基于RK3568的LVGL图形与事件驱动软件

```
/**
 * @file lv_drv_conf.h
 * Configuration file for v8.2.0
 */

/*
 * COPY THIS FILE AS lv_drv_conf.h
 */

/* clang-format off */
#if 1 /*Set it to "1" to enable the content*/

#ifndef LV_DRV_CONF_H
#define LV_DRV_CONF_H

#include "lv_conf.h"

/*********************
 * DELAY INTERFACE
 *********************/
#define LV_DRV_DELAY_INCLUDE  <stdint.h>            /*Dummy include by default*/
#define LV_DRV_DELAY_US(us)  /*delay_us(us)*/       /*Delay the given number of
microseconds*/
#define LV_DRV_DELAY_MS(ms)  /*delay_ms(ms)*/       /*Delay the given number of
milliseconds*/

/*********************
 * DISPLAY INTERFACE
 *********************/

/*------------
 *  Common
 *------------*/
#define LV_DRV_DISP_INCLUDE         <stdint.h>           /*Dummy include by
default*/
#define LV_DRV_DISP_CMD_DATA(val)  /*pin_x_set(val)*/    /*Set the command/data
pin to 'val'*/
#define LV_DRV_DISP_RST(val)       /*pin_x_set(val)*/    /*Set the reset pin to
'val'*/

/*---------
 *  SPI
 *---------*/
#define LV_DRV_DISP_SPI_CS(val)          /*spi_cs_set(val)*/     /*Set the SPI's
Chip select to 'val'*/
#define LV_DRV_DISP_SPI_WR_BYTE(data)    /*spi_wr(data)*/        /*Write a byte
the SPI bus*/
#define LV_DRV_DISP_SPI_WR_ARRAY(adr, n) /*spi_wr_mem(adr, n)*/  /*Write 'n'
bytes to SPI bus from 'adr'*/

/*------------------
 *  Parallel port
```

```
 *-----------------*/
#define LV_DRV_DISP_PAR_CS(val)          /*par_cs_set(val)*/   /*Set the Parallel
port's Chip select to 'val'*/
#define LV_DRV_DISP_PAR_SLOW             /*par_slow()*/        /*Set low speed on
the parallel port*/
#define LV_DRV_DISP_PAR_FAST             /*par_fast()*/        /*Set high speed
on the parallel port*/
#define LV_DRV_DISP_PAR_WR_WORD(data)    /*par_wr(data)*/      /*Write a word to
the parallel port*/
#define LV_DRV_DISP_PAR_WR_ARRAY(adr, n) /*par_wr_mem(adr,n)*/ /*Write 'n' bytes
to Parallel ports from 'adr'*/

/***************************
 * INPUT DEVICE INTERFACE
 ***************************/

/*----------
 *  Common
 *----------*/
#define LV_DRV_INDEV_INCLUDE     <stdint.h>             /*Dummy include by
default*/
#define LV_DRV_INDEV_RST(val)    /*pin_x_set(val)*/     /*Set the reset pin to
'val'*/
#define LV_DRV_INDEV_IRQ_READ    0 /*pn_x_read()*/      /*Read the IRQ pin*/

/*---------
 *  SPI
 *---------*/
#define LV_DRV_INDEV_SPI_CS(val)            /*spi_cs_set(val)*/     /*Set the
SPI's Chip select to 'val'*/
#define LV_DRV_INDEV_SPI_XCHG_BYTE(data)    0 /*spi_xchg(val)*/     /*Write 'val'
to SPI and give the read value*/

/*---------
 *  I2C
 *---------*/
#define LV_DRV_INDEV_I2C_START              /*i2c_start()*/       /*Make an I2C
start*/
#define LV_DRV_INDEV_I2C_STOP               /*i2c_stop()*/        /*Make an I2C
stop*/
#define LV_DRV_INDEV_I2C_RESTART            /*i2c_restart()*/     /*Make an I2C
restart*/
#define LV_DRV_INDEV_I2C_WR(data)           /*i2c_wr(data)*/      /*Write a byte
to the I1C bus*/
#define LV_DRV_INDEV_I2C_READ(last_read)    0 /*i2c_rd()*/        /*Read a byte
from the I2C bud*/


/*********************
 *  DISPLAY DRIVERS
 *********************/

/*-------------------
 *  SDL
 *-------------------*/
```

```
/* SDL based drivers for display, mouse, mousewheel and keyboard*/
#ifndef USE_SDL
# define USE_SDL 0
#endif

/* Hardware accelerated SDL driver */
#ifndef USE_SDL_GPU
# define USE_SDL_GPU 0
#endif

#if USE_SDL || USE_SDL_GPU
#  define SDL_HOR_RES     480
#  define SDL_VER_RES     320

/* Scale window by this factor (useful when simulating small screens) */
#  define SDL_ZOOM        1

/* Used to test true double buffering with only address changing.
 * Use 2 draw buffers, bith with SDL_HOR_RES x SDL_VER_RES size*/
#  define SDL_DOUBLE_BUFFERED 0

/*Eclipse: <SDL2/SDL.h>    Visual Studio: <SDL.h>*/
#  define SDL_INCLUDE_PATH    <SDL2/SDL.h>

/*Open two windows to test multi display support*/
#  define SDL_DUAL_DISPLAY            0
#endif

/*--------------------
 *  Monitor of PC
 *--------------------*/

/*DEPRECATED: Use the SDL driver instead. */
#ifndef USE_MONITOR
#  define USE_MONITOR         0
#endif

#if USE_MONITOR
#  define MONITOR_HOR_RES     480
#  define MONITOR_VER_RES     320

/* Scale window by this factor (useful when simulating small screens) */
#  define MONITOR_ZOOM        1

/* Used to test true double buffering with only address changing.
 * Use 2 draw buffers, bith with MONITOR_HOR_RES x MONITOR_VER_RES size*/
#  define MONITOR_DOUBLE_BUFFERED 0

/*Eclipse: <SDL2/SDL.h>    Visual Studio: <SDL.h>*/
#  define MONITOR_SDL_INCLUDE_PATH    <SDL2/SDL.h>

/*Open two windows to test multi display support*/
#  define MONITOR_DUAL            0
#endif

/*----------------------------------
```

```
 *  Native Windows (including mouse)
 *-----------------------------------*/
#ifndef USE_WINDOWS
#  define USE_WINDOWS       0
#endif

#if USE_WINDOWS
#  define WINDOW_HOR_RES      480
#  define WINDOW_VER_RES      320
#endif

/*----------------------------
 *  Native Windows (win32drv)
 *----------------------------*/
#ifndef USE_WIN32DRV
#  define USE_WIN32DRV       0
#endif

#if USE_WIN32DRV
/* Scale window by this factor (useful when simulating small screens) */
#  define WIN32DRV_MONITOR_ZOOM        1
#endif

/*----------------------------------------
 *  GTK drivers (monitor, mouse, keyboard
 *----------------------------------------*/
#ifndef USE_GTK
#  define USE_GTK       0
#endif

/*----------------------------------------
 *  Wayland drivers (monitor, mouse, keyboard, touchscreen)
 *----------------------------------------*/
#ifndef USE_WAYLAND
#  define USE_WAYLAND       0
#endif

#if USE_WAYLAND
/* Support for client-side decorations */
#  ifndef LV_WAYLAND_CLIENT_SIDE_DECORATIONS
#    define LV_WAYLAND_CLIENT_SIDE_DECORATIONS 1
#  endif
/* Support for (deprecated) wl-shell protocol */
#  ifndef LV_WAYLAND_WL_SHELL
#    define LV_WAYLAND_WL_SHELL 1
#  endif
/* Support for xdg-shell protocol */
#  ifndef LV_WAYLAND_XDG_SHELL
#    define LV_WAYLAND_XDG_SHELL 0
#  endif
#endif

/*----------------
 *    SSD1963
 *--------------*/
#ifndef USE_SSD1963
```

```
#  define USE_SSD1963         0
#endif

#if USE_SSD1963
#  define SSD1963_HOR_RES     LV_HOR_RES
#  define SSD1963_VER_RES     LV_VER_RES
#  define SSD1963_HT          531
#  define SSD1963_HPS         43
#  define SSD1963_LPS         8
#  define SSD1963_HPW         10
#  define SSD1963_VT          288
#  define SSD1963_VPS         12
#  define SSD1963_FPS         4
#  define SSD1963_VPW         10
#  define SSD1963_HS_NEG      0   /*Negative hsync*/
#  define SSD1963_VS_NEG      0   /*Negative vsync*/
#  define SSD1963_ORI         0   /*0, 90, 180, 270*/
#  define SSD1963_COLOR_DEPTH 16
#endif

/*----------------
 *    R61581
 *--------------*/
#ifndef USE_R61581
#  define USE_R61581          0
#endif

#if USE_R61581
#  define R61581_HOR_RES      LV_HOR_RES
#  define R61581_VER_RES      LV_VER_RES
#  define R61581_HSPL         0       /*HSYNC signal polarity*/
#  define R61581_HSL          10      /*HSYNC length (Not Implemented)*/
#  define R61581_HFP          10      /*Horitontal Front poarch (Not
Implemented)*/
#  define R61581_HBP          10      /*Horitontal Back poarch (Not Implemented
*/
#  define R61581_VSPL         0       /*VSYNC signal polarity*/
#  define R61581_VSL          10      /*VSYNC length (Not Implemented)*/
#  define R61581_VFP          8       /*Vertical Front poarch*/
#  define R61581_VBP          8       /*Vertical Back poarch */
#  define R61581_DPL          0       /*DCLK signal polarity*/
#  define R61581_EPL          1       /*ENABLE signal polarity*/
#  define R61581_ORI          0       /*0, 180*/
#  define R61581_LV_COLOR_DEPTH 16    /*Fix 16 bit*/
#endif

/*-----------------------------
 *  ST7565 (Monochrome, low res.)
 *----------------------------*/
#ifndef USE_ST7565
#  define USE_ST7565          0
#endif

#if USE_ST7565
/*No settings*/
#endif  /*USE_ST7565*/
```

```
/*-----------------------------
 *  GC9A01 (color, low res.)
 *-----------------------------*/
#ifndef USE_GC9A01
#  define USE_GC9A01          0
#endif

#if USE_GC9A01
/*No settings*/
#endif  /*USE_GC9A01*/

/*-----------------------------------------
 *  UC1610 (4 gray 160*[104|128])
 *  (EA DOGXL160 160x104 tested)
 *-----------------------------------------*/
#ifndef USE_UC1610
#  define USE_UC1610          0
#endif

#if USE_UC1610
#  define UC1610_HOR_RES         LV_HOR_RES
#  define UC1610_VER_RES         LV_VER_RES
#  define UC1610_INIT_CONTRAST   33   /* init contrast, values in [%] */
#  define UC1610_INIT_HARD_RST   0    /* 1 : hardware reset at init, 0 : software
reset */
#  define UC1610_TOP_VIEW        0    /* 0 : Bottom View, 1 : Top View */
#endif  /*USE_UC1610*/

/*-------------------------------------------------
 *  SHARP memory in pixel monochrome display series
 *      LS012B7DD01 (184x38  pixels.)
 *      LS013B7DH03 (128x128 pixels.)
 *      LS013B7DH05 (144x168 pixels.)
 *      LS027B7DH01 (400x240 pixels.) (tested)
 *      LS032B7DD02 (336x536 pixels.)
 *      LS044Q7DH01 (320x240 pixels.)
