1 Assembled Developer Manual

5/10/2023

2 Contents

This document contains the full documentation of all underlying functions within the Assembled text editor.

        2. struct text_file *load_file(name)
        3. void save_file(file)
        4. void destroy_file(file)
    5. buffer.c
2. Interface Half
    1. interface.c
        1. int register_screen(name, render, updated, local)
        2. int switch_to_screen(name)
    2. themes.c
        1. static void read_theme(file)
        2. void register_custom_colors()
        3. struct cfg_token *configure_theme(token)
5. Code Style

3 Configuration File Structure

The structure of the configuration file is extremely simple:

Command	Argument
keyboard	keyseq func-name:0x41,66,’c’
start_screen	background_disable

There are two columns separated by whitespace.

If a command begins with a ‘#’ symbol, then it is discarded.

3.1 Configuration Language Syntax

HEX = 0x0123456789ABCDEF

DECIMAL = 1234567890

OCTAL = 012345670

ASCII = Any ASCII character wrapped in single quotes (i.e. ‘A’).

STRING LITERAL = A series of alphanumeric characters including ‘_’ characters.

** Whitespace is ignored**

Program:

Command

Argument

Program

Command:

String Literal (Keyword.CMD *):

- “keyboard”
- “start_screen”
- “themes”

Argument:

STRING LITERAL (Keyword.ARG *)

| STRING LITERAL (Keyword.ARG) ‘:’ Value

Value:

STRING LITERAL [, Value]

| Integer [, Value]

| ASCII [, Value]

Integer:

HEX

| DECIMAL

| OCTAL

* Keyword.CMD is a keyword which is recognized only in the command column.

* Keyword.ARG is a keyword which is recognized only in the arguments column.

* These keywords are specified in the sections below.

* Comments can start anywhere, but always end at the end of a line. The character to declare a comment is ‘#’.

3.2 Keyboard Configuration

To configure the keyboard, the command must be “keyboard” and the argument must follow this format:

keyseq *A*:*B*

A is the name of the function which should be called when the sequence of keys in *B* are pressed.

An example of *B* is:

‘A’, 65, 0x41, 0102

This list may do with or without the commas.

3.3 Start Screen Configuration

To configure the start screen, the command must be “start_screen” and there are currently two arguments.

3.3.1 background_off

This disables the background generation within the start screen. The argument follows exactly with the heading.

3.3.2 logo_bmp:*A*

A is the path to the 64x40 monochrome BMP to be used as the logo. If *A* is not an absolute path, then the path will be prefixed with “/home/$USER/.config/assembled/”, where $USER is the name of the user. The argument follows the heading exactly.

3.4 Theme Configuration

Theme configuration comes in two parts: within the configuration file, and within the theme file.

3.4.1 config.cfg

To specify which themes file to load, use the following line is used:

```
themes          use:themes/theme.cfg
```

Here, the path is not prefixed with a ‘/’ indicating that it is not an absolute path. This means that the path is prefixed with with the configuration directory “/home/$USER/.config/assembled/”.

```
themes          use:/home/$USER/.config/assembled/themes/config.cfg
```

The above is an example of an absolute path. *Please note that $USER is a place holder for the name of the user*.

3.4.2 theme.cfg

Theme configuration values are formatted as follows:

A:*B*

Where *A* is the integer index in the 32 entry color table (zero-based) and *B* is the 0xRRGGBB representation of the color. Following is an example:

0:0x6FD184

4 Source Code Directory Structure

The directory structure in implementation currently looks like the following list:

- src/
  - editor/
    - buffer/
      - buffer.c
      - editor.c
    - config.c
    - functions.c
    - keyboard.c
  - include/
    - editor/
      - buffer/
        - buffer.h
        - editor.h
      - config.h
      - functions.h
      - keyboard.h
    - interface/
      - screens/
        - editor_scr.h
        - start.h
      - theming/
        - themes.h
      - interface.h
    - global.h
    - util.h
  - interface/
    - screens/

        - editor_scr.h
        - start.c
    - theming/
        - themes.c
    - interface.c
- main.c
- util.c

The program can be split into two parts: update and render. The editor part defines functions which are used for the functionality of the program, and interface defines functions which are used to display information to the user.

