WinAuto

Generated by Doxygen 1.8.15

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 3

Data Structure Documentation

3.1 f_queue Struct Reference

`#include <f_queue.h>`

Data Fields

- short f_type

  Describes one of three available function types to be registered (thanks to this, we know which function call should be executed during playback phase)
- int f_args [2]

  Describes up to two function arguments to be inserted into function call during playback phase.
- struct f_queue * **next**
- struct f_queue * prev

  *Pointers to the next and previous **f_type** nodes.*

3.1.1 Detailed Description

The core, most significant structure of the program. **f_queue** is a doubly linked list that if needed shifts itself into cyclic doubly linked list. **f_queue** resembles queue-alike structure. Each **f_queue** node describes a function call that needs to be performed when playing the saved recording.

Warning

There are three types of function descriptions:

- Type 1: corresponding to cursor's position
- Type 2: corresponding to registered keystrokes (including mouse device)
- Type 3: corresponding to setting the registered duration in between mouse's movements and keystrokes

Each node has up to two function arguments.

The documentation for this struct was generated from the following file:

- f_queue.h

Chapter 4

File Documentation

4.1 f_queue.h File Reference

Data Structures

- struct f_queue

4.2 files.c File Reference

```
#include <stdio.h>
#include <f_queue.h>
#include <functions_queue.h>
#include <stdbool.h>
```

Functions

- void save_recording (struct f_queue *tail, char *file_name)
- bool load_recording (struct f_queue **head, struct f_queue **tail, char *file_name)

4.2.1 Function Documentation

4.2.1.1 load_recording()

```
bool load_recording (
            struct f_queue ** head,
            struct f_queue ** tail,
            char * file_name )
```

Function reads saved recording from the .txt file into the **f_queue** linked list. Function **validates correctness** of the .txt file that is being read. If the file for some reason is corrupted, an exception occurs. In such case, corrupted file cannot be handled by playing engine, so the **memory is freed** and user is sent back to main menu.

Parameters

head	pointer to pointer to the first **f_queue** node
tail	pointer to pointer to the last **f_queue** node
file_name	pointer to file name that is being read

Returns

true if file has been read successfully
false if file cannot be opened, does not exist or is corrupted

< Basic file validation.

< In case of exception.

< **goto** is not necessarily harmful when handling errors, it makes the code simple and can even increase readability.

4.2.1.2 save_recording()

```
void save_recording (
            struct f_queue * tail,
            char * file_name )
```

Function saves the recording into .txt file, allowing user to retrieve data from it later on (in order to play the saved recording).

Parameters

tail	pointer to the last **f_queue**
file_name	pointer to saved file's name

< The data is saved in <**int**> <**int**> <**int**> fashion.

4.3 files.h File Reference

Functions

- void save_recording (struct f_queue *tail, char *file_name)
- bool load_recording (struct f_queue **head, struct f_queue **tail, char *file_name)

4.3.1 Function Documentation

4.3.1.1 load_recording()

```
bool load_recording (
            struct f_queue ** head,
            struct f_queue ** tail,
            char * file_name )
```

Function reads saved recording from the .txt file into the **f_queue** linked list. Function **validates correctness** of the .txt file that is being read. If the file for some reason is corrupted, an exception occurs. In such case, corrupted file cannot be handled by playing engine, so the **memory is freed** and user is sent back to main menu.

Parameters

head	pointer to pointer to the first **f_queue** node
tail	pointer to pointer to the last **f_queue** node
file_name	pointer to file name that is being read

Returns

true if file has been read successfully
false if file cannot be opened, does not exist or is corrupted

< Basic file validation.

< In case of exception.

< **goto** is not necessarily harmful when handling errors, it makes the code simple and can even increase readability.

4.3.1.2 save_recording()

```
void save_recording (
            struct f_queue * tail,
            char * file_name )
```

Function saves the recording into .txt file, allowing user to retrieve data from it later on (in order to play the saved recording).

Parameters

tail	pointer to the last **f_queue**
file_name	pointer to saved file's name

< The data is saved in <**int**> <**int**> <**int**> fashion.

4.4 functions_queue.c File Reference

```
#include <stddef.h>
#include <stdlib.h>
```

```
#include <f_queue.h>
```

Functions

- void add_function (struct f_queue **head, struct f_queue **tail, const short f_type, const int arg1, const int arg2)
- void free_all_but_tail (struct f_queue **head)
- void free_tail (struct f_queue **head, struct f_queue **tail)
- void free_recording (struct f_queue **head, struct f_queue **tail)
- void make_queue_cyclic (struct f_queue *head, struct f_queue *tail)
- void unmake_queue_cyclic (struct f_queue *head, struct f_queue *tail)
- void trim_head (struct f_queue **head)
- void trim_list (struct f_queue **head)

4.4.1 Function Documentation

4.4.1.1 add_function()

```
void add_function (
            struct f_queue ** head,
            struct f_queue ** tail,
            const short f_type,
            const int arg1,
            const int arg2 )
```

Function adds new **description** of function-call at the start of **f_queue**.

Warning

Parameters

head	pointer to pointer to the first node
tail	pointer to pointer to the last node
f_type	describes inserted function's type: **1** - if function is of type describing cursor's position, **2** - if function is of type describing keystrokes, **3** - if function is of type describing durations in between function calls
arg1	the first argument for the described function
arg2	the second argument for the described function

4.4.1.2 free_all_but_tail()

```
void free_all_but_tail (
            struct f_queue ** head )
```

Function **frees** all nodes from **f_queue** **except** the tail without introducing any **'if'** statements.

Parameters

head	pointer to pointer to the first **f_queue** node

4.4.1.3 free_recording()

```
void free_recording (
            struct f_queue ** head,
            struct f_queue ** tail )
```

Function frees all nodes from f_queue, without introducing any excessive **'if'** statements. This function ties up **free_all_but_tail** and **free_tail** functions.

Parameters

head	pointer to pointer to the first **f_queue** node
tail	pointer to pointer to the last **f_queue** node

4.4.1.4 free_tail()

```
void free_tail (
            struct f_queue ** head,
            struct f_queue ** tail )
```

Function **frees** the tail that has not been freed by **free_all_but_tail()** function.

Parameters

head	pointer to pointer to the first **f_queue** node
tail	pointer to pointer to the last **f_queue** node

4.4.1.5 make_queue_cyclic()

```
void make_queue_cyclic (
            struct f_queue * head,
            struct f_queue * tail )
```

The function **sets the cyclic list property** onto the **f_queue** doubly linked list. The function ties up the head and the tail together.

Parameters

head	pointer to the first **f_queue** node
tail	pointer to the last **f_queue** node

4.4.1.6 trim_head()

```
void trim_head (
            struct f_queue ** head )
```

The function removes the first element from the **f_queue** doubly linked list. No margin cases need to be considered, as they will never occur.

Parameters

head	pointer to pointer to the first list element

4.4.1.7 trim_list()

```
void trim_list (
            struct f_queue ** head )
```

The function is a wrapper function around **trim_head** function, and removes **two** starting nodes from the **f_queue** doubly linked list. No margin cases need to be considered, as they will never occur.