 *-------------------------------------------------*/
#ifndef USE_SHARP_MIP
#  define USE_SHARP_MIP       0
#endif

#if USE_SHARP_MIP
#  define SHARP_MIP_HOR_RES             LV_HOR_RES
#  define SHARP_MIP_VER_RES             LV_VER_RES
#  define SHARP_MIP_SOFT_COM_INVERSION  0
#  define SHARP_MIP_REV_BYTE(b)         /*((uint8_t) __REV(__RBIT(b)))*/
 /*Architecture / compiler dependent byte bits order reverse*/
#endif  /*USE_SHARP_MIP*/

/*-------------------------------------------------
 *  ILI9341 240X320 TFT LCD
 *-------------------------------------------------*/
#ifndef USE_ILI9341
#  define USE_ILI9341       0
#endif
```

```
#if USE_ILI9341
#  define ILI9341_HOR_RES       LV_HOR_RES
#  define ILI9341_VER_RES       LV_VER_RES
#  define ILI9341_GAMMA         1
#  define ILI9341_TEARING       0
#endif  /*USE_ILI9341*/

/*-----------------------------------------
 *  Linux frame buffer device (/dev/fbx)
 *-----------------------------------------*/
#ifndef USE_FBDEV
#  define USE_FBDEV           0
#endif

#if USE_FBDEV
#  define FBDEV_PATH          "/dev/fb0"
#endif

/*-----------------------------------------
 *  FreeBSD frame buffer device (/dev/fbx)
 *.....................................*/
#ifndef USE_BSD_FBDEV
#  define USE_BSD_FBDEV     0
#endif

#if USE_BSD_FBDEV
# define FBDEV_PATH     "/dev/fb0"
#endif

/*-----------------------------------------
 *  DRM/KMS device (/dev/dri/cardX)
 *-----------------------------------------*/
#ifndef USE_DRM
#  define USE_DRM           1
#endif

#if USE_DRM
#  define DRM_CARD          "/dev/dri/card0"
#  define DRM_CONNECTOR_ID  -1  /* -1 for the first connected one */
#endif

/*********************
 *  INPUT DEVICES
 *********************/

/*--------------
 *    XPT2046
 *--------------*/
#ifndef USE_XPT2046
#  define USE_XPT2046         0
#endif

#if USE_XPT2046
#  define XPT2046_HOR_RES     480
#  define XPT2046_VER_RES     320
```

```
#  define XPT2046_X_MIN       200
#  define XPT2046_Y_MIN       200
#  define XPT2046_X_MAX       3800
#  define XPT2046_Y_MAX       3800
#  define XPT2046_AVG         4
#  define XPT2046_X_INV       0
#  define XPT2046_Y_INV       0
#  define XPT2046_XY_SWAP     0
#endif

/*-----------------
 *    FT5406EE8
 *-----------------*/
#ifndef USE_FT5406EE8
#  define USE_FT5406EE8       0
#endif

#if USE_FT5406EE8
# define FT5406EE8_I2C_ADR   0x38                 /*7 bit address*/
#endif

/*---------------
 *  AD TOUCH
 *--------------*/
#ifndef USE_AD_TOUCH
#  define USE_AD_TOUCH        0
#endif

#if USE_AD_TOUCH
/*No settings*/
#endif


/*--------------------------------------
 * Mouse or touchpad on PC (using SDL)
 *-------------------------------------*/
/*DEPRECATED: Use the SDL driver instead. */
#ifndef USE_MOUSE
#  define USE_MOUSE           0
#endif

#if USE_MOUSE
/*No settings*/
#endif

/*-----------------------------------------
 * Mousewheel as encoder on PC (using SDL)
 *-----------------------------------------*/
/*DEPRECATED: Use the SDL driver instead. */
#ifndef USE_MOUSEWHEEL
#  define USE_MOUSEWHEEL      0
#endif

#if USE_MOUSEWHEEL
/*No settings*/
#endif
```

```
/*-----------------------------------------------
 * Touchscreen, mouse/touchpad or keyboard as libinput interface (for Linux based systems)
 *-----------------------------------------------*/
#ifndef USE_LIBINPUT
#  define USE_LIBINPUT           0
#endif

#ifndef USE_BSD_LIBINPUT
#  define USE_BSD_LIBINPUT       0
#endif

#if USE_LIBINPUT || USE_BSD_LIBINPUT
/*If only a single device of the same type is connected, you can also auto detect it, e.g.:
 *#define LIBINPUT_NAME   libinput_find_dev(LIBINPUT_CAPABILITY_TOUCH, false)*/
#  define LIBINPUT_NAME   "/dev/input/event0"        /*You can use the "evtest" Linux tool to get the list of devices and test them*/

#endif  /*USE_LIBINPUT || USE_BSD_LIBINPUT*/

/*-----------------------------------------------
 * Mouse or touchpad as evdev interface (for Linux based systems)
 *-----------------------------------------------*/
#ifndef USE_EVDEV
#  define USE_EVDEV           1
#endif

#ifndef USE_BSD_EVDEV
#  define USE_BSD_EVDEV       0
#endif

#if USE_EVDEV || USE_BSD_EVDEV
#  define EVDEV_NAME   "/dev/input/event1"        /*You can use the "evtest" Linux tool to get the list of devices and test them*/
#  define EVDEV_SWAP_AXES         0               /*Swap the x and y axes of the touchscreen*/

#  define EVDEV_CALIBRATE         0               /*Scale and offset the touchscreen coordinates by using maximum and minimum values for each axis*/

#  if EVDEV_CALIBRATE
#    define EVDEV_HOR_MIN         0               /*to invert axis swap EVDEV_XXX_MIN by EVDEV_XXX_MAX*/
#    define EVDEV_HOR_MAX      4096               /*"evtest" Linux tool can help to get the correct calibraion values>*/
#    define EVDEV_VER_MIN         0
#    define EVDEV_VER_MAX      4096
#  endif  /*EVDEV_CALIBRATE*/
#endif  /*USE_EVDEV*/

/*-----------------------------------------------
 * Full keyboard support for evdev and libinput interface
 *-----------------------------------------------*/
#  ifndef USE_XKB
```

```
#    define USE_XKB           0
#  endif

#if USE_LIBINPUT || USE_BSD_LIBINPUT || USE_EVDEV || USE_BSD_EVDEV
#  if USE_XKB
#    define XKB_KEY_MAP       { .rules = NULL, \
                                 .model = "pc101", \
                                 .layout = "us", \
                                 .variant = NULL, \
                                 .options = NULL } /*"setxkbmap -query" can help
find the right values for your keyboard*/
#  endif  /*USE_XKB*/
#endif  /*USE_LIBINPUT || USE_BSD_LIBINPUT || USE_EVDEV || USE_BSD_EVDEV*/

/*-----------------------------
 *   Keyboard of a PC (using SDL)
 *----------------------------*/
/*DEPRECATED: Use the SDL driver instead. */
#ifndef USE_KEYBOARD
#  define USE_KEYBOARD        0
#endif

#if USE_KEYBOARD
/*No settings*/
#endif

#endif  /*LV_DRV_CONF_H*/

#endif /*End of "Content enable"*/

/**
 * @file fbdev.h
 *
 */

#ifndef WINDRV_H
#define WINDRV_H

#ifdef __cplusplus
extern "C" {
#endif

/*********************
 *      INCLUDES
 *********************/
#ifndef LV_DRV_NO_CONF
#ifdef LV_CONF_INCLUDE_SIMPLE
#include "lv_drv_conf.h"
#else
#include "../lv_drv_conf.h"
#endif
#endif

#if USE_WINDOWS

#ifdef LV_LVGL_H_INCLUDE_SIMPLE
```

```
#include "lvgl.h"
#else
#include "lvgl/lvgl.h"
#endif

#include <windows.h>

/*********************
 *      DEFINES
 *********************/

/**********************
 *      TYPEDEFS
 **********************/

/**********************
 * GLOBAL PROTOTYPES
 **********************/
extern bool lv_win_exit_flag;
extern lv_disp_t *lv_windows_disp;

HWND windrv_init(void);

/**********************
 *      MACROS
 **********************/

#endif  /*USE_WINDOWS*/

#ifdef __cplusplus
} /* extern "C" */
#endif

#endif /*WIN_DRV_H*/

/**
 * @file win_drv.c
 *
 */

/*********************
 *      INCLUDES
 *********************/
#include "win_drv.h"
#if USE_WINDOWS

#include <windows.h>
#include <windowsx.h>
#include "lvgl/lvgl.h"

#if LV_COLOR_DEPTH < 16
#error Windows driver only supports true RGB colors at this time
#endif

/*********************
 *       DEFINES
```

```
 **********/

 #define WINDOW_STYLE (WS_OVERLAPPEDWINDOW & ~(WS_SIZEBOX | WS_MAXIMIZEBOX |
WS_THICKFRAME))

/**********************
 *      TYPEDEFS
 **********************/

/**********************
 *  STATIC PROTOTYPES
 **********************/
static void do_register(void);
static void win_drv_flush(lv_disp_t *drv, lv_area_t *area, const lv_color_t *
color_p);
static void win_drv_fill(int32_t x1, int32_t y1, int32_t x2, int32_t y2,
lv_color_t color);
static void win_drv_map(int32_t x1, int32_t y1, int32_t x2, int32_t y2, const
lv_color_t * color_p);
static void win_drv_read(lv_indev_t *drv, lv_indev_data_t * data);
static void msg_handler(void *param);

static COLORREF lv_color_to_colorref(const lv_color_t color);

static LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM
lParam);


/**********************
 *  GLOBAL VARIABLES
 **********************/

bool lv_win_exit_flag = false;
lv_disp_t *lv_windows_disp;

/**********************
 *  STATIC VARIABLES
 **********************/
static HWND hwnd;
static uint32_t *fbp = NULL; /* Raw framebuffer memory */
static bool mouse_pressed;
static int mouse_x, mouse_y;


/**********************
 *      MACROS
 **********************/

/**********************
 *   GLOBAL FUNCTIONS
 **********************/
const char g_szClassName[] = "LVGL";

HWND windrv_init(void)
{
    WNDCLASSEX wc;
```

```
    RECT winrect;
    HICON lvgl_icon;

    //Step 1: Registering the Window Class
    wc.cbSize        = sizeof(WNDCLASSEX);
    wc.style         = 0;
    wc.lpfnWndProc   = WndProc;
    wc.cbClsExtra    = 0;
    wc.cbWndExtra    = 0;
    wc.hInstance     = GetModuleHandle(NULL);
    lvgl_icon        = (HICON) LoadImage( NULL, "lvgl_icon.bmp", IMAGE_ICON, 0,
0, LR_LOADFROMFILE);

    if(lvgl_icon == NULL)
        lvgl_icon = LoadIcon(NULL, IDI_APPLICATION);

    wc.hIcon         = lvgl_icon;
    wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
    wc.lpszMenuName  = NULL;
    wc.lpszClassName = g_szClassName;
    wc.hIconSm       = lvgl_icon;

    if(!RegisterClassEx(&wc))
    {
        return NULL;
    }

    winrect.left = 0;
    winrect.right = WINDOW_HOR_RES - 1;
    winrect.top = 0;
    winrect.bottom = WINDOW_VER_RES - 1;
    AdjustWindowRectEx(&winrect, WINDOW_STYLE, FALSE, WS_EX_CLIENTEDGE);
    OffsetRect(&winrect, -winrect.left, -winrect.top);
    // Step 2: Creating the Window
    hwnd = CreateWindowEx(
        WS_EX_CLIENTEDGE,
        g_szClassName,
        "LVGL Simulator",
        WINDOW_STYLE,
        CW_USEDEFAULT, CW_USEDEFAULT, winrect.right, winrect.bottom,
        NULL, NULL, GetModuleHandle(NULL), NULL);

    if(hwnd == NULL)
    {
        return NULL;
    }

    ShowWindow(hwnd, SW_SHOWDEFAULT);
    UpdateWindow(hwnd);


    lv_task_create(msg_handler, 0, LV_TASK_PRIO_HIGHEST, NULL);
    lv_win_exit_flag = false;
    do_register();
}
```

```
/**********************
 *   STATIC FUNCTIONS
 **********************/

static void do_register(void)
{
    static lv_disp_draw_buf_t disp_buf_1;
    static lv_color_t buf1_1[WINDOW_HOR_RES * 100];                      /*A
buffer for 10 rows*/