An example of functionality is the keyboard. The user presses a key, which the program should interpret, this interpretation is handleld by *keyboard.c*.

Likewise, an example of rendering is the screen interface. There are various different screens which the user can access and the program needs to somehow manage which screen is being rendered and updated.

main.c is used for concatenating the two halves into one functioning program.

4.1 Editor Half

The editor is responsible for handling non-visual tasks. Examples of non-visual tasks are:

- Reading the configuration file
- Reading user input
- Interpreting user input
- Updating internal text buffers
- Converting internal text buffers to files

Currently there are three source files in the editor directory: *config.c*, *functions.c*, *keyboard.c*.

4.1.1 config.c

This file is responsible for reading configuration files. A given configuration file is lexed using the BNF tree found in 3.1. Once the file is converted to tokens, then this stream is interpreted.

This interpreter splits its work up. It first reads the first two tokens (these are apart of the command column). These two must be of type KEY and TAB.

From here, an appropriate configuration function is derived from the KEY token's value. These configuration functions must return a *struct cfg_token* * (this should be the last token that the function has touched), and accept an argument of *struct cfg_token* * (this is the starting token given by interpreter).

void interpret_token_stream(token)

token – Head of the linked list of tokens to interpret

struct token *

*struct token *cfg_lex(file)*

Returns the head of the linked list token stream.

file – Open file pointer to lex

FILE *

int read_config()

Reads the configuration file located at “/home/$USER/.config/assembled/config.cfg”.

Returns the number of extraneous lines – lines with errors – it encountered.

4.1.2 functions.c

functions.c contains the implementation of a hashmap of functions and the functions themselves.

4.1.3 keyboard.c

Contains a series of functions used for submitting and interpreting key presses, as well as a function to set up, configure, the key layers which are used to interpret each key press.

4.1.3.1 Run-time

When a key is pressed, the *key* function is called. This function pushes the key to the key stack, advances the key stack pointer, and tries to collapse the stack.

Collapsing the stack is done through the *collapse_stack* function. Which will iterate from the bottom of the stack to the top, indexing each element with the letter *i*. It will use the current value it observes at *i* as the index into the function array member of the current key layer.

If the function pointer that is retrieved is NULL, the stack pointer is reset and interpretation is stopped.

If the function pointer that is retrieved is equivalent to *layer_down* then the current key layer will be swapped to the next, and the loop will be put onto its next cycle.

Finally, if the loop has found a function which is non-NULL and a non-*layer_down* then it will jump to that function. Reseting the key stack pointer just before leaving the *collapse_stack* function.

4.1.3.2 Configuration

keyboard.c is configured through the use of the *configure_keyboard* function. As described above by section 4.1.1, *configure_keyboard* function receives a *struct cfg_token* * and returns one.

The functions starts by checking that the token it has received has a type of KEY and a value of LOOKUP_KEYSEQ. If this test passses, it moves onto the next token. This token must have a type of STR. This is the function name, and the value of the token is hashed and converted to an index.

The next token must be of type COL.

Next is a series of integers which correspond to the key sequence that should be inputted to call the function specified earlier. This series is scanned through until a non-INT or a non-COM token is reached.

Each INT token represents an index into the *function* array, which contains 256 function pointers. If the current token is not the last, then the function pointer will be equivalent to the *layer_down* function. When the last token is reached, the pointer will be set to the selected function.

void key(c)

Push the given character, c, to the key stack.

c – Character to push to the key stack

char

*struct cfg_token *configure_keyboard(token)*

Interpret the given list of tokens.

token – Head node of the current argument column

struct cfg_token *

4.1.4 editor.c

editor.c manages *text_files*. *text_files* are the internal representation of an input file. They are structured as such:

```
char *name;
FILE *file;
int load_offset;
int cy;
struct text_buffer **buffers;
struct text_buffer *active_buffer;
int active_buffer_idx;
```

name is the path to the loaded file.

file is the pointer to the loaded file.

load_offset is the offset from the beginning of the file at which the first line in the first buffer starts.

cy is the current position of the cursor on the y-axis.

buffers is a list of pointers pointing to each buffer.

active_buffer is a pointer to the buffer which the user is currently on.

active_buffer_idx is an index within *buffers* denoting the buffer the user is currently on.

