarations of the writing and clearing functions.

4.2.3 Key Events

For compatibility reasons, the library only allows the reading of key events from the input buffer, ignoring and not even enabling other types of events. To read an event, use the `tmk_readkey` function, giving it:

- As the first parameter, a time, in milliseconds, to wait for an event: a negative value makes it wait forever; zero makes it return immediately; otherwise, a positive value, makes it wait the given time.
- As the second parameter, the address of a custom filter function that receives the address of a temporary `tmk_key` struct read and that by its return of a boolean decides whether to keep or discard that event. If not required, set it to `NULL`.
- As the third parameter, the address of a `tmk_key` struct that will hold the result.

It returns the reading status, which can be:

- 0: if the reading succeded, parsing a key.
- −1: if the reading failed due to stream redirection or an error of kernel resources allocation.
- −2: if a wait time of zero was given and no input was available.
- −3: if a positive wait time was given and the timer ran out.
- −4: if the terminal window was resized, interrupting the reading.

```c
#include <tmk.h>

#ifdef __APPLE__
#define ALT "Option"
#else
#define ALT "Alt"
#endif
#define DRAGON "\xf0\x9f\x90\x89"

int main(void) {
  tmk_write("Use the following keys:\n");
  tmk_write("    - Up Arrow\n");
  tmk_write("    - A\n");
  tmk_write("    - Shift + A\n");
  tmk_write("    - %s + A\n", ALT);
  tmk_write("    - Dragon Emoji %s (copy and paste it)\n", DRAGON);
  tmk_write("Press [Esc] to exit.\n\n");
  struct tmk_key k;
  do {
    tmk_clearin();
    tmk_readkey(-1, NULL, &k);
    if (k.buf == tmk_KeyUpArr) {
      tmk_write("You used: Up Arrow.\n");
    } else if (k.buf == 'a' && !k.mods)  {
      tmk_write("You used: A.\n");
    } else if (k.buf == 'A') {
      tmk_write("You used: Shift + A.\n");
    } else if (k.buf == 'a' && k.mods & tmk_ModAlt &&
```

```
29                !(k.mods & tmk_ModCtrl)) {
30        tmk_write("You used: %s + A.\n", ALT);
31      } else if (k.buf == *(int*)DRAGON) {
32        tmk_write("You used: Dragon Emoji %s.\n", DRAGON);
33      }
34    } while (k.buf != tmk_KeyEsc);
35    return 0;
36  }
```

Code 4.4: an example that parse some keys.

In the example above, in a while true loop, keys are read and a range of predefined keys is identified, including an arrow key, key sequences with and without modifiers and an UTF-8 grapheme: an emoji. The loop is terminated if the Escape key is used.

The function allows the reading of any UTF-8 grapheme but only a limited set of keyboard keys, not including navigation (Insert, Delete, Home, End, PageUp and PageDown) and function keys (F1 – F12). It always flush the output buffer upon execution. On macOS and Linux, the terminal may not provide enough information to distinguish certain key sequences, causing them to possibly be incorrectly identified as the library has to assume one of the possible values, that happens for:

- `Shift + i` as it gets interpreted as `tmk_KeyTab`.
- `Shift + j` as it gets interpreted as `tmk_KeyRet`.
- `Option + E` as it gets interpreted as `Option + e`.
- `Option + M` as it gets interpreted as `Option + m`.
- `Option + R` as it gets interpreted as `Option + r`.

The possibility of being interrupted by window resizes allows you to catch and treat that condition in order to create software whose text user interface is responsive.

```
1  int tmk_readkey(int wait, int (*filt)(struct tmk_key*), struct tmk_key *key);
```

Code 4.5: the declaration of the `tmk_readkey` function.

The `tmk_key` struct represents the information read of a key, and it is composed of the following fields:

- `buf` (`int`): a buffer of 4 bytes that may store an UTF-8 grapheme or an enumerator from the `tmk_Key` enum, representing the key read.
- `mods` (`int`): a bitmask that may be composed by enumerators from the `tmk_Mod` enum, containing the modifier keys used.

