Vis Editor Documentation

Release 0.8

Marc André Tanner

Sep 23, 2023

CONTENTS

1 Vis **1**
- 1.1 Lifecycle . . . 1
- 1.2 Draw . . . 2
- 1.3 Windows . . . 3
- 1.4 Input . . . 4
- 1.5 Key Map . . . 4
- 1.6 Key Binding . . . 5
- 1.7 Key Action . . . 5
- 1.8 Modes . . . 6
- 1.9 Count . . . 7
- 1.10 Operators . . . 8
- 1.11 Motions . . . 9
- 1.12 Text Objects . . . 9
- 1.13 Marks . . . 10
- 1.14 Registers . . . 11
- 1.15 Macros . . . 11
- 1.16 Commands . . . 12
- 1.17 Options . . . 12
- 1.18 Modification . . . 14
- 1.19 Interaction . . . 15
- 1.20 Miscellaneous . . . 15

2 Text **17**
- 2.1 Load . . . 17
- 2.2 State . . . 18
- 2.3 Modify . . . 19
- 2.4 Access . . . 19
- 2.5 Iterator . . . 20
- 2.6 Lines . . . 22
- 2.7 History . . . 22
- 2.8 Marks . . . 23
- 2.9 Save . . . 24
- 2.10 Miscellaneous . . . 25

3 View **27**
- 3.1 Lifecycle . . . 27
- 3.2 Viewport . . . 27
- 3.3 Dimension . . . 28
- 3.4 Draw . . . 28
- 3.5 Selections . . . 28

3.6 Style 34

4 Buffer **35**

5 Array **39**

6 Map **43**

Index **45**

CHAPTER

ONE

VIS

The core Vis API.

1.1 Lifecycle

Vis ***vis_new**(Ui*, VisEvent*)

Create a new editor instance using the given user interface and event handlers.

void **vis_free**(Vis*)

Free all resources associated with this editor instance, terminates UI.

int **vis_run**(Vis*)

Enter main loop, start processing user input.

Returns
The editor exit status code.

void **vis_exit**(Vis*, int status)

Terminate editing session, the given `status` will be the return value of `vis_run`.

void vis_die (Vis *, const char *msg,...) __attribute__((noreturn

Emergency exit, print given message, perform minimal UI cleanup and exit process.

Note: This function does not return.

void format (printf, 2, 3)))

void **vis_suspend**(Vis*)

Temporarily suspend the editor process.

Note: This function will generate a `SIGTSTP` signal.

void **vis_resume**(Vis*)

Resume editor process.

Note: This function is usually called in response to a `SIGCONT` signal.

void **vis_doupdates**(Vis*, bool)

Set doupdate flag.

Note: Prevent flickering in curses by delaying window updates.

---

bool **vis_signal_handler**(Vis*, int signum, const siginfo_t *siginfo, const void *context)

Inform the editor core that a signal occurred.

Returns

Whether the signal was handled.

Note: Being designed as a library the editor core does *not* register any signal handlers on its own.

Note: The remaining arguments match the prototype of `sa_sigaction` as specified in *sigaction(2)*.

---

void **vis_interrupt**(Vis*)

Interrupt long running operation.

Note: It is invoked from *vis_signal_handler* when receiving `SIGINT`.

Warning: There is no guarantee that a long running operation is actually interrupted. It is analogous to cooperative multitasking where the operation has to voluntarily yield control.

bool **vis_interrupt_requested**(Vis*)

Check whether interruption was requested.

1.2 Draw

void **vis_draw**(Vis*)

Draw user interface.

void **vis_redraw**(Vis*)

Completely redraw user interface.

void **vis_update**(Vis*)

Blit user interface state to output device.

1.3 Windows

bool **vis_window_new**(Vis*, const char *filename)

Create a new window and load the given file.

Parameters

filename – If `NULL` a unamed, empty buffer is created.

Note: If the given file name is already opened in another window, the underlying File object is shared.

---

bool **vis_window_new_fd**(Vis*, int fd)

Create a new window associated with a file descriptor.

Note: No data is read from *fd*, but write commands without an explicit filename will instead write to the file descriptor.

---

bool **vis_window_reload**(Win*)

Reload the file currently displayed in the window from disk.

bool **vis_window_closable**(Win*)

Check whether closing the window would loose unsaved changes.

void **vis_window_close**(Win*)

Close window, redraw user interface.

bool **vis_window_split**(Win*)

Split the window, shares the underlying file object.

void **vis_window_status**(Win*, const char *status)

Change status message of this window.

void **vis_window_draw**(Win*)

void **vis_window_invalidate**(Win*)

void **vis_window_next**(Vis*)

Focus next window.

void **vis_window_prev**(Vis*)

Focus previous window.

void **vis_window_focus**(Win*)

Change currently focused window, receiving user input.

void **vis_window_swap**(Win*, Win*)

Swap location of two windows.

int **vis_window_width_get**(const Win*)

Query window width.

int **vis_window_height_get**(const Win*)

Query window height.

1.4 Input

The editor core processes input through a sequences of symbolic keys:

- Special keys such as <Enter>, <Tab> or <Backspace> as reported by termkey_strfkey.

  Note: The prefixes C-, S- and M- are used to denote the Ctrl, Shift and Alt modifiers, respectively.

- Key action names as registered with *vis_action_register*.

  Note: By convention they are prefixed with vis- as in <vis-nop>.

- Regular UTF-8 encoded input.

Note: An exhaustive list of the first two types is displayed in the :help output.

const char ***vis_keys_next**(Vis*, const char *keys)

Advance to the start of the next symbolic key.

Given the start of a symbolic key, returns a pointer to the start of the one immediately following it.

long **vis_keys_codepoint**(Vis*, const char *keys)

Convert next symbolic key to an Unicode code point, returns -1 for unknown keys.

bool **vis_keys_utf8**(Vis*, const char *keys, char utf8[static UTFmax+1])

Convert next symbolic key to a UTF-8 sequence.

Returns

Whether conversion was successful, if not utf8 is left unmodified.

Note: Guarantees that utf8 is NUL terminated on success.

void **vis_keys_feed**(Vis*, const char *keys)

Process symbolic keys as if they were user originated input.

1.5 Key Map

The key map is used to translate keys in non-input modes, *before* any key bindings are evaluated. It is intended to facilitate usage of non-latin keyboard layouts.

bool **vis_keymap_add**(Vis*, const char *key, const char *mapping)

Add a key translation.

void **vis_keymap_disable**(Vis*)

Temporarily disable the keymap for the next key press.

1.6 Key Binding

Each mode has a set of key bindings. A key binding maps a key to either another key (referred to as an alias) or a key action (implementing an editor operation).

If a key sequence is ambiguous (i.e. it is a prefix of multiple mappings) more input is awaited, until a unique mapping can be resolved.

Warning: Key aliases are always evaluated recursively.

KeyBinding ***vis_binding_new**(Vis*)

void **vis_binding_free**(Vis*, KeyBinding*)

bool **vis_mode_map**(Vis*, enum *VisMode*, bool force, const char *key, const KeyBinding*)

Set up a key binding.

Parameters

- **force** – Whether an existing mapping should be discarded.
- **key** – The symbolic key to map.
- **binding** – The binding to map.

Note: `binding->key` is always ignored in favor of `key`.

bool **vis_window_mode_map**(Win*, enum *VisMode*, bool force, const char *key, const KeyBinding*)

Analogous to `vis_mode_map`, but window specific.

bool **vis_mode_unmap**(Vis*, enum *VisMode*, const char *key)

Unmap a symbolic key in a given mode.

bool **vis_window_mode_unmap**(Win*, enum *VisMode*, const char *key)

Analogous to `vis_mode_unmap`, but window specific.

1.7 Key Action

A key action is invoked by a key binding and implements a certain editor function.

The editor operates like a finite state machine with key sequences as transition labels. Once a prefix of the input queue uniquely refers to a key action, it is invoked with the remainder of the input queue passed as argument.

Note: A triggered key action currently does not know through which key binding it was invoked. TODO: change that?

typedef const char ***KeyActionFunction**(Vis*, const char *keys, const Arg*)

Key action handling function.

Param keys

Input queue content *after* the binding which invoked this function.