    lv_disp_draw_buf_init(&disp_draw_buf_1, buf1_1, NULL, WINDOW_HOR_RES * 100);
  /*Initialize the display buffer*/


    /*-----------------------------------
     * Register the display in LVGLGL
     *----------------------------------*/

    static lv_disp_drv_t disp_drv;                         /*Descriptor of a
display driver*/
    lv_disp_drv_init(&disp_drv);                    /*Basic initialization*/

    /*Set up the functions to access to your display*/

    /*Set the resolution of the display*/
    disp_drv.hor_res = WINDOW_HOR_RES;
    disp_drv.ver_res = WINDOW_VER_RES;

    /*Used to copy the buffer's content to the display*/
    disp_drv.flush_cb = win_drv_flush;

    /*Set a display buffer*/
    disp_drv.draw_buf = &disp_buf_1;

    /*Finally register the driver*/
    lv_windows_disp = lv_disp_drv_register(&disp_drv);
    static lv_indev_drv_t indev_drv;
    lv_indev_drv_init(&indev_drv);
    indev_drv.type = LV_INDEV_TYPE_POINTER;
    indev_drv.read_cb = win_drv_read;
    lv_indev_drv_register(&indev_drv);
}

static void msg_handler(void *param)
{
    (void)param;

    MSG msg;
    BOOL bRet;
    if( (bRet = PeekMessage( &msg, NULL, 0, 0, TRUE )) != 0)
    {
        if (bRet == -1)
        {
            return;
        }
        else
```

```
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        if(msg.message == WM_QUIT)
            lv_win_exit_flag = true;
    }
}

 static void win_drv_read(lv_indev_t *drv, lv_indev_data_t * data)
{
    data->state = mouse_pressed ? LV_INDEV_STATE_PR : LV_INDEV_STATE_REL;
    data->point.x = mouse_x;
    data->point.y = mouse_y;
}

 static void on_paint(void)
 {
    HBITMAP bmp = CreateBitmap(WINDOW_HOR_RES, WINDOW_VER_RES, 1, 32, fbp);
    PAINTSTRUCT ps;

    HDC hdc = BeginPaint(hwnd, &ps);

    HDC hdcMem = CreateCompatibleDC(hdc);
    HBITMAP hbmOld = SelectObject(hdcMem, bmp);

    BitBlt(hdc, 0, 0, WINDOW_HOR_RES, WINDOW_VER_RES, hdcMem, 0, 0, SRCCOPY);

    SelectObject(hdcMem, hbmOld);
    DeleteDC(hdcMem);

    EndPaint(hwnd, &ps);
    DeleteObject(bmp);

}
/**
 * Flush a buffer to the marked area
 * @param x1 left coordinate
 * @param y1 top coordinate
 * @param x2 right coordinate
 * @param y2 bottom coordinate
 * @param color_p an array of colors
 */
static void win_drv_flush(lv_disp_t *drv, lv_area_t *area, const lv_color_t *
color_p)
{
    win_drv_map(area->x1, area->y1, area->x2, area->y2, color_p);
    lv_disp_flush_ready(drv);
}

/**
 * Put a color map to the marked area
 * @param x1 left coordinate
 * @param y1 top coordinate
 * @param x2 right coordinate
 * @param y2 bottom coordinate
```

```
 * @param color_p an array of colors
 */
static void win_drv_map(int32_t x1, int32_t y1, int32_t x2, int32_t y2, const
lv_color_t * color_p)
{
    for(int y = y1; y <= y2; y++)
    {
        for(int x = x1; x <= x2; x++)
        {
            fbp[y*WINDOW_HOR_RES+x] = lv_color_to32(*color_p);
            color_p++;
        }
    }
    InvalidateRect(hwnd, NULL, FALSE);
    UpdateWindow(hwnd);
}

static LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM
lParam)
{
    HDC hdc;
    PAINTSTRUCT ps;
    switch(msg) {
    case WM_CREATE:
        fbp = malloc(4*WINDOW_HOR_RES*WINDOW_VER_RES);
        if(fbp == NULL)
            return 1;
        SetTimer(hwnd, 0, 10, (TIMERPROC)lv_task_handler);
        SetTimer(hwnd, 1, 25, NULL);

        return 0;
    case WM_MOUSEMOVE:
    case WM_LBUTTONDOWN:
    case WM_LBUTTONUP:
        mouse_x = GET_X_LPARAM(lParam);
        mouse_y = GET_Y_LPARAM(lParam);
        if(msg == WM_LBUTTONDOWN || msg == WM_LBUTTONUP) {
            mouse_pressed = (msg == WM_LBUTTONDOWN);
        }
        return 0;
    case WM_CLOSE:
        free(fbp);
        fbp = NULL;
        DestroyWindow(hwnd);
        return 0;
    case WM_PAINT:
        on_paint();
        return 0;
    case WM_TIMER:
        lv_tick_inc(25);
        return 0;
    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;
    default:
        break;
```

```
    }
    return DefWindowProc(hwnd, msg, wParam, lParam);
}
static COLORREF lv_color_to_colorref(const lv_color_t color)
{
    uint32_t raw_color = lv_color_to32(color);
    lv_color32_t tmp;
    tmp.full = raw_color;
    uint32_t colorref = RGB(tmp.ch.red, tmp.ch.green, tmp.ch.blue);
    return colorref;
}
#endif

/**
 * @file drm.h
 *
 */

#ifndef DRM_H
#define DRM_H

#ifdef __cplusplus
extern "C" {
#endif

/*********************
 *      INCLUDES
 *********************/
#ifndef LV_DRV_NO_CONF
#ifdef LV_CONF_INCLUDE_SIMPLE
#include "lv_drv_conf.h"
#else
#include "../../lv_drv_conf.h"
#endif
#endif

#if USE_DRM

#ifdef LV_LVGL_H_INCLUDE_SIMPLE
#include "lvgl.h"
#else
#include "../../lvgl/lvgl.h"
#endif

/*********************
 *      DEFINES
 *********************/

/*********************
 *      TYPEDEFS
 *********************/

/**********************
 * GLOBAL PROTOTYPES
 **********************/
void drm_init(void);
```

```
void drm_get_sizes(lv_coord_t *width, lv_coord_t *height, uint32_t *dpi);
void drm_exit(void);
void drm_flush(lv_disp_drv_t * drv, const lv_area_t * area, lv_color_t *
color_p);
void drm_wait_vsync(lv_disp_drv_t * drv);


/**********************
 *      MACROS
 **********************/

#endif  /*USE_DRM*/

#ifdef __cplusplus
} /* extern "C" */
#endif

#endif /*DRM_H*/

/**
 * @file drm.c
 *
 */

/*********************
 *      INCLUDES
 *********************/
#include "drm.h"
#if USE_DRM

#include <unistd.h>
#include <pthread.h>
#include <time.h>
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <sys/mman.h>
#include <inttypes.h>

#include <xf86drm.h>
#include <xf86drmMode.h>
#include <drm/drm_fourcc.h>

#define DBG_TAG "drm"

#define DIV_ROUND_UP(n, d) (((n) + (d) - 1) / (d))

#define print(msg, ...) fprintf(stderr, msg, ##__VA_ARGS__);
#define err(msg, ...)  print("error: " msg "\n", ##__VA_ARGS__)
#define info(msg, ...) print(msg "\n", ##__VA_ARGS__)
#define dbg(msg, ...)  {} //print(DBG_TAG ": " msg "\n", ##__VA_ARGS__)
```

```
struct drm_buffer {
    uint32_t handle;
    uint32_t pitch;
    uint32_t offset;
    unsigned long int size;
    void * map;
    uint32_t fb_handle;
};

struct drm_dev {
    int fd;
    uint32_t conn_id, enc_id, crtc_id, plane_id, crtc_idx;
    uint32_t width, height;
    uint32_t mmWidth, mmHeight;
    uint32_t fourcc;
    drmModeModeInfo mode;
    uint32_t blob_id;
    drmModeCrtc *saved_crtc;
    drmModeAtomicReq *req;
    drmEventContext drm_event_ctx;
    drmModePlane *plane;
    drmModeCrtc *crtc;
    drmModeConnector *conn;
    uint32_t count_plane_props;
    uint32_t count_crtc_props;
    uint32_t count_conn_props;
    drmModePropertyPtr plane_props[128];
    drmModePropertyPtr crtc_props[128];
    drmModePropertyPtr conn_props[128];
    struct drm_buffer drm_bufs[2]; /* DUMB buffers */
    struct drm_buffer *cur_bufs[2]; /* double buffering handling */
} drm_dev;

static uint32_t get_plane_property_id(const char *name)
{
    uint32_t i;

    dbg("Find plane property: %s", name);

    for (i = 0; i < drm_dev.count_plane_props; ++i)
        if (!strcmp(drm_dev.plane_props[i]->name, name))
            return drm_dev.plane_props[i]->prop_id;

    dbg("Unknown plane property: %s", name);

    return 0;
}

static uint32_t get_crtc_property_id(const char *name)
{
    uint32_t i;

    dbg("Find crtc property: %s", name);

    for (i = 0; i < drm_dev.count_crtc_props; ++i)
```

```
        if (!strcmp(drm_dev.crtc_props[i]->name, name))
            return drm_dev.crtc_props[i]->prop_id;

    dbg("Unknown crtc property: %s", name);

    return 0;
}

static uint32_t get_conn_property_id(const char *name)
{
    uint32_t i;

    dbg("Find conn property: %s", name);

    for (i = 0; i < drm_dev.count_conn_props; ++i)
        if (!strcmp(drm_dev.conn_props[i]->name, name))
            return drm_dev.conn_props[i]->prop_id;

    dbg("Unknown conn property: %s", name);

    return 0;
}

static void page_flip_handler(int fd, unsigned int sequence, unsigned int tv_sec,
                  unsigned int tv_usec, void *user_data)
{
    dbg("flip");
}

static int drm_get_plane_props(void)
{
    uint32_t i;

    drmModeObjectPropertiesPtr props = drmModeObjectGetProperties(drm_dev.fd,
drm_dev.plane_id,
                                          DRM_MODE_OBJECT_PLANE);
    if (!props) {
        err("drmModeObjectGetProperties failed");
        return -1;
    }
    dbg("Found %u plane props", props->count_props);
    drm_dev.count_plane_props = props->count_props;
    for (i = 0; i < props->count_props; i++) {
        drm_dev.plane_props[i] = drmModeGetProperty(drm_dev.fd, props->props[i]);
        dbg("Added plane prop %u:%s", drm_dev.plane_props[i]->prop_id,
drm_dev.plane_props[i]->name);
    }
    drmModeFreeObjectProperties(props);

    return 0;
}

static int drm_get_crtc_props(void)
{
    uint32_t i;
```

```
    drmModeObjectPropertiesPtr props = drmModeObjectGetProperties(drm_dev.fd,
drm_dev.crtc_id,
                                          DRM_MODE_OBJECT_CRTC);
    if (!props) {
        err("drmModeObjectGetProperties failed");
        return -1;
    }
    dbg("Found %u crtc props", props->count_props);
    drm_dev.count_crtc_props = props->count_props;
    for (i = 0; i < props->count_props; i++) {
        drm_dev.crtc_props[i] = drmModeGetProperty(drm_dev.fd, props->props[i]);
        dbg("Added crtc prop %u:%s", drm_dev.crtc_props[i]->prop_id,
drm_dev.crtc_props[i]->name);
    }
    drmModeFreeObjectProperties(props);

    return 0;
}

static int drm_get_conn_props(void)
{
    uint32_t i;

    drmModeObjectPropertiesPtr props = drmModeObjectGetProperties(drm_dev.fd,
drm_dev.conn_id,
                                          DRM_MODE_OBJECT_CONNECTOR);
    if (!props) {
        err("drmModeObjectGetProperties failed");
        return -1;
    }
    dbg("Found %u connector props", props->count_props);
    drm_dev.count_conn_props = props->count_props;
    for (i = 0; i < props->count_props; i++) {
        drm_dev.conn_props[i] = drmModeGetProperty(drm_dev.fd, props->props[i]);
        dbg("Added connector prop %u:%s", drm_dev.conn_props[i]->prop_id,
drm_dev.conn_props[i]->name);
    }
    drmModeFreeObjectProperties(props);

    return 0;
}

static int drm_add_plane_property(const char *name, uint64_t value)
{
    int ret;
    uint32_t prop_id = get_plane_property_id(name);