Parameters

head	pointer to pointer to the first list element

4.4.1.8 unmake_queue_cyclic()

```
void unmake_queue_cyclic (
            struct f_queue * head,
            struct f_queue * tail )
```

The function removes the cyclic list property off of the **f_queue** doubly linked list. The head and tail are no longer tied up together.

Parameters

head	pointer to the first **f_queue** node
tail	pointer to the last **f_queue** node

4.5 functions_queue.h File Reference

Functions

- void add_function (struct f_queue **head, struct f_queue **tail, const short f_type, const int arg1, const int arg2)
- void free_all_but_tail (struct f_queue **head)
- void free_tail (struct f_queue **head, struct f_queue **tail)
- void free_recording (struct f_queue **head, struct f_queue **tail)
- void make_queue_cyclic (struct f_queue *head, struct f_queue *tail)
- void unmake_queue_cyclic (struct f_queue *head, struct f_queue *tail)
- void trim_head (struct f_queue **head)
- void trim_list (struct f_queue **head)

4.5.1 Function Documentation

4.5.1.1 add_function()

```
void add_function (
            struct f_queue ** head,
            struct f_queue ** tail,
            const short f_type,
            const int arg1,
            const int arg2 )
```

Function adds new **description** of function-call at the start of **f_queue**.

Warning

Parameters

head	pointer to pointer to the first node
tail	pointer to pointer to the last node
f_type	describes inserted function's type: **1** - if function is of type describing cursor's position, **2** - if function is of type describing keystrokes, **3** - if function is of type describing durations in between function calls
arg1	the first argument for the described function
arg2	the second argument for the described function

4.5.1.2 free_all_but_tail()

```
void free_all_but_tail (
            struct f_queue ** head )
```

Function **frees** all nodes from **f_queue** **except** the tail without introducing any **'if'** statements.

Parameters

head	pointer to pointer to the first **f_queue** node

4.5.1.3 free_recording()

```
void free_recording (
            struct f_queue ** head,
            struct f_queue ** tail )
```

Function frees all nodes from f_queue, without introducing any excessive **'if'** statements. This function ties up **free_all_but_tail** and **free_tail** functions.

Parameters

head	pointer to pointer to the first **f_queue** node
tail	pointer to pointer to the last **f_queue** node

4.5.1.4 free_tail()

```
void free_tail (
            struct f_queue ** head,
            struct f_queue ** tail )
```

Function **frees** the tail that has not been freed by **free_all_but_tail()** function.

Parameters

head	pointer to pointer to the first **f_queue** node
tail	pointer to pointer to the last **f_queue** node

4.5.1.5 make_queue_cyclic()

```
void make_queue_cyclic (
            struct f_queue * head,
            struct f_queue * tail )
```

The function **sets the cyclic list property** onto the **f_queue** doubly linked list. The function ties up the head and the tail together.

Parameters

head	pointer to the first **f_queue** node
tail	pointer to the last **f_queue** node

4.5.1.6 trim_head()

```
void trim_head (
            struct f_queue ** head )
```

The function removes the first element from the **f_queue** doubly linked list. No margin cases need to be considered, as they will never occur.

Parameters

head	pointer to pointer to the first list element

4.5.1.7 trim_list()

```
void trim_list (
            struct f_queue ** head )
```

The function is a wrapper function around **trim_head** function, and removes **two** starting nodes from the **f_queue** doubly linked list. No margin cases need to be considered, as they will never occur.

Parameters

head	pointer to pointer to the first list element

4.5.1.8 unmake_queue_cyclic()

```
void unmake_queue_cyclic (
            struct f_queue * head,
            struct f_queue * tail )
```

The function removes the cyclic list property off of the **f_queue** doubly linked list. The head and tail are no longer tied up together.

Parameters

head	pointer to the first **f_queue** node
tail	pointer to the last **f_queue** node

4.6 input_cursor.c File Reference

```
#include <windows.h>
```

Functions

- POINT get_cursor (void)

4.6.1 Function Documentation

4.6.1.1 get_cursor()

```
POINT get_cursor (
            void  )
```

Function saves current cursor's position into the **POINT** struct.

4.7 input_cursor.h File Reference

Functions

- POINT get_cursor (void)

4.7.1 Function Documentation

4.7.1.1 get_cursor()

```
POINT get_cursor (
            void  )
```

Function saves current cursor's position into the **POINT** struct.

4.8 key_codes.h File Reference

Macros

- #define **KEY_LMB** 1
- #define **KEY_RMB** 2
- #define **KEY_MMB** 4
- #define **KEY_BACK** 8
- #define **KEY_TAB** 9
- #define **KEY_RETURN** 13
- #define **KEY_SHIFT** 16
- #define **KEY_CTRL** 17
- #define **KEY_ALT** 18
- #define **KEY_CAPSLOCK** 20
- #define **KEY_ESC** 27
- #define **KEY_SPACE** 32
- #define **KEY_PGUP** 33
- #define **KEY_PGDN** 34
- #define **KEY_END** 35
- #define **KEY_HOME** 36
- #define **KEY_LEFT** 37
- #define **KEY_UP** 38
- #define **KEY_RIGHT** 39
- #define **KEY_DOWN** 40
- #define **KEY_PSCRN** 44
- #define **KEY_INS** 45
- #define **KEY_DEL** 46
- #define **KEY_0** 48
- #define **KEY_1** 49
- #define **KEY_2** 50
- #define **KEY_3** 51
- #define **KEY_4** 52
- #define **KEY_5** 53
- #define **KEY_6** 54
- #define **KEY_7** 55
- #define **KEY_8** 56
- #define **KEY_9** 57
- #define **KEY_A** 65
- #define **KEY_B** 66
- #define **KEY_C** 67
- #define **KEY_D** 68
- #define **KEY_E** 69
- #define **KEY_F** 70
- #define **KEY_G** 71
- #define **KEY_H** 72
- #define **KEY_I** 73
- #define **KEY_J** 74
- #define **KEY_K** 75
- #define **KEY_L** 76
- #define **KEY_M** 77
- #define **KEY_N** 78
- #define **KEY_O** 79
- #define **KEY_P** 80
- #define **KEY_Q** 81

- #define **KEY_R** 82
- #define **KEY_S** 83
- #define **KEY_T** 84
- #define **KEY_U** 85
- #define **KEY_V** 86
- #define **KEY_W** 87
- #define **KEY_X** 88
- #define **KEY_Y** 89
- #define **KEY_Z** 90
- #define **KEY_LWIN** 91
- #define **KEY_RWIN** 92
- #define **KEY_APPS** 93
- #define **KEY_NUMPAD0** 96
- #define **KEY_NUMPAD1** 97
- #define **KEY_NUMPAD2** 98
- #define **KEY_NUMPAD3** 99
- #define **KEY_NUMPAD4** 100
- #define **KEY_NUMPAD5** 101
- #define **KEY_NUMPAD6** 102
- #define **KEY_NUMPAD7** 103
- #define **KEY_NUMPAD8** 104
- #define **KEY_NUMPAD9** 105
- #define **KEY_MULTIPLY** 106
- #define **KEY_ADD** 107
- #define **KEY_SEPARATOR** 108
- #define **KEY_SUBTRACT** 109
- #define **KEY_DECIMAL** 110
- #define **KEY_DIVIDE** 111
- #define **KEY_F1** 112
- #define **KEY_F2** 113
- #define **KEY_F3** 114
- #define **KEY_F4** 115
- #define **KEY_F5** 116
- #define **KEY_F6** 117
- #define **KEY_F7** 118
- #define **KEY_F8** 119
- #define **KEY_F9** 120
- #define **KEY_F10** 121
- #define **KEY_F11** 122
- #define **KEY_F12** 123
- #define **KEY_NUMLCK** 144
- #define **KEY_SCLLCK** 145
- #define **KEY_SEMICLN** 186
- #define **KEY_PLUS** 187
- #define **KEY_COMMA** 188
- #define **KEY_MINUS** 189
- #define **KEY_DOT** 190
- #define **KEY_SLASH** 191
- #define **KEY_TILDE** 192
- #define **KEY_LBRACKET** 219
- #define **KEY_LINE** 220
- #define **KEY_RBRACKET** 221
- #define **KEY_QUOTE** 222