Note: An empty string "" indicates that no further input is available.

Return
Pointer to first non-consumed key.

Warning: Must be in range `[keys, keys+strlen(keys)]` or `NULL` to indicate that not enough input was available. In the latter case the function will be called again once more input has been received.

KeyAction ***vis_action_new**(Vis*, const char *name, const char *help, *KeyActionFunction**, Arg)

Create new key action.

Parameters

- **name** – The name to be used as symbolic key when registering.
- **help** – Optional single line help text.
- **func** – The function implementing the key action logic.
- **arg** – Argument passed to function.

void **vis_action_free**(Vis*, KeyAction*)

bool **vis_action_register**(Vis*, const KeyAction*)

Register key action.

Note: Makes the key action available under the pseudo key name specified in `keyaction->name`.

1.8 Modes

A mode defines *enter*, *leave* and *idle* actions and captures a set of key bindings.

Modes are hierarchical, key bindings are searched recursively towards the top of the hierarchy stopping at the first match.

enum **VisMode**

Mode specifiers.

Values:

enumerator **VIS_MODE_NORMAL**

enumerator **VIS_MODE_OPERATOR_PENDING**

enumerator **VIS_MODE_VISUAL**

enumerator **VIS_MODE_VISUAL_LINE**

Sub mode of `VIS_MODE_VISUAL`.

enumerator **VIS_MODE_INSERT**

enumerator **VIS_MODE_REPLACE**

Sub mode of `VIS_MODE_INSERT`.

enumerator **VIS_MODE_INVALID**

void **vis_mode_switch**(Vis*, enum *VisMode*)

Switch mode.

Note: Will first trigger the leave event of the currently active mode, followed by an enter event of the new mode. No events are emitted, if the specified mode is already active.

enum *VisMode* **vis_mode_get**(Vis*)

Get currently active mode.

enum *VisMode* **vis_mode_from**(Vis*, const char *name)

Translate human readable mode name to constant.

1.9 Count

Dictates how many times a motion or text object is evaluated. If none is specified, a minimal count of 1 is assumed.

int **vis_count_get**(Vis*)

Get count, might return `VIS_COUNT_UNKNOWN`.

int **vis_count_get_default**(Vis*, int def)

Get count, if none was specified, return `def`.

void **vis_count_set**(Vis*, int count)

Set a count.

void **vis_tabwidth_set**(Vis*, int tw)

Set the tabwidth.

void **vis_shell_set**(Vis*, const char *new_shell)

Set the shell.

VisCountIterator **vis_count_iterator_get**(Vis*, int def)

Get iterator initialized with current count or `def` if not specified.

VisCountIterator **vis_count_iterator_init**(Vis*, int count)

Get iterator initialized with a count value.

bool **vis_count_iterator_next**(*VisCountIterator**)

Increment iterator counter.

Returns

Whether iteration should continue.

Note: Terminates iteration if the editor was interrupted in the meantime.

VIS_COUNT_UNKNOWN

No count was specified.

struct **VisCountIterator**

#include <vis.h>

1.10 Operators

size_t() VisOperatorFunction (Vis *, Text *, OperatorContext *)

An operator performs a certain function on a given text range.

Note: The operator must return the new cursor position or EPOS if the cursor should be disposed.

Note: The last used operator can be repeated using *vis_repeat*.

int **vis_operator_register**(Vis*, VisOperatorFunction*, void *context)

Register an operator.

Returns

Operator ID. Negative values indicate an error, positive ones can be used with vis_operator.

bool **vis_operator**(Vis*, enum VisOperator, ...)

Set operator to execute.

Has immediate effect if:

- A visual mode is active.
- The same operator was already set (range will be the current line).

Otherwise the operator will be executed on the range determined by:

- A motion (see vis_motion).
- A text object (vis_textobject).

The expected varying arguments are:

- VIS_OP_JOIN a char pointer referring to the text to insert between lines.
- VIS_OP_MODESWITCH an enum VisMode indicating the mode to switch to.
- VIS_OP_REPLACE a char pointer referring to the replacement character.

void **vis_repeat**(Vis*)

Repeat last operator, possibly with a new count if one was provided in the meantime.

void **vis_cancel**(Vis*)

Cancel pending operator, reset count, motion, text object, register etc.

1.11 Motions

enum **VisMotionType**

Values:

enumerator **VIS_MOTIONTYPE_LINEWISE**

enumerator **VIS_MOTIONTYPE_CHARWISE**

size_t() VisMotionFunction (Vis *, Win *, void *context, size_t pos)

Motions take a starting position and transform it to an end position.

Note: Should a motion not be possible, the original position must be returned. TODO: we might want to change that to `EPOS`?

bool **vis_motion**(Vis*, enum VisMotion, ...)

Set motion to perform.

The following motions take an additional argument:

- `VIS_MOVE_SEARCH_FORWARD` and `VIS_MOVE_SEARCH_BACKWARD`

  The search pattern as `const char *`.

- `VIS_MOVE_{LEFT,RIGHT}_{TO,TILL}`

  The character to search for as `const char *`.

void **vis_motion_type**(Vis *vis, enum *VisMotionType*)

Force currently specified motion to behave in line or character wise mode.

int **vis_motion_register**(Vis*, void *context, VisMotionFunction*)

Register a motion function.

Returns
Motion ID. Negative values indicate an error, positive ones can be used with `vis_motion`.

1.12 Text Objects

Filerange() VisTextObjectFunction (Vis *, Win *, void *context, size_t pos)

Text objects take a starting position and return a text range.

Note: The originating position does not necessarily have to be contained in the resulting range.

int **vis_textobject_register**(Vis*, int type, void *data, VisTextObjectFunction*)

Register a new text object.

Returns
Text object ID. Negative values indicate an error, positive ones can be used with `vis_textobject`.

bool **vis_textobject**(Vis*, enum VisTextObject)
Set text object to use.

1.13 Marks

Marks keep track of a given text position.

Note: Marks are currently file local.

enum VisMark **vis_mark_from**(Vis*, char mark)
Translate between single character mark name and corresponding constant.

char **vis_mark_to**(Vis*, enum VisMark)

void **vis_mark**(Vis*, enum VisMark)
Specify mark to use.

Note: If none is specified *VIS_MARK_DEFAULT* will be used.

enum VisMark **vis_mark_used**(Vis*)

void **vis_mark_set**(Win*, enum VisMark id, *Array* *sel)
Store a set of `Fileranges` in a mark.

Parameters
- **id** – The register to use.
- **sel** – The array containing the file ranges.

Array **vis_mark_get**(Win*, enum VisMark id)
Get an array of file ranges stored in the mark.

Warning: The caller must eventually free the Array by calling `array_release`.

void **vis_mark_normalize**(*Array**)
Normalize an *Array* of Fileranges.

Removes invalid ranges, merges overlapping ones and sorts according to the start position.

bool **vis_jumplist_save**(Vis*)
Add selections of focused window to jump list.

bool **vis_jumplist_prev**(Vis*)
Navigate jump list backwards.

bool **vis_jumplist_next**(Vis*)
Navigate jump list forwards.

1.14 Registers

enum VisRegister **vis_register_from**(Vis*, char reg)

Translate between single character register name and corresponding constant.

char **vis_register_to**(Vis*, enum VisRegister)

void **vis_register**(Vis*, enum VisRegister)

Specify register to use.

Note: If none is specified *VIS_REG_DEFAULT* will be used.

---

enum VisRegister **vis_register_used**(Vis*)

Array **vis_register_get**(Vis*, enum VisRegister)

Get register content.

Returns

An array of `TextString` structs.

> **Warning:** The caller must eventually free the array resources using `array_release`.

bool **vis_register_set**(Vis*, enum VisRegister, *Array* *data)

Set register content.

Parameters

data – The array comprised of `TextString` structs.

1.15 Macros

Macros are a sequence of keys stored in a Register which can be reprocessed as if entered by the user.

> **Warning:** Macro support is currently half-baked. If you do something stupid (e.g. use mutually recursive macros), you will likely encounter stack overflows.

bool **vis_macro_record**(Vis*, enum VisRegister)

Start recording a macro.