    if (!prop_id) {
        err("Couldn't find plane prop %s", name);
        return -1;
    }

    ret = drmModeAtomicAddProperty(drm_dev.req, drm_dev.plane_id,
get_plane_property_id(name), value);
    if (ret < 0) {
```

```
        err("drmModeAtomicAddProperty (%s:%" PRIu64 ") failed: %d", name, value,
ret);
        return ret;
    }

    return 0;
}

static int drm_add_crtc_property(const char *name, uint64_t value)
{
    int ret;
    uint32_t prop_id = get_crtc_property_id(name);

    if (!prop_id) {
        err("Couldn't find crtc prop %s", name);
        return -1;
    }

    ret = drmModeAtomicAddProperty(drm_dev.req, drm_dev.crtc_id,
get_crtc_property_id(name), value);
    if (ret < 0) {
        err("drmModeAtomicAddProperty (%s:%" PRIu64 ") failed: %d", name, value,
ret);
        return ret;
    }

    return 0;
}

static int drm_add_conn_property(const char *name, uint64_t value)
{
    int ret;
    uint32_t prop_id = get_conn_property_id(name);

    if (!prop_id) {
        err("Couldn't find conn prop %s", name);
        return -1;
    }

    ret = drmModeAtomicAddProperty(drm_dev.req, drm_dev.conn_id,
get_conn_property_id(name), value);
    if (ret < 0) {
        err("drmModeAtomicAddProperty (%s:%" PRIu64 ") failed: %d", name, value,
ret);
        return ret;
    }

    return 0;
}

static int drm_dmabuf_set_plane(struct drm_buffer *buf)
{
    int ret;
    static int first = 1;
    uint32_t flags = DRM_MODE_PAGE_FLIP_EVENT;
```

```
    drm_dev.req = drmModeAtomicAlloc();

    /* On first Atomic commit, do a modeset */
    if (first) {
        drm_add_conn_property("CRTC_ID", drm_dev.crtc_id);

        drm_add_crtc_property("MODE_ID", drm_dev.blob_id);
        drm_add_crtc_property("ACTIVE", 1);

        flags |= DRM_MODE_ATOMIC_ALLOW_MODESET;

        first = 0;
    }

    drm_add_plane_property("FB_ID", buf->fb_handle);
    drm_add_plane_property("CRTC_ID", drm_dev.crtc_id);
    drm_add_plane_property("SRC_X", 0);
    drm_add_plane_property("SRC_Y", 0);
    drm_add_plane_property("SRC_W", drm_dev.width << 16);
    drm_add_plane_property("SRC_H", drm_dev.height << 16);
    drm_add_plane_property("CRTC_X", 0);
    drm_add_plane_property("CRTC_Y", 0);
    drm_add_plane_property("CRTC_W", drm_dev.width);
    drm_add_plane_property("CRTC_H", drm_dev.height);

    ret = drmModeAtomicCommit(drm_dev.fd, drm_dev.req, flags, NULL);
    if (ret) {
        err("drmModeAtomicCommit failed: %s", strerror(errno));
        drmModeAtomicFree(drm_dev.req);
        return ret;
    }

    return 0;
}

static int find_plane(unsigned int fourcc, uint32_t *plane_id, uint32_t crtc_id,
uint32_t crtc_idx)
{
    drmModePlaneResPtr planes;
    drmModePlanePtr plane;
    unsigned int i;
    unsigned int j;
    int ret = 0;
    unsigned int format = fourcc;

    planes = drmModeGetPlaneResources(drm_dev.fd);
    if (!planes) {
        err("drmModeGetPlaneResources failed");
        return -1;
    }

    dbg("drm: found planes %u", planes->count_planes);

    for (i = 0; i < planes->count_planes; ++i) {
        plane = drmModeGetPlane(drm_dev.fd, planes->planes[i]);
        if (!plane) {
```

```
            err("drmModeGetPlane failed: %s", strerror(errno));
            break;
        }

        if (!(plane->possible_crtcs & (1 << crtc_idx))) {
            drmModeFreePlane(plane);
            continue;
        }

        for (j = 0; j < plane->count_formats; ++j) {
            if (plane->formats[j] == format)
                break;
        }

        if (j == plane->count_formats) {
            drmModeFreePlane(plane);
            continue;
        }

        *plane_id = plane->plane_id;
        drmModeFreePlane(plane);

        dbg("found plane %d", *plane_id);

        break;
    }

    if (i == planes->count_planes)
        ret = -1;

    drmModeFreePlaneResources(planes);

    return ret;
}

static int drm_find_connector(void)
{
    drmModeConnector *conn = NULL;
    drmModeEncoder *enc = NULL;
    drmModeRes *res;
    int i;

    if ((res = drmModeGetResources(drm_dev.fd)) == NULL) {
        err("drmModeGetResources() failed");
        return -1;
    }

    if (res->count_crtcs <= 0) {
        err("no Crtcs");
        goto free_res;
    }

    /* find all available connectors */
    for (i = 0; i < res->count_connectors; i++) {
        conn = drmModeGetConnector(drm_dev.fd, res->connectors[i]);
        if (!conn)
```

```
            continue;

#if DRM_CONNECTOR_ID >= 0
        if (conn->connector_id != DRM_CONNECTOR_ID) {
            drmModeFreeConnector(conn);
            continue;
        }
#endif

        if (conn->connection == DRM_MODE_CONNECTED) {
            dbg("drm: connector %d: connected", conn->connector_id);
        } else if (conn->connection == DRM_MODE_DISCONNECTED) {
            dbg("drm: connector %d: disconnected", conn->connector_id);
        } else if (conn->connection == DRM_MODE_UNKNOWNCONNECTION) {
            dbg("drm: connector %d: unknownconnection", conn->connector_id);
        } else {
            dbg("drm: connector %d: unknown", conn->connector_id);
        }

        if (conn->connection == DRM_MODE_CONNECTED && conn->count_modes > 0)
            break;

        drmModeFreeConnector(conn);
        conn = NULL;
    };

    if (!conn) {
        err("suitable connector not found");
        goto free_res;
    }

    drm_dev.conn_id = conn->connector_id;
    dbg("conn_id: %d", drm_dev.conn_id);
    drm_dev.mmWidth = conn->mmWidth;
    drm_dev.mmHeight = conn->mmHeight;

    memcpy(&drm_dev.mode, &conn->modes[0], sizeof(drmModeModeInfo));

    if (drmModeCreatePropertyBlob(drm_dev.fd, &drm_dev.mode,
sizeof(drm_dev.mode),
                        &drm_dev.blob_id)) {
        err("error creating mode blob");
        goto free_res;
    }

    drm_dev.width = conn->modes[0].hdisplay;
    drm_dev.height = conn->modes[0].vdisplay;

    for (i = 0 ; i < res->count_encoders; i++) {
        enc = drmModeGetEncoder(drm_dev.fd, res->encoders[i]);
        if (!enc)
            continue;

        dbg("enc%d enc_id %d conn enc_id %d", i, enc->encoder_id, conn-
>encoder_id);
```

```
        if (enc->encoder_id == conn->encoder_id)
            break;

        drmModeFreeEncoder(enc);
        enc = NULL;
    }

    if (enc) {
        drm_dev.enc_id = enc->encoder_id;
        dbg("enc_id: %d", drm_dev.enc_id);
        drm_dev.crtc_id = enc->crtc_id;
        dbg("crtc_id: %d", drm_dev.crtc_id);
        drmModeFreeEncoder(enc);
    } else {
        /* Encoder hasn't been associated yet, look it up */
        for (i = 0; i < conn->count_encoders; i++) {
            int crtc, crtc_id = -1;

            enc = drmModeGetEncoder(drm_dev.fd, conn->encoders[i]);
            if (!enc)
                continue;

            for (crtc = 0 ; crtc < res->count_crtcs; crtc++) {
                uint32_t crtc_mask = 1 << crtc;

                crtc_id = res->crtcs[crtc];

                dbg("enc_id %d crtc%d id %d mask %x possible %x", enc-
>encoder_id, crtc, crtc_id, crtc_mask, enc->possible_crtcs);

                if (enc->possible_crtcs & crtc_mask)
                    break;
            }

            if (crtc_id > 0) {
                drm_dev.enc_id = enc->encoder_id;
                dbg("enc_id: %d", drm_dev.enc_id);
                drm_dev.crtc_id = crtc_id;
                dbg("crtc_id: %d", drm_dev.crtc_id);
                break;
            }

            drmModeFreeEncoder(enc);
            enc = NULL;
        }

        if (!enc) {
            err("suitable encoder not found");
            goto free_res;
        }

        drmModeFreeEncoder(enc);
    }

    drm_dev.crtc_idx = -1;
```

```
    for (i = 0; i < res->count_crtcs; ++i) {
        if (drm_dev.crtc_id == res->crtcs[i]) {
            drm_dev.crtc_idx = i;
            break;
        }
    }

    if (drm_dev.crtc_idx == -1) {
        err("drm: CRTC not found");
        goto free_res;
    }

    dbg("crtc_idx: %d", drm_dev.crtc_idx);

    return 0;

free_res:
    drmModeFreeResources(res);

    return -1;
}

static int drm_open(const char *path)
{
    int fd, flags;
    uint64_t has_dumb;
    int ret;

    fd = open(path, O_RDWR);
    if (fd < 0) {
        err("cannot open \"%s\"", path);
        return -1;
    }

    /* set FD_CLOEXEC flag */
    if ((flags = fcntl(fd, F_GETFD)) < 0 ||
         fcntl(fd, F_SETFD, flags | FD_CLOEXEC) < 0) {
        err("fcntl FD_CLOEXEC failed");
        goto err;
    }

    /* check capability */
    ret = drmGetCap(fd, DRM_CAP_DUMB_BUFFER, &has_dumb);
    if (ret < 0 || has_dumb == 0) {
        err("drmGetCap DRM_CAP_DUMB_BUFFER failed or \"%s\" doesn't have dumb "
            "buffer", path);
        goto err;
    }

    return fd;
err:
    close(fd);
    return -1;
}

static int drm_setup(unsigned int fourcc)
```

```
{
    int ret;
    const char *device_path = NULL;

    device_path = getenv("DRM_CARD");
    if (!device_path)
        device_path = DRM_CARD;

    drm_dev.fd = drm_open(device_path);
    if (drm_dev.fd < 0)
        return -1;

    ret = drmSetClientCap(drm_dev.fd, DRM_CLIENT_CAP_ATOMIC, 1);
    if (ret) {
        err("No atomic modesetting support: %s", strerror(errno));
        goto err;
    }

    ret = drm_find_connector();
    if (ret) {
        err("available drm devices not found");
        goto err;
    }

    ret = find_plane(fourcc, &drm_dev.plane_id, drm_dev.crtc_id,
drm_dev.crtc_idx);
    if (ret) {
        err("Cannot find plane");
        goto err;
    }

    // xjt modify
    // drm_dev.plane = drmModeGetPlane(drm_dev.fd, drm_dev.plane_id);
    // if (!drm_dev.plane) {
    //  err("Cannot get plane");
    //  goto err;
    // }
    // for mipi-dsi
    drm_dev.crtc_id = 115;
    drm_dev.conn_id = 163;
    drm_dev.plane_id = 131;

    drm_dev.crtc = drmModeGetCrtc(drm_dev.fd, drm_dev.crtc_id);
    if (!drm_dev.crtc) {
        err("Cannot get crtc");
        goto err;
    }

    drm_dev.conn = drmModeGetConnector(drm_dev.fd, drm_dev.conn_id);
    if (!drm_dev.conn) {
        err("Cannot get connector");
        goto err;
    }

    ret = drm_get_plane_props();
    if (ret) {
```

```
        err("Cannot get plane props");
        goto err;
    }

    ret = drm_get_crtc_props();
    if (ret) {
        err("Cannot get crtc props");
        goto err;
    }

    ret = drm_get_conn_props();
    if (ret) {