4.9 keys_pqueue.c File Reference

```
#include <key_codes.h>
```

Variables

- const unsigned short keys_pqueue [104]
- const int keys_pqueue_size = sizeof(keys_pqueue) / sizeof(keys_pqueue[0])

4.9.1 Variable Documentation

4.9.1.1 keys_pqueue

```
const unsigned short keys_pqueue[104]
```

Sorted from the most likely most often pressed key to least likely least often. The priority has been determined by nearly **2 000 000** registered keypresses throughout weeks of recording.

4.9.1.2 keys_pqueue_size

```
const int keys_pqueue_size = sizeof(keys_pqueue) / sizeof(keys_pqueue[0])
```

Determines the size of keys priority queue.

4.10 keys_pqueue.h File Reference

Variables

- const unsigned short keys_pqueue [104]
- const int keys_pqueue_size

4.10.1 Variable Documentation

4.10.1.1 keys_pqueue

```
const unsigned short keys_pqueue[104]
```

Represents priority queue for keys to be checked while recording process. Sorted from the most likely most often pressed key to least likely least often. The priority has been determined by nearly **2 000 000** registered keypresses throughout weeks of recording. In order to increase readibility, individual files have been introduced due to long array's definition.

Sorted from the most likely most often pressed key to least likely least often. The priority has been determined by nearly **2 000 000** registered keypresses throughout weeks of recording.

4.10.1.2 keys_pqueue_size

```
const int keys_pqueue_size
```

Determines the size of keys priority queue.

4.11 main.c File Reference

```
#include <stdio.h>
#include <time.h>
#include <windows.h>
#include <f_queue.h>
#include <functions_queue.h>
#include <recording.h>
#include <replay.h>
#include <menu.h>
#include <files.h>
#include "test/test.h"
```

Functions

- int **main** (int argc, char **argv)

4.12 menu.c File Reference

```
#include <stdio.h>
#include <key_codes.h>
#include <windows.h>
#include <f_queue.h>
#include <recording.h>
#include <files.h>
#include <replay.h>
#include <functions_queue.h>
#include <pressed_key.h>
#include <smooth_cursor.h>
```

Enumerations

- enum menu_flags {
  NO_ERRORS, ERROR_NO_TXT_SUFFIX, ERROR_READING_FILE, SAVED_HOTKEY,
  SAVED_FILE, STOPPED_PLAYBACK, STOPPED_SCREENSAVER, **HELP_SWITCH** }

Functions

- void print_help ()
- bool check_switches (int argc, char **argv)
- void draw_menu (const int flag_id)
- int get_menu_choice (void)
- int get_hotkey (void)
- bool str_ends_with (const char *source, const char *suffix)
- int get_cycles_num (void)
- void exec_play_recording (struct f_queue *head, struct f_queue *tail, const int cycles_num, const int hotkey_id)
- void init_menu (struct f_queue *head, struct f_queue *tail, const int flag_id, const int hotkey_id)

  prevents cyclic dependency
- void chosen_recording (struct f_queue *head, struct f_queue *tail, const int hotkey_id)
- void **chosen_playback** (struct f_queue *head, struct f_queue *tail, const int hotkey_id)

4.12.1 Enumeration Type Documentation

4.12.1.1 menu_flags

```
enum menu_flags
```

Enum containing various menu flags used to determine which **printf** should be displayed to the user, based on earlier program behaviour.

Enumerator

NO_ERRORS	start of definition default
ERROR_NO_TXT_SUFFIX	when user forgot to input the .txt postfix
ERROR_READING_FILE	when file was corrupted, does not exist or cannot be opened
SAVED_HOTKEY	when the hotkey has been successfully saved
SAVED_FILE	when the file saved successfully
STOPPED_PLAYBACK	when the recording playback successfully ended
STOPPED_SCREENSAVER	when the screensaver has been successfully stopped

4.12.2 Function Documentation

4.12.2.1 check_switches()

```
bool check_switches (
            int argc,
            char ** argv )
```

Function checks the command line input switches. If -h switch is found, detailed manual is printed out to the user.

4.12.2.2 chosen_recording()

```
void chosen_recording (
            struct f_queue * head,
            struct f_queue * tail,
            const int hotkey_id )
```

The function executes entire recording process when user chose **2**. Recording is stopped when **hotkey** is pressed and saved into the inputted .txt file. Hence it can be re-used afterwards for playback purposes. The function **recurseively** goes back to the menu with appropriate **menu_flags**: SAVED_FILE or ERROR_NO_TXT_SUFFIX, depending on the earlier behaviour.

Parameters

head	pointer to the front node of the **f_queue** linked list
tail	pointer to the last node of the **f_queue**</**b**< **linked list**
hotkey_id	

4.12.2.3 draw_menu()

```
void draw_menu (
            const int flag_id )
```

The function outputs relevant text data to the user. The function helps the user navigate around the program.

Parameters

flag_id	menu flag to determine expected printf result based on earlier behaviour

4.12.2.4 exec_play_recording()

```
void exec_play_recording (
            struct f_queue * head,
            struct f_queue * tail,
            const int cycles_num,
            const int hotkey_id )
```

The function executes the process of simulation of playing the recording. In case if cycles number is greater than 5, the playback loop is infinite. The playback loop ends at the end of all cycles, or **can be broken by pressing the set (or default if not set) hotkey**.

Parameters

head	pointer to the front of the **f_queue** list-queue
tail	pointer to the last node of the **f_queue** list-queue
cycles_num	the number of playback cycles
hotkey_id	the turn-off playback key switch

4.12.2.5 get_cycles_num()

```
int get_cycles_num (
            void  )
```

The function prompts user to input how many cycles of recording he wishes to playback. The input number has to be an integer greater or equal than 1, and if the input is greater than 5, then it is assumed the playback is infinitely loop. **In such case the f_queue doubly linked list-queue attains cyclic properties.**

Returns

cycles_num the desired number of cycles

4.12.2.6 get_hotkey()

```
int get_hotkey (
            void  )
```

The function saves user-inputted keystroke as a hotkey used in **2nd, 3rd and 4th** menu functions.

Warning

User needs to remember his hotkey.
For user's convenience, several hotkeys that would propably not me sense were blacklisted, including the default hotkey.