Note: Fails if a recording is already ongoing.

---

bool **vis_macro_record_stop**(Vis*)

Stop recording, fails if there is nothing to stop.

bool **vis_macro_recording**(Vis*)

Check whether a recording is currently ongoing.

bool **vis_macro_replay**(Vis*, enum VisRegister)

Replay a macro.

Note: A macro currently being recorded can not be replayed.

1.16 Commands

bool() VisCommandFunction (Vis *, Win *, void *data, bool force, const char *argv[], Selection *, Filerange *)

Command handler function.

bool **vis_cmd**(Vis*, const char *cmd)

Execute a `:`-command.

bool **vis_cmd_register**(Vis*, const char *name, const char *help, void *context, VisCommandFunction*)

Register new `:`-command.

Parameters

- **name** – The command name.
- **help** – Optional single line help text.
- **context** – User supplied context pointer passed to the handler function.
- **func** – The function implementing the command logic.

Note: Any unique prefix of the command name will invoke the command.

bool **vis_cmd_unregister**(Vis*, const char *name)

Unregister `:`-command.

1.17 Options

enum **VisOption**

Option properties.

Values:

enumerator **VIS_OPTION_TYPE_BOOL**

enumerator **VIS_OPTION_TYPE_STRING**

enumerator **VIS_OPTION_TYPE_NUMBER**

enumerator **VIS_OPTION_VALUE_OPTIONAL**

enumerator **VIS_OPTION_NEED_WINDOW**

enumerator **VIS_OPTION_DEPRECATED**

bool() VisOptionFunction (Vis *, Win *, void *context, bool toggle, enum VisOption, const char *name, Arg *value)

Option handler function.

Param win
The window to which option should apply, might be NULL.

Param context
User provided context pointer as given to vis_option_register.

Param force
Whether the option was specified with a bang !.

Param name
Name of option which was set.

Param arg
The new option value.

bool **vis_option_register**(Vis*, const char *names[], enum *VisOption*, VisOptionFunction*, void *context, const char *help)

Register a new :set option.

Parameters

- **names** – A NULL terminated array of option names.
- **option** – Option properties.
- **func** – The function handling the option.
- **context** – User supplied context pointer passed to the handler function.
- **help** – Optional single line help text.

Note: Fails if any of the given option names is already registered.

bool **vis_option_unregister**(Vis*, const char *name)

Unregister an existing :set option.

Note: Also unregisters all aliases as given to *vis_option_register*.

bool **vis_prompt_cmd**(Vis*, const char *cmd)

Execute any kind (:, ?, /) of prompt command.

int **vis_pipe**(Vis*, File*, Filerange*, const char *argv[], void *stdout_context, ssize_t (*read_stdout)(void *stdout_context, char *data, size_t len), void *stderr_context, ssize_t (*read_stderr)(void *stderr_context, char *data, size_t len), bool fullscreen)

Pipe a given file range to an external process.

If the range is invalid 'interactive' mode is enabled, meaning that stdin and stderr are passed through the underlying command, stdout points to vis' stderr.

If `argv` contains only one non-NULL element the command is executed through an intermediate shell (using `/bin/sh -c argv[0]`) that is argument expansion is performed by the shell. Otherwise the argument list will be passed unmodified to `execvp(argv[0], argv)`.

If the `read_stdout` and `read_stderr` callbacks are non-NULL they will be invoked when output from the forked process is available.

If `fullscreen` is set to `true` the external process is assumed to be a fullscreen program (e.g. curses based) and the ui context is restored accordingly.

> **Warning:** The editor core is blocked until this function returns.

Returns
The exit status of the forked process.

int **vis_pipe_collect**(Vis*, File*, Filerange*, const char *argv[], char **out, char **err, bool fullscreen)

Pipe a Filerange to an external process, return its exit status and capture everything that is written to stdout/stderr.

Parameters

- **argv** – Argument list, must be `NULL` terminated.
- **out** – Data written to `stdout`, will be `NUL` terminated.
- **err** – Data written to `stderr`, will be `NUL` terminated.
- **fullscreen** – Whether the external process is a fullscreen program (e.g. curses based)

> **Warning:** The pointers stored in `out` and `err` need to be *free(3)*-ed by the caller.

1.18 Modification

These function operate on the currently focused window but ensure that all windows which show the affected region are redrawn too.

void **vis_insert**(Vis*, size_t pos, const char *data, size_t len)

void **vis_delete**(Vis*, size_t pos, size_t len)

void **vis_replace**(Vis*, size_t pos, const char *data, size_t len)

void **vis_insert_key**(Vis*, const char *data, size_t len)

Perform insertion at all cursor positions.

void **vis_replace_key**(Vis*, const char *data, size_t len)

Perform character substitution at all cursor positions.

> **Note:** Does not replace new line characters.

void **vis_insert_tab**(Vis*)

Insert a tab at all cursor positions.

Note: Performs tab expansion according to current settings.

void **vis_insert_nl**(Vis*)

Inserts a new line character at every cursor position.

Note: Performs auto indentation according to current settings.

1.19 Interaction

void **vis_prompt_show**(Vis*, const char *title)

Display a user prompt with a certain title.

Note: The prompt is currently implemented as a single line height window.

void vis_info_show (Vis *, const char *msg,...) __attribute__((format(printf

Display a single line message.

Note: The message will automatically be hidden upon next input.

void void vis_info_hide (Vis *)

Hide informational message.

void **vis_message_show**(Vis*, const char *msg)

Display arbitrary long message in a dedicated window.

1.20 Miscellaneous

Regex ***vis_regex**(Vis*, const char *pattern)

Get a regex object matching pattern.

Parameters

regex – The regex pattern to compile, if `NULL` the most recently used one is substituted.

Returns

A Regex object or `NULL` in case of an error.

Warning: The caller must free the regex object using *text_regex_free*.

void **vis_file_snapshot**(Vis*, File*)

Take an undo snapshot to which we can later revert.

Note: Does nothing when invoked while replaying a macro.

CHAPTER

TWO

TEXT

The core text management data structure which supports efficient modifications and provides a byte string interface. Text positions are represented as `size_t`. Valid addresses are in range `[0, text_size(txt)]`. An invalid position is denoted by `EPOS`. Access to the non-contigiuos pieces is available by means of an iterator interface or a copy mechanism. Text revisions are tracked in an history graph.

Note: The text is assumed to be encoded in UTF-8.

2.1 Load

enum **TextLoadMethod**

Method used to load existing file content.

Values:

enumerator **TEXT_LOAD_AUTO**

Automatically chose best option.

enumerator **TEXT_LOAD_READ**

Read file content and copy it to an in-memory buffer.

Subsequent changes to the underlying file will have no effect on this text instance.

Note: Load time is linear in the file size.

enumerator **TEXT_LOAD_MMAP**

Memory map the file from disk.

Use file system / virtual memory subsystem as a caching layer.

Note: Load time is (almost) independent of the file size.

Warning: Inplace modifications of the underlying file will be reflected in the current text content. In particular, truncation will raise `SIGBUS` and result in data loss.

Text *text_load(const char *filename)

Create a text instance populated with the given file content.

Note: Equivalent to `text_load_method(filename, TEXT_LOAD_AUTO)`.

Text *text_loadat(int dirfd, const char *filename)

Text *text_load_method(const char *filename, enum *TextLoadMethod*)

Create a text instance populated with the given file content.

Parameters

- **filename** – The name of the file to load, if `NULL` an empty text is created.
- **method** – How the file content should be loaded.

Returns

The new Text object or `NULL` in case of an error.

Note: When attempting to load a non-regular file, `errno` will be set to:

- `EISDIR` for a directory.
- `ENOTSUP` otherwise.

Text *text_loadat_method(int dirfd, const char *filename, enum *TextLoadMethod*)

void text_free(Text*)

Release all resources associated with this text instance.

2.2 State

size_t text_size(const Text*)

Return the size in bytes of the whole text.

struct stat text_stat(const Text*)

Get file information at time of load or last save, whichever happened more recently.

Note: If an empty text instance was created using `text_load(NULL)` and it has not yet been saved, an all zero `struct stat` will be returned.

Returns

See `stat(2)` for details.

bool text_modified(const Text*)

Query whether the text contains any unsaved modifications.