The Alt (used interchangeably as the Option key on macOS) and Ctrl modifier keys can be identified by the library in letters (a – z and A – Z), Tab, Space, Backspace and Return keys by the set of flags in the `mods` bitmask. The Shift key, in contrast, may only be identified by checking the grapheme received in `buf`, as it usually modifies its regular result: for example, when used in letters, they become uppercase.

The `tmk_Mod` enum contains the available modifier keys as its enumerators, each one being a bitmask flag:

- `tmk_ModCtrl`: the Ctrl modifier key.
- `tmk_ModAlt`: the Alt modifier key.

The `tmk_Key` enum is composed by enumerators that represent keyboard keys not associated with UTF-8 graphemes, and some that are but whose values within can be used to improve readability:

- `tmk_KeyUpArr`: the up arrow key.
- `tmk_KeyDnArr`: the down arrow key.
- `tmk_KeyRgArr`: the right arrow key.
- `tmk_KeyLfArr`: the left arrow key.
- `tmk_KeyTab`: the Tab key.
- `tmk_KeyRet`: the Return key.
- `tmk_KeyEsc`: the Escape key.
- `tmk_KeySpc`: the Space key.
- `tmk_KeyBckspc`: the Backspace key.

4.2.4 Redirection

When a stream is connected to a terminal and its respective buffer, if it has one, it is considered being a "teletypewriter" (*aka* "tty"). That condition may change due to shell redirection operations. You can check the state of a stream using the `tmk_istty` function with an enumerator from the `tmk_Str` enum, returning a boolean as the result.

```c
#include <tmk.h>

#define BOOL(val_a) (val_a ? "true" : "false")

int main(void) {
  tmk_write("TTY Statuses\n");
  tmk_write("In: %s\n", BOOL(tmk_istty(tmk_StdIn)));
  tmk_write("Out: %s\n", BOOL(tmk_istty(tmk_StdOut)));
  tmk_write("Err: %s\n", BOOL(tmk_istty(tmk_StdErr)));
  return 0;
}
```

Code 4.6: an example that checks if each stream is a tty.

In the example above, the tty status of each stream is written out. Try redirecting those streams using your shell and see how the output changes accordingly. Performing these types of checks can help you treat stream redirection issues right at the initialization of your programs.

```c
int tmk_istty(int std);
```

Code 4.7: the declaration of the `tmk_istty` function.

4.3 Encoding Convertion

The Windows terminal API has a particular trait: it evolved from using ASCII encoding to UTF-16 with surrogate pairs. As the library uses UTF-8, it contains, exclusively on Windows, the function `tmk_asutf8` to convert between those two last encodings. The

convertion result, once not required anymore, must be freed using the `free` function to avoid memory leaks.

```
#ifndef _WIN32
#error "This example is exclusive for Windows."
#endif
#include <tmk.h>

int main(void) {
  char *str = tmk_asutf8(L"Here Be Dragons!");
  tmk_write("%s\n", str);
  free(str);
  return 0;
}
```

Code 4.8: an example that converts an UTF-16 encoded string to UTF-8.

In the example above, which can only be compiled on Windows, a large string, which on that operating system is encoded in UTF-16, is converted to UTF-8, written and then its buffer gets freed.

```
#ifdef _WIN32
char *tmk_asutf8(const wchar_t *wstr);
#endif
```

Code 4.9: the declaration of the `tmk_asutf8` function.

4.4 Window

The terminal, while its streams are not being redirected, is either connected to a window, in case a window manager is in use, or the whole computer screen, otherwise. The window is a grid system that can hold characters being displayed: composed by columns, horizontally, and rows, vertically.

An origin position, where both column and row are zero, is considered at the top left corner of the window. From there, those positions increase going right and down respectively.

4.4.1 Dimensions

You can get the dimensions of the window using the `tmk_getwdim` function, giving the address of a `tmk_dim` struct that will hold the result. In case of success, it returns zero, otherwise, a non-zero value, which can happen in case of stream redirection.

```
#include <tmk.h>

int main(void) {
  struct tmk_dim d;
  tmk_getwdim(&d);
  tmk_write("Total Columns: %hu\n", d.cols);
  tmk_write("Total Rows: %hu\n", d.rows);
  return 0;
}
```

Code 4.10: an example that gets the terminal window dimensions.