4.12.2.7 get_menu_choice()

```
int get_menu_choice (
            void  )
```

The function prompts user to select menu choice to futher navigate around the program. Basic input validation is performed.

4.12.2.8 init_menu()

```
void init_menu (
            struct f_queue * head,
            struct f_queue * tail,
            const int flag_id,
            const int hotkey_id )
```

prevents cyclic dependency

Recursive function that loops the menu and loops the execution of the program. The user chooses if he wants to set new hotkey, create new recording, playback old recording, start screensaver or end the program.

Parameters

head	pointer to the front node of **f_queue** doubly-linked list
tail	pointer to the last node of **f_queue** doubly-linked list
flag_id	the menu flag, depending on the value different output is displayed to the user
hotkey_id	the turn-off switch for the program (default **F5**)

default hotkey

4.12.2.9 print_help()

```
void print_help ( )
```

Function prints detailed manual to the user if -h flag was invoked.

4.12.2.10 str_ends_with()

```
bool str_ends_with (
            const char * source,
            const char * suffix )
```

The function verifies if string (array of chars) ends with given suffix (other array of chars). Used to validate if the file inputted by the user surely ends with .txt postfix.

Parameters

source	pointer to source array
suffix	pointer to desired ending suffix of soruce array

Returns

true if source ends with suffix
false otherwise

Warning

The function comes from stackoverflow.com

4.13 menu.h File Reference

Functions

- void draw_menu (const int flag_id)
- int get_menu_choice (void)
- int get_hotkey (void)
- bool str_ends_with (const char ∗source, const char ∗suffix)

- int get_cycles_num (void)
- void exec_play_recording (struct f_queue *head, struct f_queue *tail, const int cycles_num, const int hotkey↩ _id)
- void chosen_recording (struct f_queue *head, struct f_queue *tail, const int hotkey_id)
- void init_menu (struct f_queue *head, struct f_queue *tail, const int flag_id, const int hotkey_id)

  prevents cyclic dependency
- void print_help ()
- bool check_switches (int argc, char **argv)

4.13.1 Function Documentation

4.13.1.1 check_switches()

```
bool check_switches (
            int argc,
            char ** argv )
```

Function checks the command line input switches. If -h switch is found, detailed manual is printed out to the user.

4.13.1.2 chosen_recording()

```
void chosen_recording (
            struct f_queue * head,
            struct f_queue * tail,
            const int hotkey_id )
```

The function executes entire recording process when user chose **2**. Recording is stopped when **hotkey** is pressed and saved into the inputted .txt file. Hence it can be re-used afterwards for playback purposes. The function **recurseively** goes back to the menu with appropriate **menu_flags**: SAVED_FILE or ERROR_NO_TXT_SUFFIX, depending on the earlier behaviour.

Parameters

head	pointer to the front node of the **f_queue** linked list
tail	pointer to the last node of the **f_queue**</**b**< **linked list**
hotkey↩ _id	

4.13.1.3 draw_menu()

```
void draw_menu (
            const int flag_id )
```

The function outputs relevant text data to the user. The function helps the user navigate around the program.

Parameters

flag↩ _id	menu flag to determine expected printf result based on earlier behaviour

4.13.1.4 exec_play_recording()

```
void exec_play_recording (
              struct f_queue * head,
              struct f_queue * tail,
              const int cycles_num,
              const int hotkey_id )
```

The function executes the process of simulation of playing the recording. In case if cycles number is greater than 5, the playback loop is infinite. The playback loop ends at the end of all cycles, or **can be broken by pressing the set (or default if not set) hotkey**.

Parameters

head	pointer to the front of the **f_queue** list-queue
tail	pointer to the last node of the **f_queue** list-queue
cycles_num	the number of playback cycles
hotkey_id	the turn-off playback key switch

4.13.1.5 get_cycles_num()

```
int get_cycles_num (
              void  )
```

The function prompts user to input how many cycles of recording he wishes to playback. The input number has to be an integer greater or equal than 1, and if the input is greater than 5, then it is assumed the playback is infinitely loop. **In such case the f_queue doubly linked list-queue attains cyclic properties.**

Returns

cycles_num the desired number of cycles

4.13.1.6 get_hotkey()

```
int get_hotkey (
              void  )
```

The function saves user-inputted keystroke as a hotkey used in **2nd, 3rd and 4th** menu functions.

Warning

User needs to remember his hotkey.
For user's convenience, several hotkeys that would propably not me sense were blacklisted, including the default hotkey.

4.13.1.7 get_menu_choice()

```
int get_menu_choice (
            void  )
```

The function prompts user to select menu choice to futher navigate around the program. Basic input validation is performed.

4.13.1.8 init_menu()

```
void init_menu (
            struct f_queue * head,
            struct f_queue * tail,
            const int flag_id,
            const int hotkey_id )
```

prevents cyclic dependency

Recursive function that loops the menu and loops the execution of the program. The user chooses if he wants to set new hotkey, create new recording, playback old recording, start screensaver or end the program.

Parameters

head	pointer to the front node of **f_queue** doubly-linked list
tail	pointer to the last node of **f_queue** doubly-linked list
flag_id	the menu flag, depending on the value different output is displayed to the user
hotkey_id	the turn-off switch for the program (default **F5**)

default hotkey

4.13.1.9 print_help()

```
void print_help ( )
```

Function prints detailed manual to the user if -h flag was invoked.

4.13.1.10 str_ends_with()

```
bool str_ends_with (
            const char * source,
            const char * suffix )
```

The function verifies if string (array of chars) ends with given suffix (other array of chars). Used to validate if the file inputted by the user surely ends with .txt postfix.

Parameters

source	pointer to source array
suffix	pointer to desired ending suffix of soruce array

Returns

true if source ends with suffix
false otherwise

Warning

The function comes from stackoverflow.com

4.14 pressed_key.c File Reference

```
#include <windows.h>
#include <stdbool.h>
#include <keys_pqueue.h>
```

Functions

- bool check_key (const short key_id)
- void clr_system_buffer (void)
- short get_keystroke (void)

4.14.1 Function Documentation

4.14.1.1 check_key()

```
bool check_key (
            const short key_id )
```

Function checks if given key was being pressed (held) at the exact time of function call.

Returns

true if the key was being held
false otherwise

< If the most significant bit was set, the key was being held

4.14.1.2 clr_system_buffer()

```
void clr_system_buffer (
            void  )
```

Warning

Function clears GetAsyncKeyState's **'system'** buffer. Needed due to **GetAsyncKeyState** nature.

4.14.1.3 get_keystroke()

```
short get_keystroke (
            void  )
```

Returns integer value of key that was being pressed at the time of function call.

Warning

If two or more keys were pressed simultaneously, the key with the highest priority is returned.

< If no key was pressed

4.15 pressed_key.h File Reference

```
#include <stdbool.h>
```

Functions

- bool check_key (const short key_id)
- void clr_system_buffer (void)
- short get_keystroke (void)

4.15.1 Function Documentation

4.15.1.1 check_key()

```
bool check_key (
            const short key_id )
```

Function checks if given key was being pressed (held) at the exact time of function call.