2.3 Modify

bool **text_insert**(Text*, size_t pos, const char *data, size_t len)

Insert data at the given byte position.

Parameters

- **pos** – The absolute byte position.
- **data** – The data to insert.
- **len** – The length of the data in bytes.

Returns

Whether the insertion succeeded.

bool **text_delete**(Text*, size_t pos, size_t len)

Delete data at given byte position.

Parameters

- **pos** – The absolute byte position.
- **len** – The number of bytes to delete, starting from pos.

Returns

Whether the deletion succeeded.

bool **text_delete_range**(Text*, const Filerange*)

```
bool text_printf (Text *, size_t pos, const char *format,...
) __attribute__((format(printf

bool bool text_appendf (Text *, const char *format,...) __attribute__((format(printf
```

2.4 Access

The individual pieces of the text are not necessarily stored in a contiguous memory block. These functions perform a copy to such a region.

bool **text_byte_get**(const Text*, size_t pos, char *byte)

Get byte stored at pos.

Parameters

- **pos** – The absolute position.
- **byte** – Destination address to store the byte.

Returns

Whether pos was valid and byte updated accordingly.

Note: Unlike `text_iterator_byte_get()` this function does not return an artificial NUL byte at EOF.

---

size_t **text_bytes_get**(const Text*, size_t pos, size_t len, char *buf)

Store at most `len` bytes starting from `pos` into `buf`.

Parameters

- **pos** – The absolute starting position.
- **len** – The length in bytes.
- **buf** – The destination buffer.

Returns

The number of bytes (<= `len`) stored at `buf`.

> **Warning:** `buf` will not be NUL terminated.

char ***text_bytes_alloc0**(const Text*, size_t pos, size_t len)

Fetch text range into newly allocate memory region.

Parameters

- **pos** – The absolute starting position.
- **len** – The length in bytes.

Returns

A contiguous NUL terminated buffer holding the requested range, or `NULL` in error case.

> **Warning:** The returned pointer must be freed by the caller.

2.5 Iterator

An iterator points to a given text position and provides interfaces to adjust said position or read the underlying byte value. Functions which take a `char` pointer will generally assign the byte value *after* the iterator was updated.

struct **Iterator**

Iterator used to navigate the buffer content.

Captures the position within a Piece.

Note: Should be treated as an opaque type.

> **Warning:** Any change to the Text will invalidate the iterator state.

Iterator **text_iterator_get**(const Text*, size_t pos)

bool **text_iterator_init**(const Text*, *Iterator*, size_t pos)

const Text ***text_iterator_text**(const *Iterator*)

bool **text_iterator_valid**(const *Iterator**)

bool **text_iterator_has_next**(const *Iterator**)

bool **text_iterator_has_prev**(const *Iterator**)

bool **text_iterator_next**(*Iterator**)

bool **text_iterator_prev**(*Iterator**)

2.5.1 Byte

Note: For a read attempt at EOF (i.e. *text_size*) an artificial `NUL` byte which is not actually part of the file is returned.

bool **text_iterator_byte_get**(const *Iterator**, char *b)

bool **text_iterator_byte_prev**(*Iterator**, char *b)

bool **text_iterator_byte_next**(*Iterator**, char *b)

bool **text_iterator_byte_find_prev**(*Iterator**, char b)

bool **text_iterator_byte_find_next**(*Iterator**, char b)

2.5.2 Codepoint

These functions advance to the next/previous leading byte of an UTF-8 encoded Unicode codepoint by skipping over all continuation bytes of the form `10xxxxxx`.

bool **text_iterator_codepoint_next**(*Iterator* *it, char *c)

bool **text_iterator_codepoint_prev**(*Iterator* *it, char *c)

2.5.3 Grapheme Clusters

These functions advance to the next/previous grapheme cluster.

Note: The grapheme cluster boundaries are currently not implemented according to UAX#29 rules. Instead a base character followed by arbitrarily many combining character as reported by `wcwidth(3)` are skipped.

bool **text_iterator_char_next**(*Iterator**, char *c)

bool **text_iterator_char_prev**(*Iterator**, char *c)

2.6 Lines

Translate between 1 based line numbers and 0 based byte offsets.

size_t **text_pos_by_lineno**(Text*, size_t lineno)

size_t **text_lineno_by_pos**(Text*, size_t pos)

2.7 History

Interfaces to the history graph.

bool **text_snapshot**(Text*)

Create a text snapshot, that is a vertex in the history graph.

size_t **text_undo**(Text*)

Revert to previous snapshot along the main branch.

Note: Takes an implicit snapshot.

Returns

The position of the first change or EPOS, if already at the oldest state i.e. there was nothing to undo.

size_t **text_redo**(Text*)

Reapply an older change along the main branch.

Note: Takes an implicit snapshot.

Returns

The position of the first change or EPOS, if already at the newest state i.e. there was nothing to redo.

size_t **text_earlier**(Text*)

size_t **text_later**(Text*)

size_t **text_restore**(Text*, time_t)

Restore the text to the state closest to the time given.

time_t **text_state**(const Text*)

Get creation time of current state.

Note: TODO: This is currently not the same as the time of the last snapshot.

2.8 Marks

A mark keeps track of a text position. Subsequent text changes will update all marks placed after the modification point. Reverting to an older text state will hide all affected marks, redoing the changes will restore them.

Warning: Due to an optimization cached modifications (i.e. no `text_snapshot` was performed between setting the mark and issuing the changes) might not adjust mark positions accurately.

typedef uintptr_t **Mark**

A mark.

EMARK

An invalid mark, lookup of which will yield `EPOS`.

Mark **text_mark_set**(Text*, size_t pos)

Set a mark.

Note: Setting a mark to `text_size` will always return the current text size upon lookup.

Parameters

pos – The position at which to store the mark.

Returns

The mark or `EMARK` if an invalid position was given.

size_t **text_mark_get**(const Text*, *Mark*)

Lookup a mark.

Parameters

mark – The mark to look up.

Returns

The byte position or `EPOS` for an invalid mark.

2.9 Save

enum **TextSaveMethod**

Method used to save the text.

Values:

enumerator **TEXT_SAVE_AUTO**

Automatically chose best option.

enumerator **TEXT_SAVE_ATOMIC**

Save file atomically using `rename(2)`.

Creates a temporary file, restores all important meta data, before moving it atomically to its final (possibly already existing) destination using `rename(2)`. For new files, permissions are set to `0666 & ~umask`.

> **Warning:** This approach does not work if:
>
> - The file is a symbolic link.
> - The file is a hard link.
> - File ownership can not be preserved.
> - File group can not be preserved.
> - Directory permissions do not allow creation of a new file.
> - POSIX ACL can not be preserved (if enabled).
> - SELinux security context can not be preserved (if enabled).

enumerator **TEXT_SAVE_INPLACE**

Overwrite file in place.

> **Warning:** I/O failure might cause data loss.

bool **text_save**(Text*, const char *filename)

Save the whole text to the given file name.

Note: Equivalent to `text_save_method(filename, TEXT_SAVE_AUTO)`.

---

bool **text_saveat**(Text*, int dirfd, const char *filename)

bool **text_save_method**(Text*, const char *filename, enum *TextSaveMethod*)

Save the whole text to the given file name, using the specified method.

bool **text_saveat_method**(Text*, int dirfd, const char *filename, enum *TextSaveMethod*)

TextSave ***text_save_begin**(Text*, int dirfd, const char *filename, enum *TextSaveMethod*)

Setup a sequence of write operations.

The returned `TextSave` pointer can be used to write multiple, possibly non-contiguous, file ranges.

> **Warning:** For every call to `text_save_begin` there must be exactly one matching call to either `text_save_commit` or `text_save_cancel` to release the underlying resources.

ssize_t **text_save_write_range**(TextSave*, const Filerange*)

Write file range.

Returns
The number of bytes written or `-1` in case of an error.

bool **text_save_commit**(TextSave*)

Commit changes to disk.

Returns
Whether changes have been saved.

> **Note:** Releases the underlying resources and frees the given `TextSave` pointer which must no longer be used.

void **text_save_cancel**(TextSave*)

Abort a save operation.