In the example above, the dimensions of the terminal window are retrieved and then each of its components are written.

```
int tmk_getwdim(struct tmk_dim *d);
```

Code 4.11: the declaration of the `tmk_getwdim` function.

The `tmk_dim` struct represents dimensions within the terminal window, being composed of the following fields:

- `cols` (`unsigned short`): the total number of columns of the dimensions.
- `rows` (`unsigned short`): the total number of rows of the dimensions.

4.4.2 Alternate Buffer

Terminals often provide an alternate buffer, a separate window environment where software can execute and reverts to its original content upon completion. You can set its use by giving a boolean to the `tmk_setwalt` function.

```
#include <tmk.h>

int main(void) {
  tmk_setwalt(1);
  tmk_write("Here Be Dragons!\n");
  tmk_write("Press [Return] to exit.\n");
  struct tmk_key k;
  do {
    tmk_readkey(-1, NULL, &k);
  } while (k.buf != tmk_KeyRet);
  tmk_setwalt(0);
  return 0;
}
```

Code 4.12: an example that shows an usage for the alternate buffer.

In the example above, the alternate buffer is used to write a message and return back to its original contents once the Return key is pressed. In order to be seen, a key reading is required, otherwise, the alternate buffer would simply close without showing anything.

```
int tmk_setwalt(int isalt);
```

Code 4.13: the declaration of the `tmk_setwalt` function.

4.5 Font

The font used by the terminal defines the symbols it supports and is usually customizable through its configurations, however its style can also be set by software that runs within the terminal.

4.5.1 Colors

The colors of the font can be set using the function `tmk_setclr`, giving it:

- As the first parameter, an enumerator from the `tmk_Clr` enum representing the color to be set. To revert it, set the `tmk_ClrDft` value.
- As the second parameter, an enumerator from the `tmk_Lyr` enum representing the layer to be affected.

The foreground and background colors can also be swapped by giving a boolean to the `tmk_swpclr` function, creating a nice high contrast effect that can be used to indicate selection or highlighting.

```c
#include <tmk.h>

int main(void) {
  const char *msg = "Here Be Dragons!\n";
  tmk_setclr(tmk_ClrRed, tmk_LyrFg);
  tmk_write(msg);
  tmk_setclr(tmk_ClrDft, tmk_LyrFg);
  tmk_setclr(tmk_ClrRed, tmk_LyrBg);
  tmk_write(msg);
  tmk_setclr(tmk_ClrDft, tmk_LyrBg);
  tmk_swpclr(1);
  tmk_write(msg);
  tmk_swpclr(0);
  tmk_write(msg);
  return 0;
}
```

Code 4.14: an example that applies and swaps the colors of a message written.

In the example above, the color red gets applied to the foreground and background layers of a message. They are, then, swapped, creating a negative effect, and reverted back to its original colors.

```c
void tmk_setclr(int clr, int lyr);
void tmk_swpclr(int isswp);
```

Code 4.15: the declarations of the `tmk_setclr` and `tmk_swpclr` functions.

The `tmk_Clr` contains all the colors from the terminal colorscheme as its enumerators:

- `tmk_ClrDft`: the default color, used for resets.
- `tmk_ClrRed`: the red color.
- `tmk_ClrGrn`: the green color.
- `tmk_ClrYlw`: the yellow color.
- `tmk_ClrBle`: the blue color.
- `tmk_ClrMag`: the magenta color.
- `tmk_ClrCyn`: the cyan color.
- `tmk_ClrWht`: the white color.
- `tmk_ClrGry`: the gray color.

The `tmk_Lyr` contains enumerators that represents the terminal font layers:

- `tmk_LyrFg`: the foreground layer, used to refer to the graphemes itself.
- `tmk_LyrBg`: the background layer, used to refer to the background the graphemes lay on.