Returns

true if the key was being held
false otherwise

< If the most significant bit was set, the key was being held

4.15.1.2 clr_system_buffer()

```
void clr_system_buffer (
            void  )
```

Warning

Function clears GetAsyncKeyState's **'system'** buffer. Needed due to **GetAsyncKeyState** nature.

4.15.1.3 get_keystroke()

```
short get_keystroke (
            void  )
```

Returns integer value of key that was being pressed at the time of function call.

Warning

If two or more keys were pressed simultaneously, the key with the highest priority is returned.

< If no key was pressed

4.16 recording.c File Reference

```
#include <stdbool.h>
#include <windows.h>
#include <f_queue.h>
#include <input_cursor.h>
#include <functions_queue.h>
#include <pressed_key.h>
#include <key_codes.h>
#include <stdio.h>
```

Macros

- #define **_GETCURSOR** 1
- #define **_GETKEY** 2
- #define **_SLEEP** 3

Functions

- void add_cursor (struct f_queue **head, struct f_queue **tail, POINT P[2])
- void add_keystroke (struct f_queue **head, struct f_queue **tail, int key_buff[2])
- bool is_prev_sleep_func (struct f_queue **head)
- void add_sleep (struct f_queue **head, struct f_queue **tail, const int sleep_dur)
- void record (struct f_queue **head, struct f_queue **tail, const int sleep_dur, const int hotkey_id)

4.16.1 Function Documentation

4.16.1.1 add_cursor()

```
void add_cursor (
            struct f_queue ** head,
            struct f_queue ** tail,
            POINT P[2] )
```

The function inserts current cursor's position as a description of a function call into the **f_queue**. The new node is inserted at the front of **f_queue**.

Parameters

head	pointer to pointer to the first node
tail	pointer to pointer to the last node
P[2]	array of two **POINT** structures. The array contains current cursor's position **(x and y, in pixels)**.

< if current cursor pos != previous

< add it to the queue

4.16.1.2 add_keystroke()

```
void add_keystroke (
            struct f_queue ** head,
            struct f_queue ** tail,
            int key_buff[2] )
```

The function inserts keystroke, if there was one, as a description of a function call into the **f_queue**. The new node is inserted at the front of **f_queue**.

Parameters

head	pointer to pointer to the first node
tail	pointer to pointer to the last node
key_buff[2]	array of two integers. The array is a buffer for current and previous keystroke.

< if there was keystroke

< add it to the queue

4.16.1.3 add_sleep()

```
void add_sleep (
            struct f_queue ** head,
```

```
            struct f_queue ** tail,
            const int sleep_dur )
```

The function inserts fixed 10 miliseconds duration as a description of a function call into the **f_queue**. The new node is inserted at the front of **f_queue**. If previous node was of sleep type as well, then rather than adding new node, they are **summed up together**.

Parameters

head	pointer to pointer to the first node
tail	pointer to pointer to the last node
sleep_dur	waiting interval between function calls, fixed to be 10 miliseconds

< increment the previous node, rather than add new one

4.16.1.4 is_prev_sleep_func()

```
bool is_prev_sleep_func (
            struct f_queue ** head )
```

Function verifies if newly added node was describing sleep function.

Parameters

head	pointer to pointer to the first **f_queue** node

Returns

true if previous node was describing **Sleep** type of function. In this case afterwards summation with previous node is performed, in order to safe space complexity.
false otherwise

4.16.1.5 record()

```
void record (
            struct f_queue ** head,
            struct f_queue ** tail,
            const int sleep_dur,
            const int hotkey_id )
```

Keyboard/Mouse recording engine. Ties up all recording functions together. The function inserts new function descriptions as nodes into the doubly linked list-queue.

Warning

Nodes are inserted in the following consecutive order: **cursor position ->** **keystroke ->** **waiting interval.** Nodes are inserted repeatedly until hotkey (turn-off switch) is pressed. In such case recording breaks and the recording is saved into the .txt file.

Parameters

head	pointer to pointer to the front of the **f_queue**
tail	pointer to pointer to the last node of the **f_queue**
sleep_dur	constant fixed waiting interval in between recording consecutive cursor movements/keystrokes, fixed to be 10 miliseconds
hotkey↩ _id	turn-off key switch

< buffer for curr and prev pressed key

< buffer for curr and prev cursor position

< stop recording when 'hotkey' is pressed

4.17 recording.h File Reference

```
#include <stdbool.h>
```

Functions

- void add_cursor (struct f_queue **head, struct f_queue **tail, POINT P[2])
- void add_keystroke (struct f_queue **head, struct f_queue **tail, int key_buff[2])
- bool is_prev_sleep_func (struct f_queue **head)
- void add_sleep (struct f_queue **head, struct f_queue **tail, const int sleep_dur)
- void record (struct f_queue **head, struct f_queue **tail, const int sleep_dur, const int hotkey_id)

4.17.1 Function Documentation

4.17.1.1 add_cursor()

```
void add_cursor (
            struct f_queue ** head,
            struct f_queue ** tail,
            POINT P[2] )
```

The function inserts current cursor's position as a description of a function call into the **f_queue**. The new node is inserted at the front of **f_queue**.

Parameters

head	pointer to pointer to the first node
tail	pointer to pointer to the last node
P[2]	array of two **POINT** structures. The array contains current cursor's position **(x and y, in pixels)**.

< if current cursor pos != previous

< add it to the queue

4.17.1.2 add_keystroke()

```
void add_keystroke (
            struct f_queue ** head,
            struct f_queue ** tail,
            int key_buff[2] )
```

The function inserts keystroke, if there was one, as a description of a function call into the **f_queue**. The new node is inserted at the front of **f_queue**.

Parameters

head	pointer to pointer to the first node
tail	pointer to pointer to the last node
key_buff[2]	array of two integers. The array is a buffer for current and previous keystroke.

< if there was keystroke

< add it to the queue

4.17.1.3 add_sleep()

```
void add_sleep (
            struct f_queue ** head,
            struct f_queue ** tail,
            const int sleep_dur )
```

The function inserts fixed 10 miliseconds duration as a description of a function call into the **f_queue**. The new node is inserted at the front of **f_queue**. If previous node was of sleep type as well, then rather than adding new node, they are **summed up together**.

Parameters

head	pointer to pointer to the first node
tail	pointer to pointer to the last node
sleep_dur	waiting interval between function calls, fixed to be 10 miliseconds

< increment the previous node, rather than add new one

4.17.1.4 is_prev_sleep_func()

```
bool is_prev_sleep_func (
            struct f_queue ** head )
```

Function verifies if newly added node was describing sleep function.

Parameters

head	pointer to pointer to the first **f_queue** node

Returns

true if previous node was describing **Sleep** type of function. In this case afterwards summation with previous node is performed, in order to safe space complexity.
false otherwise

4.17.1.5 record()

```
void record (
             struct f_queue ** head,
             struct f_queue ** tail,
             const int sleep_dur,
             const int hotkey_id )
```

Keyboard/Mouse recording engine. Ties up all recording functions together. The function inserts new function descriptions as nodes into the doubly linked list-queue.

Warning

Nodes are inserted in the following consecutive order: **cursor position ->** **keystroke ->** **waiting interval.** Nodes are inserted repeatedly until hotkey (turn-off switch) is pressed. In such case recording breaks and the recording is saved into the .txt file.