> **Note:** Does not guarantee to undo the previous writes (they might have been performed in-place). However, it releases the underlying resources and frees the given `TextSave` pointer which must no longer be used.

ssize_t **text_write**(const Text*, int fd)

Write whole text content to file descriptor.

Returns
The number of bytes written or `-1` in case of an error.

ssize_t **text_write_range**(const Text*, const Filerange*, int fd)

Write file range to file descriptor.

Returns
The number of bytes written or `-1` in case of an error.

2.10 Miscellaneous

bool **text_mmaped**(const Text*, const char *ptr)

Check whether `ptr` is part of a memory mapped region associated with this text instance.

CHAPTER

THREE

VIEW

Provides a viewport of a text instance and mangages selections.

3.1 Lifecycle

View ***view_new**(Text*)

void **view_free**(View*)

void **view_ui**(View*, UiWin*)

Text ***view_text**(View*)

void **view_reload**(View*, Text*)

3.2 Viewport

The cursor of the primary selection is always visible.

Filerange **view_viewport_get**(View*)

Get the currently displayed text range.

bool **view_coord_get**(View*, size_t pos, Line **line, int *row, int *col)

Get window coordinate of text position.

Parameters

- **pos** – The position to query.
- **line** – Will be updated with screen line on which `pos` resides.
- **row** – Will be updated with zero based window row on which `pos` resides.
- **col** – Will be updated with zero based window column on which `pos` resides.

Returns

Whether `pos` is visible. If not, the pointer arguments are left unmodified.

size_t **view_screenline_goto**(View*, int n)

Get position at the start of the `n`-th window line, counting from 1.

Line ***view_lines_first**(View*)

Get first screen line.

Line *view_lines_last(View*)
    Get last non-empty screen line.

size_t view_slide_up(View*, int lines)

size_t view_slide_down(View*, int lines)

size_t view_scroll_up(View*, int lines)

size_t view_scroll_down(View*, int lines)

size_t view_scroll_page_up(View*)

size_t view_scroll_page_down(View*)

size_t view_scroll_halfpage_up(View*)

size_t view_scroll_halfpage_down(View*)

void view_redraw_top(View*)

void view_redraw_center(View*)

void view_redraw_bottom(View*)

void view_scroll_to(View*, size_t pos)

3.3 Dimension

bool view_resize(View*, int width, int height)

int view_height_get(View*)

int view_width_get(View*)

3.4 Draw

void view_invalidate(View*)

void view_draw(View*)

bool view_update(View*)

3.5 Selections

A selection is a non-empty, directed range with two endpoints called *cursor* and *anchor*. A selection can be anchored in which case the anchor remains fixed while only the position of the cursor is adjusted. For non-anchored selections both endpoints are updated. A singleton selection covers one character on which both cursor and anchor reside. There always exists a primary selection which remains visible (i.e. changes to its position will adjust the viewport).

3.5.1 Creation and Destruction

Selection *`view_selections_new`(View*, size_t pos)

Create a new singleton selection at the given position.

Note: New selections are created non-anchored.

Warning: Fails if position is already covered by a selection.

Selection *`view_selections_new_force`(View*, size_t pos)

Create a new selection even if position is already covered by an existing selection.

Note: This should only be used if the old selection is eventually disposed.

---

bool `view_selections_dispose`(Selection*)

Dispose an existing selection.

Warning: Not applicable for the last existing selection.

bool `view_selections_dispose_force`(Selection*)

Forcefully dispose an existing selection.

If called for the last existing selection, it will be reduced and marked for destruction. As soon as a new selection is created this one will be disposed.

Selection *`view_selection_disposed`(View*)

Query state of primary selection.

If the primary selection was marked for destruction, return it and clear destruction flag.

void `view_selections_dispose_all`(View*)

Dispose all but the primary selection.

void `view_selections_normalize`(View*)

Dispose all invalid and merge all overlapping selections.

void `view_selections_set_all`(View*, *Array**, bool anchored)

Replace currently active selections.

Parameters

- **`array`** – The array of `Filerange` objects.
- **`anchored`** – Whether *all* selection should be anchored.

Array `view_selections_get_all`(View*)

Get array containing a `Fileranges` for each selection.

3.5.2 Navigation

Selection *`view_selections_primary_get`(View*)

void `view_selections_primary_set`(Selection*)

Selection *`view_selections`(View*)
: Get first selection.

Selection *`view_selections_prev`(Selection*)
: Get immediate predecessor of selection.

Selection *`view_selections_next`(Selection*)
: Get immediate successor of selection.

int `view_selections_count`(View*)
: Get number of existing selections.

Note: Is always at least 1.

---

int `view_selections_number`(Selection*)
: Get selection index.

Note: Is always in range `[0, count-1]`.

---

int `view_selections_column_count`(View*)
: Get maximal number of selections on a single line.

Selection *`view_selections_column`(View*, int column)
: Starting from the start of the text, get the `column`-th selection on a line.

  Parameters
  `column` – The zero based column index.

Selection *`view_selections_column_next`(Selection*, int column)
: Get the next `column`-th selection on a line.

  Parameters
  `column` – The zero based column index.

3.5.3 Cover

Filerange `view_selections_get`(Selection*)
: Get an inclusive range of the selection cover.

bool `view_selections_set`(Selection*, const Filerange*)
: Set selection cover.

  Updates both cursor and anchor.

void **view_selection_clear**(Selection*)

Reduce selection to character currently covered by the cursor.

Note: Sets selection to non-anchored mode.

---

void **view_selections_clear_all**(View*)

Reduce *all* currently active selections.

void **view_selections_flip**(Selection*)

Flip selection orientation.

Swap cursor and anchor.

Note: Has no effect on singleton selections.

3.5.4 Anchor

void **view_selections_anchor**(Selection*, bool anchored)

Anchor selection.

Further updates will only update the cursor, the anchor will remain fixed.

bool **view_selections_anchored**(Selection*)

Check whether selection is anchored.

3.5.5 Cursor

Selection endpoint to which cursor motions apply.

Properties

size_t **view_cursors_pos**(Selection*)

Get position of selection cursor.

size_t **view_cursors_line**(Selection*)

Get 1-based line number of selection cursor.

size_t **view_cursors_col**(Selection*)

Get 1-based column of selection cursor.

Note: Counts the number of graphemes on the logical line up to the cursor position.

Line ***view_cursors_line_get**(Selection*)

Get screen line of selection cursor.

int **view_cursors_cell_get**(Selection*)

Get zero based index of screen cell on which selection cursor currently resides.

> **Warning:** Returns `-1` if the selection cursor is currently not visible.

Placement

void **view_cursors_to**(Selection*, size_t pos)

Place cursor of selection at `pos`.

> **Note:** If the selection is not anchored, both selection endpoints will be adjusted to form a singleton selection covering one character starting at *pos*. Otherwise only the selection cursor will be changed while the anchor remains fixed.

void **view_cursors_scroll_to**(Selection*, size_t pos)

Adjusts window viewport until the requested position becomes visible.

> **Note:** For all but the primary selection this is equivalent to `view_selection_to`.

> **Warning:** Repeatedly redraws the window content. Should only be used for short distances between current cursor position and destination.

void **view_cursors_place**(Selection*, size_t line, size_t col)

Place cursor on given (line, column) pair.

Parameters

- **line** – the 1-based line number
- **col** – the 1 based column

> **Note:** Except for the different addressing format this is equivalent to *view_selection_to*.

int **view_cursors_cell_set**(Selection*, int cell)

Place selection cursor on zero based window cell index.

> **Warning:** Fails if the selection cursor is currently not visible.

Motions

These functions perform motions based on the current selection cursor position.

size_t **view_line_down**(Selection*)

size_t **view_line_up**(Selection*)

size_t **view_screenline_down**(Selection*)

size_t **view_screenline_up**(Selection*)

size_t **view_screenline_begin**(Selection*)

size_t **view_screenline_middle**(Selection*)

size_t **view_screenline_end**(Selection*)

3.5.6 Primary Selection

These are convenience function which operate on the primary selection.

void **view_cursor_to**(View*, size_t pos)

Move primary selection cursor to the given position.

Makes sure that position is visible.