4.5.2 Weights

The font can have its weight set using the `tmk_setwgt` function by giving it an enumerator value from the `tmk_Wgt` enum, representing the weight. To revert it, use the `tmk_WgtDft` value. This feature is also called "brightness" or "color intensity".

```
1  #include <tmk.h>
2
3  int main(void) {
4    const char *msg = "Here Be Dragons!\n";
5    tmk_setwgt(tmk_WgtBld);
6    tmk_write(msg);
7    tmk_setwgt(tmk_WgtDim);
8    tmk_write(msg);
9    tmk_setwgt(tmk_WgtDft);
10   tmk_write(msg);
11   return 0;
12 }
```

Code 4.16: an example that sets font weights.

In the example above, the different font weights are applied to a message written and then reverted back to its original one.

```
1  void tmk_setwgt(int wgt);
```

Code 4.17: the declaration of the `tmk_setwgt` function.

The `tmk_Wgt` enum is composed by the available font weights as its enumerators:

- `tmk_WgtDft`: the default weight, used for resets.
- `tmk_WgtBld`: the bold weight, usually rendered with bold weight and/or light colors.
- `tmk_WgtDim`: the dim weight, usually rendered with faint colors.

4.6 Cursor

The cursor is a specific position in the terminal window that is used as reference when writing data to it from its output streams. Usually, it has a visible appearance in the form of a shape, where the user can see where it is typing at.

4.6.1 Visibility

The visibility of the cursor can be set by giving a boolean to the `tmk_setcvis` function.

```c
#include <tmk.h>

int main(void) {
  struct tmk_key k;
  tmk_write("Cursor is now invisible.\n");
  tmk_write("Press [Return] to exit.\n");
  tmk_setcvis(0);
  do {
    tmk_readkey(-1, NULL, &k);
  } while (k.buf != tmk_KeyRet);
  tmk_setcvis(1);
  return 0;
}
```

Code 4.18: an example that sets the cursor visibility.

In the example above, the cursor is made hidden and then its visibility is revert back. Reading a key is necessary so the changes can be seen between the function calls.

```c
void tmk_setcvis(int isvis);
```

Code 4.19: the declaration of the `tmk_setcvis` function.

4.6.2 Shapes

The shape of the cursor can be set by giving an enumerator from the `tmk_Shp` enum, representing the shape, to the `tmk_setcshp` function. To revert it, use the `tmk_ShpDft` value.

```c
#include <tmk.h>

static int filt(struct tmk_key *k) {
  return k->buf == tmk_KeyRet;
}

int main(void) {
  struct tmk_key k;
  tmk_write("Use [Return] to advance between the cursor shapes.\n\n");
  tmk_setcshp(tmk_ShpUnd);
  tmk_write("[1/3] Underline Shape ");
  tmk_readkey(-1, filt, &k);
  tmk_setcshp(tmk_ShpBlk);
  tmk_write("\n[2/3] Block Shape ");
  tmk_readkey(-1, filt, &k);
  tmk_setcshp(tmk_ShpBar);
  tmk_write("\n[3/3] Bar Shape ");
  tmk_readkey(-1, filt, &k);
  tmk_setcshp(tmk_ShpDft);
  tmk_write("\n");
  return 0;
}
```

Code 4.20: an example that sets cursor shapes.

In the example above, the different cursor shapes are applied and then reverted back to its original one. Reading a key is necessary so the changes can be seen between the function calls.

```
1  void tmk_setcshp(int shp);
```

Code 4.21: the declaration of the `tmk_setcshp` function.

The `tmk_Shp` contains the available cursor shapes as its enumerators:

- `tmk_ShpDft`: the default shape, used for resets.
- `tmk_ShpBUnd`: the blinking variant of the underline shape.
- `tmk_ShpUnd`: the non-blinking variant of the underline shape.
- `tmk_ShpBBlk`: the blinking variant of the block shape.
- `tmk_ShpBlk`: the non-blinking variant of the block shape.
- `tmk_ShpBBar`: the blinking variant of the bar shape.
- `tmk_ShpBar`: the non-blinking variant of the bar shape.

4.6.3 Position

The position of the cursor in the window can be get by giving the address of a `tmk_pos` struct to the `tmk_getcpos` function. On macOS and Linux, as it needs to read a terminal answer given through the input buffer, it clears it upon execution. You can set its position by giving a `tmk_pos` struct to the `tmk_setcpos` function.