Buffers can also be thought of as the columns.

Columns are described by the following *column_descriptor* structure:

```
int column_count;
int *column_posistions;
int delimiter;
```

column_count represents the number of integers within *column_positions*.

column_posistions holds the list of starting positions for each column going left to right.

delimiter is the character by which a column boundary is indicated.

There are a total of 32 column descriptors, giving the user 32 different column layouts.

4.1.4.1 Configuration

The *editor.c* file is also responsible for reading in the columns specified in the configuration file. The command to configure columns is “columns” and there are two arguments:

- define:[*columns*]:*delimiter*:*posistion*
  - *columns* is a list of integers separated

*struct text_file *load_file(name)*

This function initializes a *text_file* structure with the contents of the given file.

name – The absolute path to the file

char *

void save_file(file)

This function writes the file. It from the first to last buffer, writing the current line of the buffer. Inbetween every buffer the current column descriptor’s delimiter is inserted.

file – The text file to save

struct text_file *

void destroy_file(file)

This function frees the memory occupied by a given text file.

file – The text file to destroy

struct text_file *

4.1.5 buffer.c

buffer.c is responsible for managing everything relating to the active text file's active text buffer. Depending on the input from the keyboard, it will either insert a character, insert a new line, delete a character, or delete a line.

There are two structures buffers use: *text_buffer* and *line_list_element*.

text_buffer structures describe the actual buffer itself, and their constitution is as follows:

```
int cx;
int col_start;
int col_end;
struct line_list_element *head;
struct line_list_element *current;
```

cx is the cursor's position on the x-axis in the buffer. *col_start* and *col_end* define the boundaries of the buffer. *head* points to the first line in the buffer, and *current* points to the current element.

line_list_element is constituted as such:

```
char *contents;
struct line_list_element *next;
struct line_list_element *prev;
```

contents is the string containing a single line of the loaded file (excluding the new line character '\n'). *next* and *prev* point the the next and previous elements within the linked list respectively.

*struct text_buffer *new_buffer(col_start, col_end)*

Returns a pointer to a newly initialized buffer.

col_start – The column at which this buffer starts

Int

This value must always be positive.

col_end – The column at which this buffer ends

Int

This value may be negative one, which represents the maximum x coordinate.

void destroy_buffer(buffer);

Frees the given buffer and its members.

buffer – The buffer to destroy

struct text_buffer *

void buffer_char_insert(char c);

In the active text file’s active buffer, place this character at cy and cx.

c – The character to place into the buffer

char

void buffer_char_del();

In the active text file’s active text buffer remove a character at cy and cx.

4.2 Interface Half

The interface manages all possible screens which can be displayed to the user at any time.

interface/interface.c contains two functions: register_screen and switch_to_screen.

Screens should have a *register*_screen-name function which registers them into the hashmap. Functions *render*, *update*, and *local* should all be static and not defined in the screen’s header.

4.2.1 interface.c

int register_screen(name, render, update, local)

Calculates the index of the screen within the *screens* hashmap, and sets the attributes of the screen structure at that index to the given arguments.

Returns the index of the screen in the hashmap.

name – The name of the screen, used as the key within the hashmap of screens

char *

render – A function pointer to the code which renders the screen to the TUI

void (*)(struct render_context *)

update – A function pointer to the code which updates the screen

void (*)(struct render_context *)

local – A function pointer for handling special cases

void (*)()

Special cases occur for events like navigation keys. When a key mapped to a function such as “up” is pressed, the “up” function is called. This function is then told to call into the local handler of the currently active screen with the code LOCAL_ARROW_UP.

int switch_to_screen(name)

Switches the *active_screen* pointer to point to the specified screen.

Returns the indedx of the screen in the hashmap.

name – The name of the screen to switch to

*char **

4.2.2 themes.c

This file manages the initialization of colors.

static void read_theme(file)

Lexes the theme file using the *cfg_lex* function.

Populates *custom_colors* table.

file – Open file to lex

FILE *

void register_custom_colors()

This function is invoked by *init_ncurses* in main.c to register all custom colors with ncurses.

struct cfg_token *configure_theme(token)

token - Head node of the current argument column

***struct cfg_token ***

5 Code Style