Note: If position was not visible before, we attempt to show surrounding context. The viewport will be adjusted such that the line holding the cursor is shown in the middle of the window.

---

size_t **view_cursor_get**(View*)

Get cursor position of primary selection.

Filerange **view_selection_get**(View*)

Get primary selection.

Note: Is always a non-empty range.

3.5.7 Save and Restore

Filerange **view_regions_restore**(View*, SelectionRegion*)

bool **view_regions_save**(View*, Filerange*, SelectionRegion*)

3.6 Style

void **view_options_set**(View*, enum UiOption options)

enum UiOption **view_options_get**(View*)

void **view_colorcolumn_set**(View*, int col)

int **view_colorcolumn_get**(View*)

void **view_wrapcolumn_set**(View*, int col)

int **view_wrapcolumn_get**(View*)

bool **view_breakat_set**(View*, const char *breakat)

const char ***view_breakat_get**(View*)

void **view_tabwidth_set**(View*, int tabwidth)

Set how many spaces are used to display a tab `\t` character.

int **view_tabwidth_get**(View*)

Get how many spaces are used to display a tab `\t` character.

bool **view_style_define**(View*, enum UiStyle, const char *style)

Define a display style.

void **view_style**(View*, enum UiStyle, size_t start, size_t end)

Apply a style to a text range.

char ***view_symbol_eof_get**(View*)

CHAPTER

FOUR

BUFFER

A dynamically growing buffer storing arbitrary data.

Note: Used for Register, *not* Text content.

Functions

void **buffer_init**(*Buffer**)
: Initialize a *Buffer* object.

void **buffer_release**(*Buffer**)
: Release all resources, reinitialize buffer.

void **buffer_clear**(*Buffer**)
: Set buffer length to zero, keep allocated memory.

bool **buffer_reserve**(*Buffer**, size_t size)
: Reserve space to store at least `size` bytes.

bool **buffer_grow**(*Buffer**, size_t len)
: Reserve space for at least `len` *more* bytes.

bool **buffer_terminate**(*Buffer**)
: If buffer is non-empty, make sure it is `NUL` terminated.

bool **buffer_put**(*Buffer**, const void *data, size_t len)
: Set buffer content, growing the buffer as needed.

bool **buffer_put0**(*Buffer**, const char *data)
: Set buffer content to `NUL` terminated data.

bool **buffer_remove**(*Buffer**, size_t pos, size_t len)
: Remove `len` bytes starting at `pos`.

bool **buffer_insert**(*Buffer**, size_t pos, const void *data, size_t len)
: Insert `len` bytes of `data` at `pos`.

bool **buffer_insert0**(*Buffer**, size_t pos, const char *data)
: Insert NUL-terminated data at pos.

bool **buffer_append**(*Buffer**, const void *data, size_t len)
: Append further content to the end.

bool **buffer_append0**(*Buffer**, const char *data)

Append NUL-terminated data.

bool **buffer_prepend**(*Buffer**, const void *data, size_t len)

Insert `len` bytes of `data` at the start.

bool **buffer_prepend0**(*Buffer**, const char *data)

Insert NUL-terminated data at the start.

bool buffer_printf (Buffer *, const char *fmt,...) __attribute__((format(printf

Set formatted buffer content, ensures NUL termination on success.

bool bool buffer_appendf (Buffer *, const char *fmt,...) __attribute__((format(printf

Append formatted buffer content, ensures NUL termination on success.

bool bool size_t buffer_length0 (Buffer *)

Return length of a buffer without trailing NUL byte.

size_t **buffer_length**(*Buffer**)

Return length of a buffer including possible NUL byte.

size_t **buffer_capacity**(*Buffer**)

Return current maximal capacity in bytes of this buffer.

const char ***buffer_content0**(*Buffer**)

Get pointer to buffer data.

Guaranteed to return a NUL terminated string even if buffer is empty.

const char ***buffer_content**(*Buffer**)

Get pointer to buffer data.

> **Warning:** Might be NULL, if empty. Might not be NUL terminated.

char ***buffer_move**(*Buffer**)

Borrow underlying buffer data.

> **Warning:** The caller is responsible to `free(3)` it.

struct **Buffer**

#include <buffer.h> A dynamically growing buffer storing arbitrary data.

Public Members

char *`data`

Data pointer, `NULL` if empty.

size_t `len`

Current length of data.

size_t `size`

Maximal capacity of the buffer.

CHAPTER

FIVE

ARRAY

A dynamically growing array, there exist two typical ways to use it:

1. To hold pointers to externally allocated memory regions.

   Use `array_init` for initialization, an element has the size of a pointer. Use the functions suffixed with `_ptr` to manage your pointers. The cleanup function `array_release_full` must only be used with this type of array.

2. To hold arbitrary sized objects.

   Use `array_init_sized` to specify the size of a single element. Use the regular (i.e. without the `_ptr` suffix) functions to manage your objects. Functions like `array_add` and `array_set` will copy the object into the array, `array_get` will return a pointer to the object stored within the array.

Functions

void **array_init**(*Array**)

Initialize an *Array* object to store pointers.

Note: Is equivalent to `array_init_sized(arr, sizeof(void*))`.

---

void **array_init_sized**(*Array**, size_t elem_size)

Initialize an *Array* object to store arbitrarily sized objects.

void **array_init_from**(*Array**, const *Array* *from)

Initialize *Array* by using the same element size as in `from`.

void **array_release**(*Array**)

Release storage space.

Reinitializes *Array* object.

void **array_release_full**(*Array**)

Release storage space and call `free(3)` for each stored pointer.

> **Warning:** Assumes array elements to be pointers.

void **array_clear**(*Array**)

Empty array, keep allocated memory.

bool **array_reserve**(*Array**, size_t count)

Reserve memory to store at least `count` elements.

void ***array_get**(const *Array**, size_t idx)

Get array element.

> **Warning:** Returns a pointer to the allocated array region. Operations which might cause reallocations (e.g. the insertion of new elements) might invalidate the pointer.

bool **array_set**(*Array**, size_t idx, void *item)

Set array element.

> **Note:** Copies the `item` into the Array. If `item` is `NULL` the corresponding memory region will be cleared.

void ***array_get_ptr**(const *Array**, size_t idx)

Dereference pointer stored in array element.

bool **array_set_ptr**(*Array**, size_t idx, void *item)

Store the address to which `item` points to into the array.

bool **array_add**(*Array**, void *item)

Add element to the end of the array.

bool **array_add_ptr**(*Array**, void *item)

Add pointer to the end of the array.

bool **array_remove**(*Array**, size_t idx)

Remove an element by index.

> **Note:** Might not shrink underlying memory region.

size_t **array_length**(const *Array**)

Number of elements currently stored in the array.

size_t **array_capacity**(const *Array**)

Number of elements which can be stored without enlarging the array.

bool **array_truncate**(*Array**, size_t length)

Remove all elements with index greater or equal to `length`, keep allocated memory.

bool **array_resize**(*Array**, size_t length)

Change length.

> **Note:** Has to be less or equal than the capacity. Newly accessible elements preserve their previous values.

void **array_sort**(*Array**, int (*compar)(const void*, const void*))

Sort array, the comparision function works as for `qsort(3)`.

bool **array_push**(*Array**, void *item)

Push item onto the top of the stack.

Note: Is equivalent to `array_add(arr, item)`.

---

void ***array_pop**(*Array**)

Get and remove item at the top of the stack.

Warning: The same ownership rules as for `array_get` apply.

void ***array_peek**(const *Array**)

Get item at the top of the stack without removing it.

Warning: The same ownership rules as for `array_get` apply.

struct **Array**

#include <array.h> A dynamically growing array.

Public Members

char ***items**

size_t **elem_size**

Data pointer, NULL if empty.

size_t **len**

Size of one array element.

size_t **count**

Number of currently stored items.

CHAPTER

SIX

MAP

Crit-bit tree based map which supports unique prefix queries and ordered iteration.

Functions

Map *`map_new`(void)

Allocate a new map.

void *`map_get`(const Map*, const char *key)

Lookup a value, returns `NULL` if not found.

void *`map_first`(const Map*, const char **key)

Get first element of the map, or `NULL` if empty.