Parameters

head	pointer to pointer to the front of the **f_queue**
tail	pointer to pointer to the last node of the **f_queue**
sleep_dur	constant fixed waiting interval in between recording consecutive cursor movements/keystrokes, fixed to be 10 miliseconds
hotkey_id	turn-off key switch

< buffer for curr and prev pressed key

< buffer for curr and prev cursor position

< stop recording when 'hotkey' is pressed

4.18 replay.c File Reference

```
#include <windows.h>
#include <f_queue.h>
#include <pressed_key.h>
```

Macros

- #define **WINVER** 0x0500
- #define **_GETCURSOR** 1
- #define **_GETKEY** 2
- #define **_SLEEP** 3

Functions

- void send_input (const int KEY_CODE)
- void play_recording (struct f_queue *tail, const int hotkey_id)

4.18.1 Function Documentation

4.18.1.1 play_recording()

```
void play_recording (
            struct f_queue * tail,
            const int hotkey_id )
```

The recording replay (playback) simulation engine. The function simulates the recording saved in .txt file. The playback's precision is almost accurate. The variation between playback and recording's duration is less than **5%** and usually below **1%.** The variation is dependant on recording's keystrokes/mouse movements intensity, and user's CPU speed. Starting from the tail iterates through entire **f_queue** doubly linked list-queue. Executes functions corresponding to the given **f_type**. The cursor's position is set when **f_type** describes cursor (f_type is 1). The cursor's keystroke is simulated when **f_type** describes keystroke event (f_type is 2). The sleep function (waiting interval for next cursor's movement or keystroke) is performed when **f_type** describes Sleep function (f_type is 3).

Parameters

tail	pointer to the last node of **f_queue** doubly linked list-queue. The playback starts from the tail.
hotkey_id	turn-off playback key switch

< Simulates cursor's position

< Simulates keystroke

< Simulates waiting interval in between keystrokes and/or cursor movements

4.18.1.2 send_input()

```
void send_input (
            const int KEY_CODE )
```

Function sends keyboard or mouse input event during the recording playback phase. The event is chosen on retrieved **KEY_CODE**. The key codes of **2 or less** correspond to mouse event. Other key codes correspond to keyboard event. The events are performed at an instant of time.

Parameters

KEY_CODE	

mouse event

keyboard event

4.19 replay.h File Reference

Functions

- void send_input (const int KEY_CODE)
- void play_recording (struct f_queue ∗tail, const int hotkey_id)

4.19.1 Function Documentation

4.19.1.1 play_recording()

```
void play_recording (
            struct f_queue * tail,
            const int hotkey_id )
```

The recording replay (playback) simulation engine. The function simulates the recording saved in .txt file. The playback's precision is almost accurate. The variation between playback and recording's duration is less than **5%** and usually below **1%.** The variation is dependant on recording's keystrokes/mouse movements intensity, and user's CPU speed. Starting from the tail iterates through entire **f_queue** doubly linked list-queue. Executes functions corresponding to the given **f_type**. The cursor's position is set when **f_type** describes cursor (f_type is 1). The cursor's keystroke is simulated when **f_type** describes keystroke event (f_type is 2). The sleep function (waiting interval for next cursor's movement or keystroke) is performed when **f_type** describes Sleep function (f_type is 3).

Parameters

tail	pointer to the last node of **f_queue** doubly linked list-queue. The playback starts from the tail.
hotkey↩ _id	turn-off playback key switch

< Simulates cursor's position

< Simulates keystroke

< Simulates waiting interval in between keystrokes and/or cursor movements

4.19.1.2 send_input()

```
void send_input (
            const int KEY_CODE )
```

Function sends keyboard or mouse input event during the recording playback phase. The event is chosen on retrieved **KEY_CODE**. The key codes of **2 or less** correspond to mouse event. Other key codes correspond to keyboard event. The events are performed at an instant of time.

Parameters

KEY_CODE	

mouse event

keyboard event

4.20 smooth_cursor.c File Reference

```
#include <windows.h>
#include <time.h>
#include <stdio.h>
#include <pressed_key.h>
#include <smooth_cursor.h>
```

Functions

- void move_cursor (const int x1, const int y1, const int x2, const int y2, const int duration)
- void direction_RD (float ∗x1, float ∗y1, float x_jump, float y_jump, float sleep_delay)
- void direction_RU (float ∗x1, float ∗y1, float x_jump, float y_jump, float sleep_delay)
- void direction_LD (float ∗x1, float ∗y1, float x_jump, float y_jump, float sleep_delay)
- void direction_LU (float ∗x1, float ∗y1, float x_jump, float y_jump, float sleep_delay)
- void smooth_transition (void(∗direction)(float ∗, float ∗, const float, const float, const float), float ∗x1, float ∗y1, const float x_jump, const float y_jump, const float sleep_delay, short num_of_jumps, const short x2, const short y2, const short duration)
- short smooth_cursor_fps (float x1, float y1, const short x2, const short y2, const short duration, const short fps)
- int get_input (const int MIN, const int MAX)
- void wrapper_get_input (int ∗const speed, int ∗const min_fps)
- void exec_screen_saver (const int hotkey_id)
- void screen_saver (int x2, int y2, const int screen_width, const int screen_height, const int speed, const int min_fps, const int hotkey_id)

4.20.1 Function Documentation

4.20.1.1 direction_LD()

```
void direction_LD (
            float * x1,
            float * y1,
            float x_jump,
            float y_jump,
            float sleep_delay )
```

Translates current cursor position along **Left-Down (LD)** directional vector. Sets the cursor position into the computed position.

Parameters

x1	pointer to **x** coordinate
y1	pointer to **y** coordinate
x_jump	jump to be removed from **x1** coordinate
y_jump	jump to be added to **y1** coordinate
sleep_delay	sleep delay in between consecutive cursor jumps (float is intended)

4.20.1.2 direction_LU()

```
void direction_LU (
            float * x1,
            float * y1,
            float x_jump,
            float y_jump,
            float sleep_delay )
```

Translates current cursor position along **Left-Up (LU)** directional vector. Sets the cursor position into the computed position.

Parameters

x1	pointer to **x** coordinate
y1	pointer to **y** coordinate
x_jump	jump to be removed to **x1** coordinate
y_jump	jump to be removed from **y1** coordinate
sleep_delay	sleep delay in between consecutive cursor jumps (float is intended)

4.20.1.3 direction_RD()

```
void direction_RD (
            float * x1,
            float * y1,
            float x_jump,
            float y_jump,
            float sleep_delay )
```

Translates current cursor position along **Right-Down (RD)** directional vector. Sets the cursor position into the computed position.

Parameters

x1	pointer to **x** coordinate
y1	pointer to **y** coordinate
x_jump	jump to be added to **x1** coordinate
y_jump	jump to be added to **y1** coordinate
sleep_delay	sleep delay in between consecutive cursor jumps (float is intended)

4.20.1.4 direction_RU()

```
void direction_RU (
            float * x1,
            float * y1,
            float x_jump,
            float y_jump,
            float sleep_delay )
```

Translates current cursor position along **Right-Up (RU)** directional vector. Sets the cursor position into the computed position.