        err("Cannot get connector props");
        goto err;
    }

    drm_dev.drm_event_ctx.version = DRM_EVENT_CONTEXT_VERSION;
    drm_dev.drm_event_ctx.page_flip_handler = page_flip_handler;
    drm_dev.fourcc = fourcc;

    info("drm: Found plane_id: %u connector_id: %d crtc_id: %d",
        drm_dev.plane_id, drm_dev.conn_id, drm_dev.crtc_id);

    info("drm: %dx%d (%dmm X% dmm) pixel format %c%c%c%c",
         drm_dev.width, drm_dev.height, drm_dev.mmWidth, drm_dev.mmHeight,
         (fourcc>>0)&0xff, (fourcc>>8)&0xff, (fourcc>>16)&0xff,
(fourcc>>24)&0xff);

    return 0;

err:
    close(drm_dev.fd);
    return -1;
}

static int drm_allocate_dumb(struct drm_buffer *buf)
{
    struct drm_mode_create_dumb creq;
    struct drm_mode_map_dumb mreq;
    uint32_t handles[4] = {0}, pitches[4] = {0}, offsets[4] = {0};
    int ret;

    /* create dumb buffer */
    memset(&creq, 0, sizeof(creq));
    creq.width = drm_dev.width;
    creq.height = drm_dev.height;
    creq.bpp = LV_COLOR_DEPTH;
    ret = drmIoctl(drm_dev.fd, DRM_IOCTL_MODE_CREATE_DUMB, &creq);
    if (ret < 0) {
        err("DRM_IOCTL_MODE_CREATE_DUMB fail");
        return -1;
    }

    buf->handle = creq.handle;
    buf->pitch = creq.pitch;
    dbg("pitch %d", buf->pitch);
```

```
    buf->size = creq.size;
    dbg("size %d", buf->size);

    /* prepare buffer for memory mapping */
    memset(&mreq, 0, sizeof(mreq));
    mreq.handle = creq.handle;
    ret = drmIoctl(drm_dev.fd, DRM_IOCTL_MODE_MAP_DUMB, &mreq);
    if (ret) {
        err("DRM_IOCTL_MODE_MAP_DUMB fail");
        return -1;
    }

    buf->offset = mreq.offset;

    /* perform actual memory mapping */
    buf->map = mmap(0, creq.size, PROT_READ | PROT_WRITE, MAP_SHARED, drm_dev.fd,
mreq.offset);
    if (buf->map == MAP_FAILED) {
        err("mmap fail");
        return -1;
    }

    /* clear the framebuffer to 0 (= full transparency in ARGB8888) */
    memset(buf->map, 0, creq.size);

    /* create framebuffer object for the dumb-buffer */
    handles[0] = creq.handle;
    pitches[0] = creq.pitch;
    offsets[0] = 0;
    ret = drmModeAddFB2(drm_dev.fd, drm_dev.width, drm_dev.height,
drm_dev.fourcc,
                handles, pitches, offsets, &buf->fb_handle, 0);
    if (ret) {
        err("drmModeAddFB fail");
        return -1;
    }

    return 0;
}

static int drm_setup_buffers(void)
{
    int ret;

    /* Allocate DUMB buffers */
    ret = drm_allocate_dumb(&drm_dev.drm_bufs[0]);
    if (ret)
        return ret;

    ret = drm_allocate_dumb(&drm_dev.drm_bufs[1]);
    if (ret)
        return ret;

    /* Set buffering handling */
    drm_dev.cur_bufs[0] = NULL;
    drm_dev.cur_bufs[1] = &drm_dev.drm_bufs[0];
```

```
    return 0;
}

void drm_wait_vsync(lv_disp_drv_t *disp_drv)
{
    int ret;
    fd_set fds;
    FD_ZERO(&fds);
    FD_SET(drm_dev.fd, &fds);

    do {
        ret = select(drm_dev.fd + 1, &fds, NULL, NULL, NULL);
    } while (ret == -1 && errno == EINTR);

    if (ret < 0) {
        err("select failed: %s", strerror(errno));
        drmModeAtomicFree(drm_dev.req);
        drm_dev.req = NULL;
        return;
    }

    if (FD_ISSET(drm_dev.fd, &fds))
        drmHandleEvent(drm_dev.fd, &drm_dev.drm_event_ctx);

    drmModeAtomicFree(drm_dev.req);
    drm_dev.req = NULL;
}

void drm_flush(lv_disp_drv_t *disp_drv, const lv_area_t *area, lv_color_t
*color_p)
{
    struct drm_buffer *fbuf = drm_dev.cur_bufs[1];
    lv_coord_t w = (area->x2 - area->x1 + 1);
    lv_coord_t h = (area->y2 - area->y1 + 1);
    int i, y;

    dbg("x %d:%d y %d:%d w %d h %d", area->x1, area->x2, area->y1, area->y2, w,
h);

    /* Partial update */
    if ((w != drm_dev.width || h != drm_dev.height) && drm_dev.cur_bufs[0])
        memcpy(fbuf->map, drm_dev.cur_bufs[0]->map, fbuf->size);

    for (y = 0, i = area->y1 ; i <= area->y2 ; ++i, ++y) {
                memcpy((uint8_t *)fbuf->map + (area->x1 * (LV_COLOR_SIZE/8)) +
(fbuf->pitch * i),
                       (uint8_t *)color_p + (w * (LV_COLOR_SIZE/8) * y),
               w * (LV_COLOR_SIZE/8));
    }

    if (drm_dev.req)
        drm_wait_vsync(disp_drv);

    /* show fbuf plane */
    if (drm_dmabuf_set_plane(fbuf)) {
```

```
        err("Flush fail");
        return;
    }
    else
        dbg("Flush done");

    if (!drm_dev.cur_bufs[0])
        drm_dev.cur_bufs[1] = &drm_dev.drm_bufs[1];
    else
        drm_dev.cur_bufs[1] = drm_dev.cur_bufs[0];

    drm_dev.cur_bufs[0] = fbuf;

    lv_disp_flush_ready(disp_drv);
}

#if LV_COLOR_DEPTH == 32
#define DRM_FOURCC DRM_FORMAT_ARGB8888
#elif LV_COLOR_DEPTH == 16
#define DRM_FOURCC DRM_FORMAT_RGB565
#else
#error LV_COLOR_DEPTH not supported
#endif

void drm_get_sizes(lv_coord_t *width, lv_coord_t *height, uint32_t *dpi)
{
    if (width)
        *width = drm_dev.width;

    if (height)
        *height = drm_dev.height;

    if (dpi && drm_dev.mmWidth)
        *dpi = DIV_ROUND_UP(drm_dev.width * 25400, drm_dev.mmWidth * 1000);
}

void drm_init(void)
{
    int ret;

    ret = drm_setup(DRM_FOURCC);
    if (ret) {
        close(drm_dev.fd);
        drm_dev.fd = -1;
        return;
    }

    ret = drm_setup_buffers();
    if (ret) {
        err("DRM buffer allocation failed");
        close(drm_dev.fd);
        drm_dev.fd = -1;
        return;
    }

    info("DRM subsystem and buffer mapped successfully");
```

```
}

void drm_exit(void)
{
    close(drm_dev.fd);
    drm_dev.fd = -1;
}

#endif

/**
 * @file fbdev.h
 *
 */

#ifndef FBDEV_H
#define FBDEV_H

#ifdef __cplusplus
extern "C" {
#endif

/*********************
 *      INCLUDES
 *********************/
#ifndef LV_DRV_NO_CONF
#ifdef LV_CONF_INCLUDE_SIMPLE
#include "lv_drv_conf.h"
#else
#include "../../lv_drv_conf.h"
#endif
#endif

#if USE_FBDEV || USE_BSD_FBDEV

#ifdef LV_LVGL_H_INCLUDE_SIMPLE
#include "lvgl.h"
#else
#include "lvgl/lvgl.h"
#endif

/*********************
 *      DEFINES
 *********************/

/*********************
 *      TYPEDEFS
 *********************/

/**********************
 * GLOBAL PROTOTYPES
 **********************/
void fbdev_init(void);
void fbdev_exit(void);
void fbdev_flush(lv_disp_drv_t * drv, const lv_area_t * area, lv_color_t *
color_p);
```

```c
void fbdev_get_sizes(uint32_t *width, uint32_t *height, uint32_t *dpi);
/**
 * Set the X and Y offset in the variable framebuffer info.
 * @param xoffset horizontal offset
 * @param yoffset vertical offset
 */
void fbdev_set_offset(uint32_t xoffset, uint32_t yoffset);


/**********************
 *      MACROS
 **********************/

#endif  /*USE_FBDEV*/

#ifdef __cplusplus
} /* extern "C" */
#endif

#endif /*FBDEV_H*/

/**
 * @file fbdev.c
 *
 */

/*********************
 *      INCLUDES
 *********************/
#include "fbdev.h"
#if USE_FBDEV || USE_BSD_FBDEV

#include <stdlib.h>
#include <unistd.h>
#include <stddef.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/ioctl.h>

#if USE_BSD_FBDEV
#include <sys/fcntl.h>
#include <sys/time.h>
#include <sys/consio.h>
#include <sys/fbio.h>
#else  /* USE_BSD_FBDEV */
#include <linux/fb.h>
#endif /* USE_BSD_FBDEV */

/*********************
 *      DEFINES
 *********************/
#ifndef FBDEV_PATH
#define FBDEV_PATH  "/dev/fb0"
#endif
```

```c
#ifndef DIV_ROUND_UP
#define DIV_ROUND_UP(n, d) (((n) + (d) - 1) / (d))
#endif

/**********************
 *      TYPEDEFS
 **********************/

/**********************
 *      STRUCTURES
 **********************/

struct bsd_fb_var_info{
    uint32_t xoffset;
    uint32_t yoffset;
    uint32_t xres;
    uint32_t yres;
    int bits_per_pixel;
 };

struct bsd_fb_fix_info{
    long int line_length;
    long int smem_len;
};

/**********************
 *  STATIC PROTOTYPES
 **********************/

/**********************
 *  STATIC VARIABLES
 **********************/
#if USE_BSD_FBDEV
static struct bsd_fb_var_info vinfo;
static struct bsd_fb_fix_info finfo;
#else
static struct fb_var_screeninfo vinfo;
static struct fb_fix_screeninfo finfo;
#endif /* USE_BSD_FBDEV */
static char *fbp = 0;
static long int screensize = 0;
static int fbfd = 0;

/**********************
 *      MACROS
 **********************/

#if USE_BSD_FBDEV
#define FBIOBLANK FBIO_BLANK
#endif /* USE_BSD_FBDEV */

/**********************
 *   GLOBAL FUNCTIONS
 **********************/

void fbdev_init(void)
```

```
{
    // Open the file for reading and writing
    fbfd = open(FBDEV_PATH, O_RDWR);
    if(fbfd == -1) {
        perror("Error: cannot open framebuffer device");
        return;
    }
    LV_LOG_INFO("The framebuffer device was opened successfully");

    // Make sure that the display is on.
    if (ioctl(fbfd, FBIOBLANK, FB_BLANK_UNBLANK) != 0) {
        perror("ioctl(FBIOBLANK)");
        return;
    }

#if USE_BSD_FBDEV
    struct fbtype fb;
    unsigned line_length;

    //Get fb type
    if (ioctl(fbfd, FBIOGTYPE, &fb) != 0) {
        perror("ioctl(FBIOGTYPE)");
        return;
    }

    //Get screen width
    if (ioctl(fbfd, FBIO_GETLINEWIDTH, &line_length) != 0) {
        perror("ioctl(FBIO_GETLINEWIDTH)");
        return;
    }

    vinfo.xres = (unsigned) fb.fb_width;
    vinfo.yres = (unsigned) fb.fb_height;
    vinfo.bits_per_pixel = fb.fb_depth;
    vinfo.xoffset = 0;
    vinfo.yoffset = 0;
    finfo.line_length = line_length;
    finfo.smem_len = finfo.line_length * vinfo.yres;
#else /* USE_BSD_FBDEV */

    // Get fixed screen information
    if(ioctl(fbfd, FBIOGET_FSCREENINFO, &finfo) == -1) {
        perror("Error reading fixed information");
        return;
    }

    // Get variable screen information
    if(ioctl(fbfd, FBIOGET_VSCREENINFO, &vinfo) == -1) {
        perror("Error reading variable information");
        return;
    }
#endif /* USE_BSD_FBDEV */

    LV_LOG_INFO("%dx%d, %dbpp", vinfo.xres, vinfo.yres, vinfo.bits_per_pixel);

    // Figure out the size of the screen in bytes
```

```
    screensize =  finfo.smem_len; //finfo.line_length * vinfo.yres;

    // Map the device to memory
    fbp = (char *)mmap(0, screensize, PROT_READ | PROT_WRITE, MAP_SHARED, fbfd,
0);
    if((intptr_t)fbp == -1) {
        perror("Error: failed to map framebuffer device to memory");
        return;
    }

    // Don't initialise the memory to retain what's currently displayed / avoid
clearing the screen.
    // This is important for applications that only draw to a subsection of the
full framebuffer.