Parameters
`key` – Updated with the key of the first element.

void *`map_closest`(const Map*, const char *prefix)

Lookup element by unique prefix match.

Parameters
`prefix` – The prefix to search for.

Returns
The corresponding value, if the given prefix is unique. Otherwise `NULL`. If no such prefix exists, then `errno` is set to `ENOENT`.

bool `map_contains`(const Map*, const char *prefix)

Check whether the map contains the given prefix.

whether it can be extended to match a key of a map element.

bool `map_put`(Map*, const char *key, const void *value)

Store a key value pair in the map.

Returns
False if we run out of memory (`errno = ENOMEM`), or if the key already appears in the map (`errno = EEXIST`).

void *`map_delete`(Map*, const char *key)

Remove a map element.

Returns
The removed entry or `NULL` if no such element exists.

bool **map_copy**(Map *dest, Map *src)

Copy all entries from `src` into `dest`, overwrites existing entries in `dest`.

void **map_iterate**(const Map*, bool (*handle)(const char *key, void *value, void *data), const void *data)

Ordered iteration over a map.

Invokes the passed callback for every map entry. If `handle` returns false, the iteration will stop.

Parameters

- **handle** – A function invoked for ever map element.
- **data** – A context pointer, passed as last argument to `handle`.

const Map ***map_prefix**(const Map*, const char *prefix)

Get a sub map matching a prefix.

> **Warning:** This returns a pointer into the original map. Do not alter the map while using the return value.

bool **map_empty**(const Map*)

Test whether the map is empty (contains no elements).

void **map_clear**(Map*)

Empty the map.

void **map_free**(Map*)

Release all memory associated with this map.

void **map_free_full**(Map*)

Call `free(3)` for every map element, then free the map itself.

> **Warning:** Assumes map elements to be pointers.

INDEX

A

Array (*C++ struct*), 41
Array::count (*C++ member*), 41
Array::elem_size (*C++ member*), 41
Array::items (*C++ member*), 41
Array::len (*C++ member*), 41
array_add (*C++ function*), 40
array_add_ptr (*C++ function*), 40
array_capacity (*C++ function*), 40
array_clear (*C++ function*), 39
array_get (*C++ function*), 40
array_get_ptr (*C++ function*), 40
array_init (*C++ function*), 39
array_init_from (*C++ function*), 39
array_init_sized (*C++ function*), 39
array_length (*C++ function*), 40
array_peek (*C++ function*), 41
array_pop (*C++ function*), 41
array_push (*C++ function*), 40
array_release (*C++ function*), 39
array_release_full (*C++ function*), 39
array_remove (*C++ function*), 40
array_reserve (*C++ function*), 39
array_resize (*C++ function*), 40
array_set (*C++ function*), 40
array_set_ptr (*C++ function*), 40
array_sort (*C++ function*), 40
array_truncate (*C++ function*), 40

B

Buffer (*C++ struct*), 36
Buffer::data (*C++ member*), 37
Buffer::len (*C++ member*), 37
Buffer::size (*C++ member*), 37
buffer_append (*C++ function*), 35
buffer_append0 (*C++ function*), 35
buffer_capacity (*C++ function*), 36
buffer_clear (*C++ function*), 35
buffer_content (*C++ function*), 36
buffer_content0 (*C++ function*), 36
buffer_grow (*C++ function*), 35
buffer_init (*C++ function*), 35
buffer_insert (*C++ function*), 35
buffer_insert0 (*C++ function*), 35
buffer_length (*C++ function*), 36
buffer_move (*C++ function*), 36
buffer_prepend (*C++ function*), 36
buffer_prepend0 (*C++ function*), 36
buffer_put (*C++ function*), 35
buffer_put0 (*C++ function*), 35
buffer_release (*C++ function*), 35
buffer_remove (*C++ function*), 35
buffer_reserve (*C++ function*), 35
buffer_terminate (*C++ function*), 35

E

EMARK (*C macro*), 23

I

Iterator (*C++ struct*), 20

K

KeyActionFunction (*C++ type*), 5

M

map_clear (*C++ function*), 44
map_closest (*C++ function*), 43
map_contains (*C++ function*), 43
map_copy (*C++ function*), 43
map_delete (*C++ function*), 43
map_empty (*C++ function*), 44
map_first (*C++ function*), 43
map_free (*C++ function*), 44
map_free_full (*C++ function*), 44
map_get (*C++ function*), 43
map_iterate (*C++ function*), 44
map_new (*C++ function*), 43
map_prefix (*C++ function*), 44
map_put (*C++ function*), 43
Mark (*C++ type*), 23

T

text_byte_get (*C++ function*), 19

text_bytes_alloc0 (C++ function), 20
text_bytes_get (C++ function), 19
text_delete (C++ function), 19
text_delete_range (C++ function), 19
text_earlier (C++ function), 22
text_free (C++ function), 18
text_insert (C++ function), 19
text_iterator_byte_find_next (C++ function), 21
text_iterator_byte_find_prev (C++ function), 21
text_iterator_byte_get (C++ function), 21
text_iterator_byte_next (C++ function), 21
text_iterator_byte_prev (C++ function), 21
text_iterator_char_next (C++ function), 21
text_iterator_char_prev (C++ function), 21
text_iterator_codepoint_next (C++ function), 21
text_iterator_codepoint_prev (C++ function), 21
text_iterator_get (C++ function), 20
text_iterator_has_next (C++ function), 21
text_iterator_has_prev (C++ function), 21
text_iterator_init (C++ function), 20
text_iterator_next (C++ function), 21
text_iterator_prev (C++ function), 21
text_iterator_text (C++ function), 20
text_iterator_valid (C++ function), 20
text_later (C++ function), 22
text_lineno_by_pos (C++ function), 22
text_load (C++ function), 18
text_load_method (C++ function), 18
text_loadat (C++ function), 18
text_loadat_method (C++ function), 18
text_mark_get (C++ function), 23
text_mark_set (C++ function), 23
text_mmaped (C++ function), 25
text_modified (C++ function), 18
text_pos_by_lineno (C++ function), 22
text_redo (C++ function), 22
text_restore (C++ function), 22
text_save (C++ function), 24
text_save_begin (C++ function), 24
text_save_cancel (C++ function), 25
text_save_commit (C++ function), 25
text_save_method (C++ function), 24
text_save_write_range (C++ function), 25
text_saveat (C++ function), 24
text_saveat_method (C++ function), 24
text_size (C++ function), 18
text_snapshot (C++ function), 22
text_stat (C++ function), 18
text_state (C++ function), 22
text_undo (C++ function), 22
text_write (C++ function), 25
text_write_range (C++ function), 25
TextLoadMethod (C++ enum), 17
TextLoadMethod::TEXT_LOAD_AUTO (C++ enumerator), 17
TextLoadMethod::TEXT_LOAD_MMAP (C++ enumerator), 17
TextLoadMethod::TEXT_LOAD_READ (C++ enumerator), 17
TextSaveMethod (C++ enum), 24
TextSaveMethod::TEXT_SAVE_ATOMIC (C++ enumerator), 24
TextSaveMethod::TEXT_SAVE_AUTO (C++ enumerator), 24
TextSaveMethod::TEXT_SAVE_INPLACE (C++ enumerator), 24