As alternative to having a precise position, you can also move it a number of steps in a specific direction, represented by the enumerators of the `tmk_Drt` enum, giving those parameters to the `tmk_mvcpos` function.

All positions given are always restrained within the window, so exceeding them will just place the cursor at its limits, never overflowing or causing errors.

```
1  #include <tmk.h>
2
3  int main(void) {
4    struct tmk_key k;
5    struct tmk_pos p;
6    tmk_setwalt(1);
7    tmk_setcpos((struct tmk_pos){2, 1});
8    tmk_getcpos(&p);
9    tmk_write("Cursor Position\n");
10   tmk_mvcpos(4, tmk_DrtRg);
11   tmk_write("Column: %hu\n", p.col);
12   tmk_mvcpos(4, tmk_DrtRg);
13   tmk_write("Row: %hu\n", p.row);
14   tmk_mvcpos(1, tmk_DrtDn);
15   tmk_mvcpos(2, tmk_DrtRg);
16   tmk_write("Press the [Return] key to exit");
17   do {
18     tmk_readkey(-1, NULL, &k);
19   } while (k.buf != tmk_KeyRet);
20   tmk_setwalt(0);
21   return 0;
```

```
22 }
```

Code 4.22: an example that interacts with the cursor position in different ways.

In the example above, the cursor is moved within the alternate window buffer using those different positioning functions in order to write its current position after the first move. Reading a key is necessary so the changes can be seen after the function calls.

```
1 int tmk_getcpos(struct tmk_pos *p);
2 void tmk_setcpos(struct tmk_pos p);
3 void tmk_mvcpos(unsigned short stp, int drt);
```

Code 4.23: the declarations of the cursor positioning function.

The `tmk_pos` struct is composed of the following fields:

- `col` (`unsigned short`): the column component of the position.
- `row` (`unsigned short`): the row component of the position.

The `tmk_Drt` enum contains the directions the cursor can move to as its enumerators:

- `tmk_DrtUp`: the up direction.
- `tmk_DrtDn`: the down direction.
- `tmk_DrtRg`: the right direction.
- `tmk_DrtLf`: the left direction.

4.7 Bell

Terminals usually have a bell feature to call for the user attention. To ring the bell, use the `tmk_ringbell` function. Unless disabled, that may emit a sound, show a bell symbol and/or flash the window.

```
1 void tmk_ringbell(void);
```

Code 4.24: the declaration of the `tmk_ringbell` function.

4.8 Arguments

The terminal receives arguments given through the shell and those arguments are encoded differently dependending on the operating system in use. As a cross-platform solution, the library provides the `tmk_getargs` function that you can use by giving it the first two arguments you received through the `main` function and the address of a `tmk_args` struct to hold the result. Once not required anymore, the arguments received must be freed using the `tmk_freeargs` function to avoid memory leaks.

The arguments returned by the function are both encoded in UTF-8 and UTF-16 (this one, exclusively on Windows) and do not include the binary path as per usual in C.

```
1 #include <tmk.h>
2
3 int main(int argc, const char **argv) {
4   struct tmk_args args;
```

```
5   tmk_getargs(argc, argv, &args);
6   tmk_write("Arguments:\n");
7   for (int i = 0; i < args.total; ++i) {
8     tmk_write("  [%d] %s\n", i, args.asutf8[i]);
9   }
10  tmk_write("Total: %d\n", args.total);
11  tmk_freeargs(&args);
12  return 0;
13 }
```

Code 4.25: an example that gets the command line arguments.

In the example above, the command line arguments are received and then written out with their total on the bottom.

```
1 void tmk_getargs(int argc, const char **argv, struct tmk_args *a);
2 void tmk_freeargs(struct tmk_args *a);
```

Code 4.26: the declarations of the arguments function.

The `tmk_args` struct is composed by the following fields:

- `total` (`int`): the total number of arguments.
- `asutf8` (`const char **`): the arguments encoded in UTF-8 encoding.
- `asutf16` (`const wchar_t **`): exclusive for Windows, the arguments encoded in UTF-16 encoding.