    LV_LOG_INFO("The framebuffer device was mapped to memory successfully");

}

void fbdev_exit(void)
{
    close(fbfd);
}

/**
 * Flush a buffer to the marked area
 * @param drv pointer to driver where this function belongs
 * @param area an area where to copy `color_p`
 * @param color_p an array of pixels to copy to the `area` part of the screen
 */
void fbdev_flush(lv_disp_drv_t * drv, const lv_area_t * area, lv_color_t *
color_p)
{
    if(fbp == NULL ||
            area->x2 < 0 ||
            area->y2 < 0 ||
            area->x1 > (int32_t)vinfo.xres - 1 ||
            area->y1 > (int32_t)vinfo.yres - 1) {
        lv_disp_flush_ready(drv);
        return;
    }

    /*Truncate the area to the screen*/
    int32_t act_x1 = area->x1 < 0 ? 0 : area->x1;
    int32_t act_y1 = area->y1 < 0 ? 0 : area->y1;
    int32_t act_x2 = area->x2 > (int32_t)vinfo.xres - 1 ? (int32_t)vinfo.xres - 1
: area->x2;
    int32_t act_y2 = area->y2 > (int32_t)vinfo.yres - 1 ? (int32_t)vinfo.yres - 1
: area->y2;



    lv_coord_t w = (act_x2 - act_x1 + 1);
    long int location = 0;
    long int byte_location = 0;
    unsigned char bit_location = 0;
```

```
    /*32 or 24 bit per pixel*/
    if(vinfo.bits_per_pixel == 32 || vinfo.bits_per_pixel == 24) {
        uint32_t * fbp32 = (uint32_t *)fbp;
        int32_t y;
        for(y = act_y1; y <= act_y2; y++) {
            location = (act_x1 + vinfo.xoffset) + (y + vinfo.yoffset) *
finfo.line_length / 4;
            memcpy(&fbp32[location], (uint32_t *)color_p, (act_x2 - act_x1 + 1) *
4);
            color_p += w;
        }
    }
    /*16 bit per pixel*/
    else if(vinfo.bits_per_pixel == 16) {
        uint16_t * fbp16 = (uint16_t *)fbp;
        int32_t y;
        for(y = act_y1; y <= act_y2; y++) {
            location = (act_x1 + vinfo.xoffset) + (y + vinfo.yoffset) *
finfo.line_length / 2;
            memcpy(&fbp16[location], (uint32_t *)color_p, (act_x2 - act_x1 + 1) *
2);
            color_p += w;
        }
    }
    /*8 bit per pixel*/
    else if(vinfo.bits_per_pixel == 8) {
        uint8_t * fbp8 = (uint8_t *)fbp;
        int32_t y;
        for(y = act_y1; y <= act_y2; y++) {
            location = (act_x1 + vinfo.xoffset) + (y + vinfo.yoffset) *
finfo.line_length;
            memcpy(&fbp8[location], (uint32_t *)color_p, (act_x2 - act_x1 + 1));
            color_p += w;
        }
    }
    /*1 bit per pixel*/
    else if(vinfo.bits_per_pixel == 1) {
        uint8_t * fbp8 = (uint8_t *)fbp;
        int32_t x;
        int32_t y;
        for(y = act_y1; y <= act_y2; y++) {
            for(x = act_x1; x <= act_x2; x++) {
                location = (x + vinfo.xoffset) + (y + vinfo.yoffset) *
vinfo.xres;
                byte_location = location / 8; /* find the byte we need to change
*/
                bit_location = location % 8; /* inside the byte found, find the
bit we need to change */
                fbp8[byte_location] &= ~(((uint8_t)(1)) << bit_location);
                fbp8[byte_location] |= ((uint8_t)(color_p->full)) <<
bit_location;
                color_p++;
            }

            color_p += area->x2 - act_x2;
        }
```

```
    } else {
        /*Not supported bit per pixel*/
    }

    //May be some direct update command is required
    //ret = ioctl(state->fd, FBIO_UPDATE, (unsigned long)((uintptr_t)rect));

    lv_disp_flush_ready(drv);
}

void fbdev_get_sizes(uint32_t *width, uint32_t *height, uint32_t *dpi) {
    if (width)
        *width = vinfo.xres;

    if (height)
        *height = vinfo.yres;

    if (dpi && vinfo.height)
        *dpi = DIV_ROUND_UP(vinfo.xres * 254, vinfo.width * 10);
}

void fbdev_set_offset(uint32_t xoffset, uint32_t yoffset) {
    vinfo.xoffset = xoffset;
    vinfo.yoffset = yoffset;
}

/**********************
 *   STATIC FUNCTIONS
 **********************/

#endif

/**
 * @file fbdev.c
 *
 */

/*********************
 *      INCLUDES
 *********************/
#include "fbdev.h"
#if USE_FBDEV || USE_BSD_FBDEV

#include <stdlib.h>
#include <unistd.h>
#include <stddef.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/ioctl.h>

#if USE_BSD_FBDEV
#include <sys/fcntl.h>
#include <sys/time.h>
#include <sys/consio.h>
#include <sys/fbio.h>
```

```
#else  /* USE_BSD_FBDEV */
#include <linux/fb.h>
#endif /* USE_BSD_FBDEV */

/*********************
 *      DEFINES
 *********************/
#ifndef FBDEV_PATH
#define FBDEV_PATH  "/dev/fb0"
#endif

#ifndef DIV_ROUND_UP
#define DIV_ROUND_UP(n, d) (((n) + (d) - 1) / (d))
#endif

/**********************
 *      TYPEDEFS
 **********************/

/**********************
 *      STRUCTURES
 **********************/

struct bsd_fb_var_info{
    uint32_t xoffset;
    uint32_t yoffset;
    uint32_t xres;
    uint32_t yres;
    int bits_per_pixel;
 };

struct bsd_fb_fix_info{
    long int line_length;
    long int smem_len;
};

/**********************
 *  STATIC PROTOTYPES
 **********************/

/**********************
 *  STATIC VARIABLES
 **********************/
#if USE_BSD_FBDEV
static struct bsd_fb_var_info vinfo;
static struct bsd_fb_fix_info finfo;
#else
static struct fb_var_screeninfo vinfo;
static struct fb_fix_screeninfo finfo;
#endif /* USE_BSD_FBDEV */
static char *fbp = 0;
static long int screensize = 0;
static int fbfd = 0;

/**********************
 *      MACROS
```

```
 **********************/

#if USE_BSD_FBDEV
#define FBIOBLANK FBIO_BLANK
#endif /* USE_BSD_FBDEV */

/**********************
 *   GLOBAL FUNCTIONS
 **********************/

void fbdev_init(void)
{
    // Open the file for reading and writing
    fbfd = open(FBDEV_PATH, O_RDWR);
    if(fbfd == -1) {
        perror("Error: cannot open framebuffer device");
        return;
    }
    LV_LOG_INFO("The framebuffer device was opened successfully");

    // Make sure that the display is on.
    if (ioctl(fbfd, FBIOBLANK, FB_BLANK_UNBLANK) != 0) {
        perror("ioctl(FBIOBLANK)");
        return;
    }

#if USE_BSD_FBDEV
    struct fbtype fb;
    unsigned line_length;

    //Get fb type
    if (ioctl(fbfd, FBIOGTYPE, &fb) != 0) {
        perror("ioctl(FBIOGTYPE)");
        return;
    }

    //Get screen width
    if (ioctl(fbfd, FBIO_GETLINEWIDTH, &line_length) != 0) {
        perror("ioctl(FBIO_GETLINEWIDTH)");
        return;
    }

    vinfo.xres = (unsigned) fb.fb_width;
    vinfo.yres = (unsigned) fb.fb_height;
    vinfo.bits_per_pixel = fb.fb_depth;
    vinfo.xoffset = 0;
    vinfo.yoffset = 0;
    finfo.line_length = line_length;
    finfo.smem_len = finfo.line_length * vinfo.yres;
#else /* USE_BSD_FBDEV */

    // Get fixed screen information
    if(ioctl(fbfd, FBIOGET_FSCREENINFO, &finfo) == -1) {
        perror("Error reading fixed information");
        return;
    }
```

```
    // Get variable screen information
    if(ioctl(fbfd, FBIOGET_VSCREENINFO, &vinfo) == -1) {
        perror("Error reading variable information");
        return;
    }
#endif /* USE_BSD_FBDEV */

    LV_LOG_INFO("%dx%d, %dbpp", vinfo.xres, vinfo.yres, vinfo.bits_per_pixel);

    // Figure out the size of the screen in bytes
    screensize =  finfo.smem_len; //finfo.line_length * vinfo.yres;

    // Map the device to memory
    fbp = (char *)mmap(0, screensize, PROT_READ | PROT_WRITE, MAP_SHARED, fbfd,
0);
    if((intptr_t)fbp == -1) {
        perror("Error: failed to map framebuffer device to memory");
        return;
    }

    // Don't initialise the memory to retain what's currently displayed / avoid
clearing the screen.
    // This is important for applications that only draw to a subsection of the
full framebuffer.

    LV_LOG_INFO("The framebuffer device was mapped to memory successfully");

}

void fbdev_exit(void)
{
    close(fbfd);
}

/**
 * Flush a buffer to the marked area
 * @param drv pointer to driver where this function belongs
 * @param area an area where to copy `color_p`
 * @param color_p an array of pixels to copy to the `area` part of the screen
 */
void fbdev_flush(lv_disp_drv_t * drv, const lv_area_t * area, lv_color_t *
color_p)
{
    if(fbp == NULL ||
            area->x2 < 0 ||
            area->y2 < 0 ||
            area->x1 > (int32_t)vinfo.xres - 1 ||
            area->y1 > (int32_t)vinfo.yres - 1) {
        lv_disp_flush_ready(drv);
        return;
    }

    /*Truncate the area to the screen*/
    int32_t act_x1 = area->x1 < 0 ? 0 : area->x1;
    int32_t act_y1 = area->y1 < 0 ? 0 : area->y1;
```

```
    int32_t act_x2 = area->x2 > (int32_t)vinfo.xres - 1 ? (int32_t)vinfo.xres - 1
: area->x2;
    int32_t act_y2 = area->y2 > (int32_t)vinfo.yres - 1 ? (int32_t)vinfo.yres - 1
: area->y2;


    lv_coord_t w = (act_x2 - act_x1 + 1);
    long int location = 0;
    long int byte_location = 0;
    unsigned char bit_location = 0;

    /*32 or 24 bit per pixel*/
    if(vinfo.bits_per_pixel == 32 || vinfo.bits_per_pixel == 24) {
        uint32_t * fbp32 = (uint32_t *)fbp;
        int32_t y;
        for(y = act_y1; y <= act_y2; y++) {
            location = (act_x1 + vinfo.xoffset) + (y + vinfo.yoffset) *
finfo.line_length / 4;
            memcpy(&fbp32[location], (uint32_t *)color_p, (act_x2 - act_x1 + 1) *
4);
            color_p += w;
        }
    }
    /*16 bit per pixel*/
    else if(vinfo.bits_per_pixel == 16) {
        uint16_t * fbp16 = (uint16_t *)fbp;
        int32_t y;
        for(y = act_y1; y <= act_y2; y++) {
            location = (act_x1 + vinfo.xoffset) + (y + vinfo.yoffset) *
finfo.line_length / 2;
            memcpy(&fbp16[location], (uint32_t *)color_p, (act_x2 - act_x1 + 1) *
2);
            color_p += w;
        }
    }
    /*8 bit per pixel*/
    else if(vinfo.bits_per_pixel == 8) {
        uint8_t * fbp8 = (uint8_t *)fbp;
        int32_t y;
        for(y = act_y1; y <= act_y2; y++) {
            location = (act_x1 + vinfo.xoffset) + (y + vinfo.yoffset) *
finfo.line_length;
            memcpy(&fbp8[location], (uint32_t *)color_p, (act_x2 - act_x1 + 1));
            color_p += w;
        }
    }
    /*1 bit per pixel*/
    else if(vinfo.bits_per_pixel == 1) {
        uint8_t * fbp8 = (uint8_t *)fbp;
        int32_t x;
        int32_t y;
        for(y = act_y1; y <= act_y2; y++) {
            for(x = act_x1; x <= act_x2; x++) {
                location = (x + vinfo.xoffset) + (y + vinfo.yoffset) *
vinfo.xres;
```

```
                byte_location = location / 8; /* find the byte we need to change
*/
                bit_location = location % 8; /* inside the byte found, find the
bit we need to change */
                fbp8[byte_location] &= ~(((uint8_t)(1)) << bit_location);
                fbp8[byte_location] |= ((uint8_t)(color_p->full)) <<
bit_location;
                color_p++;
            }

            color_p += area->x2 - act_x2;
        }
    } else {
        /*Not supported bit per pixel*/
    }

    //May be some direct update command is required
    //ret = ioctl(state->fd, FBIO_UPDATE, (unsigned long)((uintptr_t)rect));

    lv_disp_flush_ready(drv);
}

void fbdev_get_sizes(uint32_t *width, uint32_t *height, uint32_t *dpi) {
    if (width)
        *width = vinfo.xres;

    if (height)
        *height = vinfo.yres;

    if (dpi && vinfo.height)
        *dpi = DIV_ROUND_UP(vinfo.xres * 254, vinfo.width * 10);
}

void fbdev_set_offset(uint32_t xoffset, uint32_t yoffset) {
    vinfo.xoffset = xoffset;
    vinfo.yoffset = yoffset;
}

/**********************
 *   STATIC FUNCTIONS
 **********************/

#endif

/**
 * @file evdev.c
 *
 */

/*********************
 *      INCLUDES
 *********************/
#include "evdev.h"
#if USE_EVDEV != 0 || USE_BSD_EVDEV

#include <stdio.h>
```

```c
#include <unistd.h>
#include <fcntl.h>
#if USE_BSD_EVDEV
#include <dev/evdev/input.h>
#else
#include <linux/input.h>
#endif

#if USE_XKB
#include "xkb.h"
#endif /* USE_XKB */

/**********************
 *      DEFINES
 **********************/

/**********************
 *      TYPEDEFS
 **********************/

/**********************
 *  STATIC PROTOTYPES
 **********************/
int map(int x, int in_min, int in_max, int out_min, int out_max);

/**********************
 *  STATIC VARIABLES
 **********************/
int evdev_fd = -1;
int evdev_root_x;
int evdev_root_y;
int evdev_button;

int evdev_key_val;

/**********************
 *      MACROS
 **********************/

/**********************
 *   GLOBAL FUNCTIONS
 **********************/

/**
 * Initialize the evdev interface
 */
void evdev_init(void)
{
    if (!evdev_set_file(EVDEV_NAME)) {
        return;
    }

#if USE_XKB
    xkb_init();
#endif
}
```

```
/**
 * reconfigure the device file for evdev
 * @param dev_name set the evdev device filename
 * @return true: the device file set complete
 *         false: the device file doesn't exist current system
 */
bool evdev_set_file(char* dev_name)
{
     if(evdev_fd != -1) {
        close(evdev_fd);
     }
#if USE_BSD_EVDEV
     evdev_fd = open(dev_name, O_RDWR | O_NOCTTY);
#else
     evdev_fd = open(dev_name, O_RDWR | O_NOCTTY | O_NDELAY);
#endif

     if(evdev_fd == -1) {
        perror("unable to open evdev interface:");
        return false;
     }

#if USE_BSD_EVDEV
     fcntl(evdev_fd, F_SETFL, O_NONBLOCK);
#else
     fcntl(evdev_fd, F_SETFL, O_ASYNC | O_NONBLOCK);
#endif

     evdev_root_x = 0;
     evdev_root_y = 0;
     evdev_key_val = 0;
     evdev_button = LV_INDEV_STATE_REL;

     return true;
}
/**
 * Get the current position and state of the evdev