V

view_breakat_get (C++ function), 34
view_breakat_set (C++ function), 34
view_colorcolumn_get (C++ function), 34
view_colorcolumn_set (C++ function), 34
view_coord_get (C++ function), 27
view_cursor_get (C++ function), 33
view_cursor_to (C++ function), 33
view_cursors_cell_get (C++ function), 31
view_cursors_cell_set (C++ function), 32
view_cursors_col (C++ function), 31
view_cursors_line (C++ function), 31
view_cursors_line_get (C++ function), 31
view_cursors_place (C++ function), 32
view_cursors_pos (C++ function), 31
view_cursors_scroll_to (C++ function), 32
view_cursors_to (C++ function), 32
view_draw (C++ function), 28
view_free (C++ function), 27
view_height_get (C++ function), 28
view_invalidate (C++ function), 28
view_line_down (C++ function), 33
view_line_up (C++ function), 33
view_lines_first (C++ function), 27
view_lines_last (C++ function), 27
view_new (C++ function), 27
view_options_get (C++ function), 34
view_options_set (C++ function), 34
view_redraw_bottom (C++ function), 28
view_redraw_center (C++ function), 28
view_redraw_top (C++ function), 28
view_regions_restore (C++ function), 33
view_regions_save (C++ function), 33
view_reload (C++ function), 27
view_resize (C++ function), 28
view_screenline_begin (C++ function), 33
view_screenline_down (C++ function), 33
view_screenline_end (C++ function), 33
view_screenline_goto (C++ function), 27
view_screenline_middle (C++ function), 33

view_screenline_up (C++ function), 33
view_scroll_down (C++ function), 28
view_scroll_halfpage_down (C++ function), 28
view_scroll_halfpage_up (C++ function), 28
view_scroll_page_down (C++ function), 28
view_scroll_page_up (C++ function), 28
view_scroll_to (C++ function), 28
view_scroll_up (C++ function), 28
view_selection_clear (C++ function), 30
view_selection_disposed (C++ function), 29
view_selection_get (C++ function), 33
view_selections (C++ function), 30
view_selections_anchor (C++ function), 31
view_selections_anchored (C++ function), 31
view_selections_clear_all (C++ function), 31
view_selections_column (C++ function), 30
view_selections_column_count (C++ function), 30
view_selections_column_next (C++ function), 30
view_selections_count (C++ function), 30
view_selections_dispose (C++ function), 29
view_selections_dispose_all (C++ function), 29
view_selections_dispose_force (C++ function), 29
view_selections_flip (C++ function), 31
view_selections_get (C++ function), 30
view_selections_get_all (C++ function), 29
view_selections_new (C++ function), 29
view_selections_new_force (C++ function), 29
view_selections_next (C++ function), 30
view_selections_normalize (C++ function), 29
view_selections_number (C++ function), 30
view_selections_prev (C++ function), 30
view_selections_primary_get (C++ function), 30
view_selections_primary_set (C++ function), 30
view_selections_set (C++ function), 30
view_selections_set_all (C++ function), 29
view_slide_down (C++ function), 28
view_slide_up (C++ function), 28
view_style (C++ function), 34
view_style_define (C++ function), 34
view_symbol_eof_get (C++ function), 34
view_tabwidth_get (C++ function), 34
view_tabwidth_set (C++ function), 34
view_text (C++ function), 27
view_ui (C++ function), 27
view_update (C++ function), 28
view_viewport_get (C++ function), 27
view_width_get (C++ function), 28
view_wrapcolumn_get (C++ function), 34
view_wrapcolumn_set (C++ function), 34
vis_action_free (C++ function), 6
vis_action_new (C++ function), 6
vis_action_register (C++ function), 6
vis_binding_free (C++ function), 5
vis_binding_new (C++ function), 5
vis_cancel (C++ function), 8
vis_cmd (C++ function), 12
vis_cmd_register (C++ function), 12
vis_cmd_unregister (C++ function), 12
vis_count_get (C++ function), 7
vis_count_get_default (C++ function), 7
vis_count_iterator_get (C++ function), 7
vis_count_iterator_init (C++ function), 7
vis_count_iterator_next (C++ function), 7
vis_count_set (C++ function), 7
VIS_COUNT_UNKNOWN (C macro), 7
vis_delete (C++ function), 14
vis_doupdates (C++ function), 1
vis_draw (C++ function), 2
vis_exit (C++ function), 1
vis_file_snapshot (C++ function), 15
vis_free (C++ function), 1
vis_insert (C++ function), 14
vis_insert_key (C++ function), 14
vis_insert_nl (C++ function), 15
vis_insert_tab (C++ function), 14
vis_interrupt (C++ function), 2
vis_interrupt_requested (C++ function), 2
vis_jumplist_next (C++ function), 10
vis_jumplist_prev (C++ function), 10
vis_jumplist_save (C++ function), 10
vis_keymap_add (C++ function), 4
vis_keymap_disable (C++ function), 4
vis_keys_codepoint (C++ function), 4
vis_keys_feed (C++ function), 4
vis_keys_next (C++ function), 4
vis_keys_utf8 (C++ function), 4
vis_macro_record (C++ function), 11
vis_macro_record_stop (C++ function), 11
vis_macro_recording (C++ function), 11
vis_macro_replay (C++ function), 11
vis_mark (C++ function), 10
vis_mark_from (C++ function), 10
vis_mark_get (C++ function), 10
vis_mark_normalize (C++ function), 10
vis_mark_set (C++ function), 10
vis_mark_to (C++ function), 10
vis_mark_used (C++ function), 10
vis_message_show (C++ function), 15
vis_mode_from (C++ function), 7
vis_mode_get (C++ function), 7
vis_mode_map (C++ function), 5
vis_mode_switch (C++ function), 7
vis_mode_unmap (C++ function), 5
vis_motion (C++ function), 9
vis_motion_register (C++ function), 9
vis_motion_type (C++ function), 9
vis_new (C++ function), 1
vis_operator (C++ function), 8

vis_operator_register (C++ function), 8
vis_option_register (C++ function), 13
vis_option_unregister (C++ function), 13
vis_pipe (C++ function), 13
vis_pipe_collect (C++ function), 14
vis_prompt_cmd (C++ function), 13
vis_prompt_show (C++ function), 15
vis_redraw (C++ function), 2
vis_regex (C++ function), 15
vis_register (C++ function), 11
vis_register_from (C++ function), 11
vis_register_get (C++ function), 11
vis_register_set (C++ function), 11
vis_register_to (C++ function), 11
vis_register_used (C++ function), 11
vis_repeat (C++ function), 8
vis_replace (C++ function), 14
vis_replace_key (C++ function), 14
vis_resume (C++ function), 1
vis_run (C++ function), 1
vis_shell_set (C++ function), 7
vis_signal_handler (C++ function), 2
vis_suspend (C++ function), 1
vis_tabwidth_set (C++ function), 7
vis_textobject (C++ function), 10
vis_textobject_register (C++ function), 9
vis_update (C++ function), 2
vis_window_closable (C++ function), 3
vis_window_close (C++ function), 3
vis_window_draw (C++ function), 3
vis_window_focus (C++ function), 3
vis_window_height_get (C++ function), 3
vis_window_invalidate (C++ function), 3
vis_window_mode_map (C++ function), 5
vis_window_mode_unmap (C++ function), 5
vis_window_new (C++ function), 3
vis_window_new_fd (C++ function), 3
vis_window_next (C++ function), 3
vis_window_prev (C++ function), 3
vis_window_reload (C++ function), 3
vis_window_split (C++ function), 3
vis_window_status (C++ function), 3
vis_window_swap (C++ function), 3
vis_window_width_get (C++ function), 3
VisCountIterator (C++ struct), 8
VisMode (C++ enum), 6
VisMode::VIS_MODE_INSERT (C++ enumerator), 6
VisMode::VIS_MODE_INVALID (C++ enumerator), 7
VisMode::VIS_MODE_NORMAL (C++ enumerator), 6
VisMode::VIS_MODE_OPERATOR_PENDING (C++ enumerator), 6
VisMode::VIS_MODE_REPLACE (C++ enumerator), 7
VisMode::VIS_MODE_VISUAL (C++ enumerator), 6
VisMode::VIS_MODE_VISUAL_LINE (C++ enumerator), 6
VisMotionType (C++ enum), 9
VisMotionType::VIS_MOTIONTYPE_CHARWISE (C++ enumerator), 9
VisMotionType::VIS_MOTIONTYPE_LINEWISE (C++ enumerator), 9
VisOption (C++ enum), 12
VisOption::VIS_OPTION_DEPRECATED (C++ enumerator), 13
VisOption::VIS_OPTION_NEED_WINDOW (C++ enumerator), 12
VisOption::VIS_OPTION_TYPE_BOOL (C++ enumerator), 12
VisOption::VIS_OPTION_TYPE_NUMBER (C++ enumerator), 12
VisOption::VIS_OPTION_TYPE_STRING (C++ enumerator), 12
VisOption::VIS_OPTION_VALUE_OPTIONAL (C++ enumerator), 12