Parameters

x1	pointer to **x** coordinate
y1	pointer to **y** coordinate
x_jump	jump to be added to **x1** coordinate
y_jump	jump to be removed from **y1** coordinate
sleep_delay	sleep delay in between consecutive cursor jumps (float is intended)

4.20.1.5 exec_screen_saver()

```
void exec_screen_saver (
            const int hotkey_id )
```

The function prepares **screensaving** cursor animation for execution. The function checks the user's monitor resolution and matches it accordingly.

Parameters

hotkey_id	toggle off key

4.20.1.6 get_input()

```
int get_input (
            const int MIN,
            const int MAX )
```

Wrapper function for **scanf** further usage. The input request occurs in a loop until proper data value is entered by the user.

Parameters

MIN	defines lower bound of the correct data set
MAX	defines upper bound of the correct data set

4.20.1.7 move_cursor()

```
void move_cursor (
            const int x1,
            const int y1,
            const int x2,
            const int y2,
            const int duration )
```

This neat function is currently not in use, however might be used in the future.

4.20.1.8 screen_saver()

```
void screen_saver (
            int x2,
            int y2,
            const int screen_width,
            const int screen_height,
            const int speed,
            const int min_fps,
            const int hotkey_id )
```

The function performs looped **screensaving** cursor animation. The animation is broken when **hotkey is HELD**. The animation can have various speeds and various smoothness (FPS) feelings, depending on user's entered values in **wrapper_get_input** function.

Warning

This is a recursive function and its stop condition is pressing the hotkey.

Parameters

x2	previous x cursor's position
y2	previous y cursor's position
screen_width	
screen_height	
speed	speed of the cursor
min_fps	minimal FPS of the animation
hotkey_id	the toggle-off hotkey's ID

< stop condition: screensaver ends when hotkey is HELD

< The next x cursor's position is computed with random function.

< The next y cursor's position is computed with random function.

< Duration is also computed with random function, and depends on cursor's speed.

< FPS is also computed with random function, and depends on min_fps parameter. (smoothness parameter)

< The key has to be held in order to stop the process, not instantenously pressed.

< recursive call

4.20.1.9 smooth_cursor_fps()

```
short smooth_cursor_fps (
            float x1,
            float y1,
            const short x2,
            const short y2,
            const short duration,
            const short fps )
```

The function computes data for accurate smooth cursor translation and then sends it to the main cursor transition function.

Parameters

x1	initial cursor's position on x (float is intended)
y1	intitial cursor's position on y axis (float is intended)
x2	final cursor's x position
y2	final cursor's y position
duration	duration of the transition
fps	FPS of the translation. **Custom feature that imitates smooth, less smooth or lagging cursor depending on the value.**

< absolute value: total distance the cursor has to travel along x axis cannot be negative

< absolute value: total distance the cursor has to travel along y axis cannot be negative

4.20.1.10 smooth_transition()

```
void smooth_transition (
            void(*)(float *, float *, const float, const float, const float) direction,
            float * x1,
            float * y1,
            const float x_jump,
            const float y_jump,
            const float sleep_delay,
            short num_of_jumps,
            const short x2,
            const short y2,
            const short duration )
```

The function accurately moves the cursor from initial to final position in a smooth fashion.

Parameters

direction	pointer to function returing void (the function takes directional vector as a parameters)
x1	pointer to x1 coordinate
x2	pointer to y1 coordinate
x_jump	the jump of cursor along x axis, jump between two consecutive cursor moves
y_jump	the jump of cursor along y axis, jump between two consecutive cursor moves
sleep_delay	waiting duration in between cursor jumps
num_of_jumps	total number of jumps the cursor does during the entire cursor transition
x2	final x position
y2	final y position
duration	duration of the entire smooth cursor translation process

< rarely final position's fix is needed, in case if cursor is off by 1 pixel

< transition is not 100% accurate in time (usually +- 5%), so if transition took less than expected, wait

4.20.1.11 wrapper_get_input()

```
void wrapper_get_input (
            int *const speed,
            int *const min_fps )
```

Wrapper function for get_input. The function prompts user to enter speed of the cursor and minimal value of FPS for the **screensaver** animation.

Parameters

speed	speed of the cursor, values from 1 to 10
min_fps	minimal fps of the screensaving animation's cursor's movement. Values from 1 to 99.

< The value has to be inverted in order to make "logical sense".

4.21 smooth_cursor.h File Reference

Functions

- void move_cursor (const int x1, const int y1, const int x2, const int y2, const int duration)
- void direction_RD (float *x1, float *y1, float x_jump, float y_jump, float sleep_delay)
- void direction_RU (float *x1, float *y1, float x_jump, float y_jump, float sleep_delay)
- void direction_LD (float *x1, float *y1, float x_jump, float y_jump, float sleep_delay)
- void direction_LU (float *x1, float *y1, float x_jump, float y_jump, float sleep_delay)
- void smooth_transition (void(*direction)(float *, float *, const float, const float, const float), float *x1, float *y1, const float x_jump, const float y_jump, const float sleep_delay, short num_of_jumps, const short x2, const short y2, const short duration)
- short smooth_cursor_fps (float x1, float y1, const short x2, const short y2, const short duration, const short fps)
- int get_input (const int MIN, const int MAX)

- void wrapper_get_input (int ∗const speed, int ∗const min_fps)
- void exec_screen_saver (const int hotkey_id)
- void screen_saver (int x2, int y2, const int screen_width, const int screen_height, const int speed, const int min_fps, const int hotkey_id)

4.21.1 Function Documentation

4.21.1.1 direction_LD()

```
void direction_LD (
            float * x1,
            float * y1,
            float x_jump,
            float y_jump,
            float sleep_delay )
```

Translates current cursor position along **Left-Down (LD)** directional vector. Sets the cursor position into the computed position.

Parameters

x1	pointer to **x** coordinate
y1	pointer to **y** coordinate
x_jump	jump to be removed from **x1** coordinate
y_jump	jump to be added to **y1** coordinate
sleep_delay	sleep delay in between consecutive cursor jumps (float is intended)

4.21.1.2 direction_LU()

```
void direction_LU (
            float * x1,
            float * y1,
            float x_jump,
            float y_jump,
            float sleep_delay )
```

Translates current cursor position along **Left-Up (LU)** directional vector. Sets the cursor position into the computed position.

Parameters

x1	pointer to **x** coordinate
y1	pointer to **y** coordinate
x_jump	jump to be removed to **x1** coordinate
y_jump	jump to be removed from **y1** coordinate
sleep_delay	sleep delay in between consecutive cursor jumps (float is intended)

4.21.1.3 direction_RD()

```
void direction_RD (
            float * x1,
            float * y1,
            float x_jump,
            float y_jump,
            float sleep_delay )
```

Translates current cursor position along **Right-Down (RD)** directional vector. Sets the cursor position into the computed position.

Parameters

x1	pointer to **x** coordinate
y1	pointer to **y** coordinate
x_jump	jump to be added to **x1** coordinate
y_jump	jump to be added to **y1** coordinate
sleep_delay	sleep delay in between consecutive cursor jumps (float is intended)

4.21.1.4 direction_RU()

```
void direction_RU (
            float * x1,
            float * y1,
            float x_jump,
            float y_jump,
            float sleep_delay )
```

Translates current cursor position along **Right-Up (RU)** directional vector. Sets the cursor position into the computed position.