 * @param data store the evdev data here
 */
void evdev_read(lv_indev_drv_t * drv, lv_indev_data_t * data)
{
    struct input_event in;

    while(read(evdev_fd, &in, sizeof(struct input_event)) > 0) {
        if(in.type == EV_REL) {
            if(in.code == REL_X)
                #if EVDEV_SWAP_AXES
                    evdev_root_y += in.value;
                #else
                    evdev_root_x += in.value;
                #endif
            else if(in.code == REL_Y)
                #if EVDEV_SWAP_AXES
                    evdev_root_x += in.value;
                #else
                    evdev_root_y += in.value;
```

```
                #endif
        } else if(in.type == EV_ABS) {
            if(in.code == ABS_X)
                #if EVDEV_SWAP_AXES
                    evdev_root_y = in.value;
                #else
                    evdev_root_x = in.value;
                #endif
            else if(in.code == ABS_Y)
                #if EVDEV_SWAP_AXES
                    evdev_root_x = in.value;
                #else
                    evdev_root_y = in.value;
                #endif
            else if(in.code == ABS_MT_POSITION_X)
                                #if EVDEV_SWAP_AXES
                                        evdev_root_y = in.value;
                                #else
                                        evdev_root_x = in.value;
                                #endif
            else if(in.code == ABS_MT_POSITION_Y)
                                #if EVDEV_SWAP_AXES
                                        evdev_root_x = in.value;
                                #else
                                        evdev_root_y = in.value;
                                #endif
            else if(in.code == ABS_MT_TRACKING_ID) {
                                if(in.value == -1)
                                    evdev_button = LV_INDEV_STATE_REL;
                                else if(in.value == 0)
                                    evdev_button = LV_INDEV_STATE_PR;
            }
        } else if(in.type == EV_KEY) {
            if(in.code == BTN_MOUSE || in.code == BTN_TOUCH) {
                if(in.value == 0)
                    evdev_button = LV_INDEV_STATE_REL;
                else if(in.value == 1)
                    evdev_button = LV_INDEV_STATE_PR;
            } else if(drv->type == LV_INDEV_TYPE_KEYPAD) {
#if USE_XKB
                data->key = xkb_process_key(in.code, in.value != 0);
#else
                switch(in.code) {
                    case KEY_BACKSPACE:
                        data->key = LV_KEY_BACKSPACE;
                        break;
                    case KEY_ENTER:
                        data->key = LV_KEY_ENTER;
                        break;
                    case KEY_PREVIOUS:
                        data->key = LV_KEY_PREV;
                        break;
                    case KEY_NEXT:
                        data->key = LV_KEY_NEXT;
                        break;
                    case KEY_UP:
```

```
                        data->key = LV_KEY_UP;
                        break;
                    case KEY_LEFT:
                        data->key = LV_KEY_LEFT;
                        break;
                    case KEY_RIGHT:
                        data->key = LV_KEY_RIGHT;
                        break;
                    case KEY_DOWN:
                        data->key = LV_KEY_DOWN;
                        break;
                    case KEY_TAB:
                        data->key = LV_KEY_NEXT;
                        break;
                    default:
                        data->key = 0;
                        break;
                }
#endif /* USE_XKB */
                if (data->key != 0) {
                    /* Only record button state when actual output is produced to
prevent widgets from refreshing */
                    data->state = (in.value) ? LV_INDEV_STATE_PR :
LV_INDEV_STATE_REL;
                }
                evdev_key_val = data->key;
                evdev_button = data->state;
                return;
            }
        }
    }

    if(drv->type == LV_INDEV_TYPE_KEYPAD) {
        /* No data retrieved */
        data->key = evdev_key_val;
        data->state = evdev_button;
        return;
    }
    if(drv->type != LV_INDEV_TYPE_POINTER)
        return ;
    /*Store the collected data*/

#if EVDEV_CALIBRATE
    data->point.x = map(evdev_root_x, EVDEV_HOR_MIN, EVDEV_HOR_MAX, 0, drv->disp-
>driver->hor_res);
    data->point.y = map(evdev_root_y, EVDEV_VER_MIN, EVDEV_VER_MAX, 0, drv->disp-
>driver->ver_res);
#else
    data->point.x = evdev_root_x;
    data->point.y = evdev_root_y;
#endif

    data->state = evdev_button;

    if(data->point.x < 0)
      data->point.x = 0;
```

```
    if(data->point.y < 0)
      data->point.y = 0;
    if(data->point.x >= drv->disp->driver->hor_res)
      data->point.x = drv->disp->driver->hor_res - 1;
    if(data->point.y >= drv->disp->driver->ver_res)
      data->point.y = drv->disp->driver->ver_res - 1;

    return ;
}

/***********************
 *   STATIC FUNCTIONS
 ***********************/
int map(int x, int in_min, int in_max, int out_min, int out_max)
{
  return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
}

#endif

/**
 * @file libinput.h
 *
 */

#ifndef LVGL_LIBINPUT_H
#define LVGL_LIBINPUT_H

#ifdef __cplusplus
extern "C" {
#endif

/***********************
 *      INCLUDES
 ***********************/
#ifndef LV_DRV_NO_CONF
#ifdef LV_CONF_INCLUDE_SIMPLE
#include "lv_drv_conf.h"
#else
#include "../../lv_drv_conf.h"
#endif
#endif

#if USE_LIBINPUT || USE_BSD_LIBINPUT

#ifdef LV_LVGL_H_INCLUDE_SIMPLE
#include "lvgl.h"
#else
#include "lvgl/lvgl.h"
#endif

#include <poll.h>

#if USE_XKB
#include "xkb.h"
#endif /* USE_XKB */
```

```
/*********************
 *      DEFINES
 *********************/

/**********************
 *      TYPEDEFS
 **********************/
typedef enum {
  LIBINPUT_CAPABILITY_NONE     = 0,
  LIBINPUT_CAPABILITY_KEYBOARD = 1U << 0,
  LIBINPUT_CAPABILITY_POINTER  = 1U << 1,
  LIBINPUT_CAPABILITY_TOUCH    = 1U << 2
} libinput_capability;

typedef struct {
  int fd;
  struct pollfd fds[1];

  int button;
  int key_val;
  lv_point_t most_recent_touch_point;

  struct libinput *libinput_context;
  struct libinput_device *libinput_device;

#if USE_XKB
  xkb_drv_state_t xkb_state;
#endif /* USE_XKB */
} libinput_drv_state_t;

/**********************
 * GLOBAL PROTOTYPES
 **********************/

/**
 * find connected input device with specific capabilities
 * @param capabilities required device capabilities
 * @param force_rescan erase the device cache (if any) and rescan the file system
for available devices
 * @return device node path (e.g. /dev/input/event0) for the first matching
device or NULL if no device was found.
 *         The pointer is safe to use until the next forceful device search.
 */
char *libinput_find_dev(libinput_capability capabilities, bool force_rescan);
/**
 * find connected input devices with specific capabilities
 * @param capabilities required device capabilities
 * @param devices pre-allocated array to store the found device node paths (e.g.
/dev/input/event0). The pointers are
 *                safe to use until the next forceful device search.
 * @param count maximum number of devices to find (the devices array should be at
least this long)
 * @param force_rescan erase the device cache (if any) and rescan the file system
for available devices
 * @return number of devices that were found
```

```
 */
size_t libinput_find_devs(libinput_capability capabilities, char **found, size_t
count, bool force_rescan);
/**
 * Prepare for reading input via libinput using the default driver state. Use
this function if you only want
 * to connect a single device.
 */
void libinput_init(void);
/**
 * Prepare for reading input via libinput using a specific driver state. Use this
function if you want to
 * connect multiple devices.
 * @param state driver state to initialize
 * @param path input device node path (e.g. /dev/input/event0)
 */
void libinput_init_state(libinput_drv_state_t *state, char* path);
/**
 * Reconfigure the device file for libinput using the default driver state. Use
this function if you only want
 * to connect a single device.
 * @param dev_name input device node path (e.g. /dev/input/event0)
 * @return true: the device file set complete
 *         false: the device file doesn't exist current system
 */
bool libinput_set_file(char* dev_name);
/**
 * Reconfigure the device file for libinput using a specific driver state. Use
this function if you want to
 * connect multiple devices.
 * @param state the driver state to configure
 * @param dev_name input device node path (e.g. /dev/input/event0)
 * @return true: the device file set complete
 *         false: the device file doesn't exist current system
 */
bool libinput_set_file_state(libinput_drv_state_t *state, char* dev_name);
/**
 * Read available input events via libinput using the default driver state. Use
this function if you only want
 * to connect a single device.
 * @param indev_drv driver object itself
 * @param data store the libinput data here
 */
void libinput_read(lv_indev_drv_t * indev_drv, lv_indev_data_t * data);
/**
 * Read available input events via libinput using a specific driver state. Use
this function if you want to
 * connect multiple devices.
 * @param state the driver state to use
 * @param indev_drv driver object itself
 * @param data store the libinput data here
 */
void libinput_read_state(libinput_drv_state_t * state, lv_indev_drv_t *
indev_drv, lv_indev_data_t * data);

/**********************
```

```
 *      MACROS
 **********************/

#endif /* USE_LIBINPUT || USE_BSD_LIBINPUT */

#ifdef __cplusplus
} /* extern "C" */
#endif

#endif /* LVGL_LIBINPUT_H */

/**
 * @file libinput.c
 *
 */

/*********************
 *      INCLUDES
 *********************/
#include "libinput_drv.h"
#if USE_LIBINPUT || USE_BSD_LIBINPUT

#include <stdio.h>
#include <unistd.h>
#include <linux/limits.h>
#include <fcntl.h>
#include <errno.h>
#include <stdbool.h>
#include <dirent.h>
#include <libinput.h>

#if USE_BSD_LIBINPUT
#include <dev/evdev/input.h>
#else
#include <linux/input.h>
#endif

/*********************
 *      DEFINES
 *********************/

/**********************
 *      TYPEDEFS
 **********************/
struct input_device {
  libinput_capability capabilities;
  char *path;
};

/**********************
 *  STATIC PROTOTYPES
 **********************/
static bool rescan_devices(void);
static bool add_scanned_device(char *path, libinput_capability capabilities);
static void reset_scanned_devices(void);
```

```
static void read_pointer(libinput_drv_state_t *state, struct libinput_event
*event);
static void read_keypad(libinput_drv_state_t *state, struct libinput_event
*event);

static int open_restricted(const char *path, int flags, void *user_data);
static void close_restricted(int fd, void *user_data);

/**********************
 *  STATIC VARIABLES
 **********************/
static struct input_device *devices = NULL;
static size_t num_devices = 0;

static libinput_drv_state_t default_state = { .most_recent_touch_point = { .x =
0, .y = 0 } };

static const int timeout = 0; // do not block
static const nfds_t nfds = 1;

static const struct libinput_interface interface = {
  .open_restricted = open_restricted,
  .close_restricted = close_restricted,
};

/**********************
 *      MACROS
 **********************/

/**********************
 *   GLOBAL FUNCTIONS
 **********************/

/**
 * find connected input device with specific capabilities
 * @param capabilities required device capabilities
 * @param force_rescan erase the device cache (if any) and rescan the file system
for available devices
 * @return device node path (e.g. /dev/input/event0) for the first matching
device or NULL if no device was found.
 *         The pointer is safe to use until the next forceful device search.
 */
char *libinput_find_dev(libinput_capability capabilities, bool force_rescan) {
  char *path = NULL;
  libinput_find_devs(capabilities, &path, 1, force_rescan);
  return path;
}

/**
 * find connected input devices with specific capabilities
 * @param capabilities required device capabilities
 * @param devices pre-allocated array to store the found device node paths (e.g.
/dev/input/event0). The pointers are
 *                safe to use until the next forceful device search.
 * @param count maximum number of devices to find (the devices array should be at
least this long)
```

```
 * @param force_rescan erase the device cache (if any) and rescan the file system
for available devices
 * @return number of devices that were found
 */
size_t libinput_find_devs(libinput_capability capabilities, char **found, size_t
count, bool force_rescan) {
  if ((!devices || force_rescan) && !rescan_devices()) {
    return 0;
  }

  size_t num_found = 0;

  for (size_t i = 0; i < num_devices && num_found < count; ++i) {
    if (devices[i].capabilities & capabilities) {
      found[num_found] = devices[i].path;
      num_found++;
    }
  }

  return num_found;
}

/**
 * Reconfigure the device file for libinput using the default driver state. Use
this function if you only want
 * to connect a single device.
 * @param dev_name input device node path (e.g. /dev/input/event0)
 * @return true: the device file set complete
 *         false: the device file doesn't exist current system
 */
bool libinput_set_file(char* dev_name)
{
  return libinput_set_file_state(&default_state, dev_name);
}

/**
 * Reconfigure the device file for libinput using a specific driver state. Use
this function if you want to
 * connect multiple devices.
 * @param state the driver state to configure
 * @param dev_name input device node path (e.g. /dev/input/event0)
 * @return true: the device file set complete
 *         false: the device file doesn't exist current system
 */
bool libinput_set_file_state(libinput_drv_state_t *state, char* dev_name)
{
  // This check *should* not be necessary, yet applications crashes even on NULL
handles.
  // citing libinput.h:libinput_path_remove_device:
  // > If no matching device exists, this function does nothing.
  if (state->libinput_device) {
    state->libinput_device = libinput_device_unref(state->libinput_device);
    libinput_path_remove_device(state->libinput_device);
  }
```

```
  state->libinput_device = libinput_path_add_device(state->libinput_context,
dev_name);
  if(!state->libinput_device) {
    perror("unable to add device to libinput context:");
    return false;
  }
  state->libinput_device = libinput_device_ref(state->libinput_device);
  if(!state->libinput_device) {
    perror("unable to reference device within libinput context:");
    return false;
  }

  state->button = LV_INDEV_STATE_REL;
  state->key_val = 0;

  return true;
}

/**
 * Prepare for reading input via libinput using the default driver state. Use
this function if you only want
 * to connect a single device.
 */
void libinput_init(void)
{
  libinput_init_state(&default_state, LIBINPUT_NAME);
}

/**
 * Prepare for reading input via libinput using the a specific driver state. Use
this function if you want to
 * connect multiple devices.
 * @param state driver state to initialize
 * @param path input device node path (e.g. /dev/input/event0)
 */
void libinput_init_state(libinput_drv_state_t *state, char* path)
{
  state->libinput_device = NULL;
  state->libinput_context = libinput_path_create_context(&interface, NULL);

  if(path == NULL || !libinput_set_file_state(state, path)) {
      fprintf(stderr, "unable to add device \"%s\" to libinput context: %s\n",
path ? path : "NULL", strerror(errno));
      return;
  }
  state->fd = libinput_get_fd(state->libinput_context);

  /* prepare poll */
  state->fds[0].fd = state->fd;
  state->fds[0].events = POLLIN;
  state->fds[0].revents = 0;

#if USE_XKB
  xkb_init_state(&(state->xkb_state));
#endif
}
```

```
/**
 * Read available input events via libinput using the default driver state. Use
this function if you only want
 * to connect a single device.
 * @param indev_drv driver object itself
 * @param data store the libinput data here
 */
void libinput_read(lv_indev_drv_t * indev_drv, lv_indev_data_t * data)
{
  libinput_read_state(&default_state, indev_drv, data);
}

/**
 * Read available input events via libinput using a specific driver state. Use
this function if you want to
 * connect multiple devices.
 * @param state the driver state to use
 * @param indev_drv driver object itself
 * @param data store the libinput data here
 */
void libinput_read_state(libinput_drv_state_t * state, lv_indev_drv_t *
indev_drv, lv_indev_data_t * data)
{
  struct libinput_event *event;
  int rc = 0;

  rc = poll(state->fds, nfds, timeout);
  switch (rc){
    case -1:
      perror(NULL);
    case 0:
      goto report_most_recent_state;
    default:
      break;
  }
  libinput_dispatch(state->libinput_context);
  while((event = libinput_get_event(state->libinput_context)) != NULL) {
    switch (indev_drv->type) {
      case LV_INDEV_TYPE_POINTER:
        read_pointer(state, event);
        break;
      case LV_INDEV_TYPE_KEYPAD:
        read_keypad(state, event);
        break;
      default:
        break;
    }
    libinput_event_destroy(event);
  }
report_most_recent_state:
  data->point.x = state->most_recent_touch_point.x;
  data->point.y = state->most_recent_touch_point.y;
  data->state = state->button;
  data->key = state->key_val;
}
```

```
/***********************
 *   STATIC FUNCTIONS
 ***********************/

/**
 * rescan all attached evdev devices and store capable ones into the static
devices array for quick later filtering
 * @return true if the operation succeeded
 */
static bool rescan_devices(void) {
  reset_scanned_devices();

  DIR *dir;
  struct dirent *ent;
  if (!(dir = opendir("/dev/input"))) {
    perror("unable to open directory /dev/input");
    return false;
  }

  struct libinput *context = libinput_path_create_context(&interface, NULL);

  while ((ent = readdir(dir))) {
    if (strncmp(ent->d_name, "event", 5) != 0) {
      continue;
    }

    /* 11 characters for /dev/input/ + length of name + 1 NUL terminator */
    char *path = malloc((11 + strlen(ent->d_name) + 1) * sizeof(char));
    if (!path) {
      perror("could not allocate memory for device node path");
      libinput_unref(context);
      reset_scanned_devices();
      return false;
    }
    strcpy(path, "/dev/input/");
    strcat(path, ent->d_name);

    struct libinput_device *device = libinput_path_add_device(context, path);
    if(!device) {
      perror("unable to add device to libinput context");
      free(path);
      continue;
    }

    /* The device pointer is guaranteed to be valid until the next
libinput_dispatch. Since we're not dispatching events
     * as part of this function, we don't have to increase its reference count to
keep it alive.
     *
https://wayland.freedesktop.org/libinput/doc/latest/api/group__base.html#gaa79749
6f0150b482a4e01376bd33a47b */

    libinput_capability capabilities = LIBINPUT_CAPABILITY_NONE;
    if (libinput_device_has_capability(device, LIBINPUT_DEVICE_CAP_KEYBOARD)
```

```
        && (libinput_device_keyboard_has_key(device, KEY_ENTER) ||
libinput_device_keyboard_has_key(device, KEY_KPENTER)))
    {
      capabilities |= LIBINPUT_CAPABILITY_KEYBOARD;
    }
    if (libinput_device_has_capability(device, LIBINPUT_DEVICE_CAP_POINTER)) {
      capabilities |= LIBINPUT_CAPABILITY_POINTER;
    }
    if (libinput_device_has_capability(device, LIBINPUT_DEVICE_CAP_TOUCH)) {
      capabilities |= LIBINPUT_CAPABILITY_TOUCH;
    }

    libinput_path_remove_device(device);

    if (capabilities == LIBINPUT_CAPABILITY_NONE) {
      free(path);
      continue;
    }

    if (!add_scanned_device(path, capabilities)) {
      free(path);
      libinput_unref(context);
      reset_scanned_devices();
      return false;
    }
  }

  libinput_unref(context);
  return true;
}

/**
 * add a new scanned device to the static devices array, growing its size when
necessary
 * @param path device file path
 * @param capabilities device input capabilities
 * @return true if the operation succeeded
 */
static bool add_scanned_device(char *path, libinput_capability capabilities) {
  /* Double array size every 2^n elements */
  if ((num_devices & (num_devices + 1)) == 0) {
      struct input_device *tmp = realloc(devices, (2 * num_devices + 1) *
sizeof(struct input_device));
      if (!tmp) {
        perror("could not reallocate memory for devices array");
        return false;
      }
      devices = tmp;
  }

  devices[num_devices].path = path;
  devices[num_devices].capabilities = capabilities;
  num_devices++;

  return true;
}
```

```
/**
 * reset the array of scanned devices and free any dynamically allocated memory
 */
static void reset_scanned_devices(void) {
  if (!devices) {
    return;
  }

  for (size_t i = 0; i < num_devices; ++i) {
    free(devices[i].path);
  }
  free(devices);

  devices = NULL;
  num_devices = 0;
}

/**
 * Handle libinput touch / pointer events
 * @param state driver state to use
 * @param event libinput event
 */
static void read_pointer(libinput_drv_state_t *state, struct libinput_event
*event) {
  struct libinput_event_touch *touch_event = NULL;
  struct libinput_event_pointer *pointer_event = NULL;
  enum libinput_event_type type = libinput_event_get_type(event);

  /* We need to read unrotated display dimensions directly from the driver
because libinput won't account
   * for any rotation inside of LVGL */
  lv_disp_drv_t *drv = lv_disp_get_default()->driver;

  switch (type) {
    case LIBINPUT_EVENT_TOUCH_MOTION:
    case LIBINPUT_EVENT_TOUCH_DOWN:
      touch_event = libinput_event_get_touch_event(event);
      lv_coord_t x = libinput_event_touch_get_x_transformed(touch_event, drv-
>physical_hor_res > 0 ? drv->physical_hor_res : drv->hor_res) - drv->offset_x;
      lv_coord_t y = libinput_event_touch_get_y_transformed(touch_event, drv-
>physical_ver_res > 0 ? drv->physical_ver_res : drv->ver_res) - drv->offset_y;
      if (x < 0 || x > drv->hor_res || y < 0 || y > drv->ver_res) {
        break; /* ignore touches that are out of bounds */
      }
      state->most_recent_touch_point.x = x;
      state->most_recent_touch_point.y = y;
      state->button = LV_INDEV_STATE_PR;
      break;
    case LIBINPUT_EVENT_TOUCH_UP:
      state->button = LV_INDEV_STATE_REL;
      break;
    case LIBINPUT_EVENT_POINTER_MOTION:
      pointer_event = libinput_event_get_pointer_event(event);
      state->most_recent_touch_point.x +=
libinput_event_pointer_get_dx(pointer_event);
```

```
      state->most_recent_touch_point.y +=
libinput_event_pointer_get_dy(pointer_event);
      state->most_recent_touch_point.x = LV_CLAMP(0, state-
>most_recent_touch_point.x, drv->hor_res - 1);
      state->most_recent_touch_point.y = LV_CLAMP(0, state-
>most_recent_touch_point.y, drv->ver_res - 1);
      break;
    case LIBINPUT_EVENT_POINTER_BUTTON:
      pointer_event = libinput_event_get_pointer_event(event);
      enum libinput_button_state button_state =
libinput_event_pointer_get_button_state(pointer_event);
      state->button = button_state == LIBINPUT_BUTTON_STATE_RELEASED ?
LV_INDEV_STATE_REL : LV_INDEV_STATE_PR;
      break;
    default:
      break;
  }
}

/**
 * Handle libinput keyboard events
 * @param state driver state to use
 * @param event libinput event
 */
static void read_keypad(libinput_drv_state_t *state, struct libinput_event
*event) {
  struct libinput_event_keyboard *keyboard_event = NULL;
  enum libinput_event_type type = libinput_event_get_type(event);
  switch (type) {
    case LIBINPUT_EVENT_KEYBOARD_KEY:
      keyboard_event = libinput_event_get_keyboard_event(event);
      enum libinput_key_state key_state =
libinput_event_keyboard_get_key_state(keyboard_event);
      uint32_t code = libinput_event_keyboard_get_key(keyboard_event);
#if USE_XKB
      state->key_val = xkb_process_key_state(&(state->xkb_state), code, key_state
== LIBINPUT_KEY_STATE_PRESSED);
#else
      switch(code) {
        case KEY_BACKSPACE:
          state->key_val = LV_KEY_BACKSPACE;
          break;
        case KEY_ENTER:
          state->key_val = LV_KEY_ENTER;
          break;
        case KEY_PREVIOUS:
          state->key_val = LV_KEY_PREV;
          break;
        case KEY_NEXT:
          state->key_val = LV_KEY_NEXT;
          break;
        case KEY_UP:
          state->key_val = LV_KEY_UP;
          break;
        case KEY_LEFT:
          state->key_val = LV_KEY_LEFT;
```

```
          break;
        case KEY_RIGHT:
          state->key_val = LV_KEY_RIGHT;
          break;
        case KEY_DOWN:
          state->key_val = LV_KEY_DOWN;
          break;
        case KEY_TAB:
          state->key_val = LV_KEY_NEXT;
          break;
        default:
          state->key_val = 0;
          break;
      }
#endif /* USE_XKB */
      if (state->key_val != 0) {
        /* Only record button state when actual output is produced to prevent
widgets from refreshing */
        state->button = (key_state == LIBINPUT_KEY_STATE_RELEASED) ?
LV_INDEV_STATE_REL : LV_INDEV_STATE_PR;
      }
      break;
    default:
      break;
  }
}

static int open_restricted(const char *path, int flags, void *user_data)
{
  LV_UNUSED(user_data);
  int fd = open(path, flags);
  return fd < 0 ? -errno : fd;
}

static void close_restricted(int fd, void *user_data)
{
  LV_UNUSED(user_data);
  close(fd);
}

#endif /* USE_LIBINPUT || USE_BSD_LIBINPUT */
```