Parameters

x1	pointer to **x** coordinate
y1	pointer to **y** coordinate
x_jump	jump to be added to **x1** coordinate
y_jump	jump to be removed from **y1** coordinate
sleep_delay	sleep delay in between consecutive cursor jumps (float is intended)

4.21.1.5 exec_screen_saver()

```
void exec_screen_saver (
            const int hotkey_id )
```

The function prepares **screensaving** cursor animation for execution. The function checks the user's monitor resolution and matches it accordingly.

Parameters

hotkey_id	toggle off key

4.21.1.6 get_input()

```
int get_input (
            const int MIN,
            const int MAX )
```

Wrapper function for **scanf** further usage. The input request occurs in a loop until proper data value is entered by the user.

Parameters

MIN	defines lower bound of the correct data set
MAX	defines upper bound of the correct data set

4.21.1.7 move_cursor()

```
void move_cursor (
            const int x1,
            const int y1,
            const int x2,
            const int y2,
            const int duration )
```

This neat function is currently not in use, however might be used in the future.

4.21.1.8 screen_saver()

```
void screen_saver (
            int x2,
            int y2,
            const int screen_width,
            const int screen_height,
            const int speed,
            const int min_fps,
            const int hotkey_id )
```

The function performs looped **screensaving** cursor animation. The animation is broken when **hotkey is HELD**. The animation can have various speeds and various smoothness (FPS) feelings, depending on user's entered values in **wrapper_get_input** function.

Warning

This is a recursive function and its stop condition is pressing the hotkey.

Parameters

x2	previous x cursor's position
y2	previous y cursor's position
screen_width	
screen_height	
speed	speed of the cursor
min_fps	minimal FPS of the animation
hotkey_id	the toggle-off hotkey's ID

< stop condition: screensaver ends when hotkey is HELD

< The next x cursor's position is computed with random function.

< The next y cursor's position is computed with random function.

< Duration is also computed with random function, and depends on cursor's speed.

< FPS is also computed with random function, and depends on min_fps parameter. (smoothness parameter)

< The key has to be held in order to stop the process, not instantenously pressed.

< recursive call

4.21.1.9 smooth_cursor_fps()

```
short smooth_cursor_fps (
             float x1,
             float y1,
             const short x2,
             const short y2,
             const short duration,
             const short fps )
```

The function computes data for accurate smooth cursor translation and then sends it to the main cursor transition function.

Parameters

x1	initial cursor's position on x (float is intended)
y1	intitial cursor's position on y axis (float is intended)
x2	final cursor's x position
y2	final cursor's y position
duration	duration of the transition
fps	FPS of the translation. **Custom feature that imitates smooth, less smooth or lagging cursor depending on the value.**

< absolute value: total distance the cursor has to travel along x axis cannot be negative

< absolute value: total distance the cursor has to travel along y axis cannot be negative

4.21.1.10 smooth_transition()

```
void smooth_transition (
             void(*)(float *, float *, const float, const float, const float) direction,
             float * x1,
             float * y1,
             const float x_jump,
             const float y_jump,
             const float sleep_delay,
             short num_of_jumps,
             const short x2,
             const short y2,
             const short duration )
```

The function accurately moves the cursor from initial to final position in a smooth fashion.

Parameters

direction	pointer to function returing void (the function takes directional vector as a parameters)
x1	pointer to x1 coordinate
x2	pointer to y1 coordinate
x_jump	the jump of cursor along x axis, jump between two consecutive cursor moves
y_jump	the jump of cursor along y axis, jump between two consecutive cursor moves
sleep_delay	waiting duration in between cursor jumps
num_of_jumps	total number of jumps the cursor does during the entire cursor transition
x2	final x position
y2	final y position
duration	duration of the entire smooth cursor translation process

< rarely final position's fix is needed, in case if cursor is off by 1 pixel

< transition is not 100% accurate in time (usually +- 5%), so if transition took less than expected, wait

4.21.1.11 wrapper_get_input()

```
void wrapper_get_input (
             int *const speed,
             int *const min_fps )
```

Wrapper function for get_input. The function prompts user to enter speed of the cursor and minimal value of FPS for the **screensaver** animation.

Parameters

speed	speed of the cursor, values from 1 to 10
min_fps	minimal fps of the screensaving animation's cursor's movement. Values from 1 to 99.

< The value has to be inverted in order to make "logical sense".

4.22 test/test.c File Reference

```
#include <stdio.h>
#include <windows.h>
#include <../pressed_key.h>
#include <../keys_pqueue.h>
#include <../f_queue.h>
```

Functions

- void __test_keystrokes (const int how_many, const int sleep_dur)
- void __test_print_pqueue (void)
- void __test_print_f_queue (struct f_queue *head)
- void __test_print_f_queue_back (struct f_queue *tail)

4.22.1 Function Documentation

4.22.1.1 __test_keystrokes()

```
void __test_keystrokes (
            const int how_many,
            const int sleep_dur )
```

Testing purposes, no longer needed. Could be useful in the future.

Parameters

how_many	number of key_presses to be checked @sleep_dur waiting interval

4.22.1.2 __test_print_f_queue()

```
void __test_print_f_queue (
            struct f_queue * head )
```

The function prints the **f_queue** forwards. No longer used, however could be needed for future testing/debugging purposes.

Parameters

head	pointer to the first node of the **f_queue**

4.22.1.3 __test_print_f_queue_back()

```
void __test_print_f_queue_back (
            struct f_queue * tail )
```

The function prints the **f_queue** backwards. No longer used, however could be needed for future testing/debugging purposes.

Parameters

tail	pointer to the last node of the **f_queue**

4.22.1.4 __test_print_pqueue()

```
void __test_print_pqueue (
            void  )
```

The function prints the priority queue of keys to be checked during recording process. No longer used, however could be needed for future testing/debugging purposes.

4.23 test/test.h File Reference

Functions

- void __test_keystrokes (const int how_many, const int sleep_dur)
- void __test_print_pqueue (void)
- void __test_print_f_queue (struct f_queue *head)
- void __test_print_f_queue_back (struct f_queue *tail)

4.23.1 Function Documentation

4.23.1.1 __test_keystrokes()

```
void __test_keystrokes (
            const int how_many,
            const int sleep_dur )
```

Testing purposes, no longer needed. Could be useful in the future.

Parameters

how_many	number of key_presses to be checked @sleep_dur waiting interval

4.23.1.2 __test_print_f_queue()

```
void __test_print_f_queue (
            struct f_queue * head )
```

The function prints the **f_queue** forwards. No longer used, however could be needed for future testing/debugging purposes.

Parameters

head	pointer to the first node of the **f_queue**

4.23.1.3 __test_print_f_queue_back()

```
void __test_print_f_queue_back (
            struct f_queue * tail )
```

The function prints the **f_queue** backwards. No longer used, however could be needed for future testing/debugging purposes.

Parameters

tail	pointer to the last node of the **f_queue**

4.23.1.4 __test_print_pqueue()

```
void __test_print_pqueue (
            void  )
```

The function prints the priority queue of keys to be checked during recording process. No longer used, however could be needed for future testing/debugging purposes.