LCD Touch Lib V2

Daniel Casini, 463919, MECS

November 7, 2015

Contents

List of Figures

Listings

1 LCD Touch Lib

Before going in the details of the code generation, in this section the LCD Touch Library is presented. The library has been developed in order to abstract the low level functions provided by *element14*, thus allowing to generate simplier code. The library can be divided in two main parts:

- Screen (with layers) management
- Touch management

1.1 Screen management

Developing the LCD part of the library it is necessary to decide which kind of widgets are necessary to implement. In this project have been developed only the widgets needed to implement the **Fuel Station Case Study**, (reported at the end of this document) but in any case it is easy to extend it implementing whatever kind of widget, using the structure explained in this section. The widgets implemented in the library are:

- (Image) Buttons
- Textlines
- Labels
- Screen pages

An important aspect to take into account is whether to allow superimposition among widgets, thus creating the needs for a **layered structure**. In order to implement it, the data structure that has been used is a **tree**. In particular, since it is possibile to create more pages (and to switch among them), *an array of trees have been defined*, each one with a **LCD Page widget** as root. Obviously, each page can contain an undefined number of childs, so the trees are *generic trees*, implemented as **left-child right-sibling binary trees**. In order to clarify this point, in Fig. 1 and 2 an example is reported, representing both the logical structure and how it is implemented.

Hence, when the *drawWidget()* method is called with a particular widget as an argument, the whole subtree with the given widget as root is re-drawn. Referring the previous example, calling the function **drawWidget(button1)**, both Button1 and Label2 (that is part of the Button1 subtree) are re-drawn.

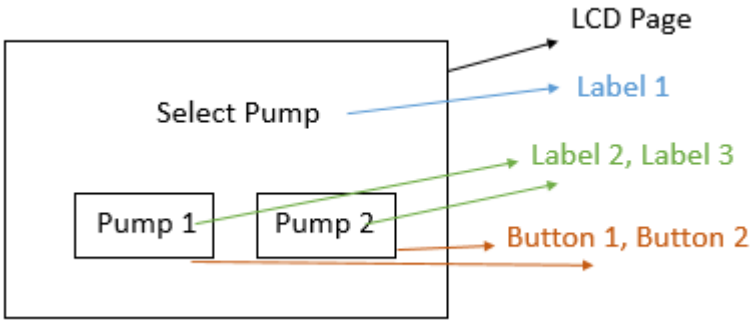

Figure 1: LCD Layer Implementation Example

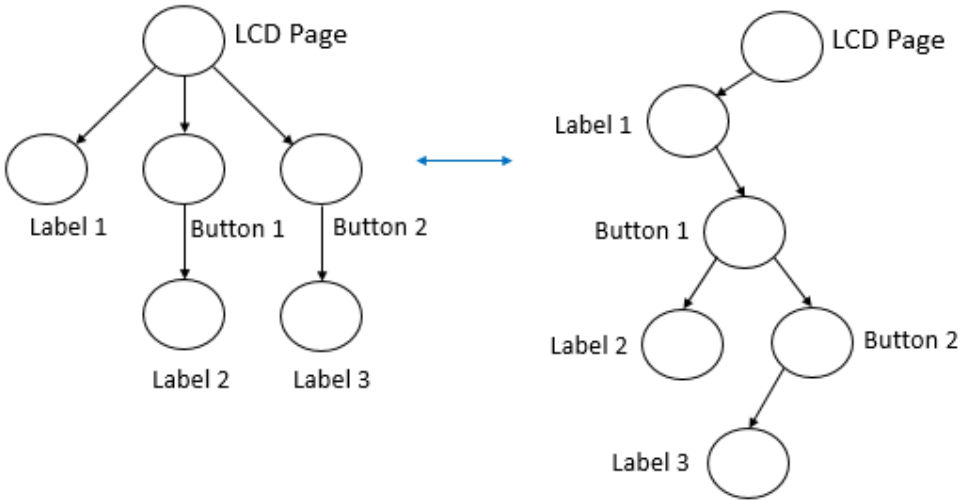

Figure 2: LCD Layer Logical - Real Tree

Let us analyze now the data structure used to represent a widget. In the LCD_Touch_Lib, all the widgets are represented by the following fields:

- Widget type
- Touch area bounds, i.e., the x and y coordinates of the top left and bottom right corners of the widget bounding box
- A global widget id
- A pointer to the **particular data structure** associated to a specific type of widget (e.g. BUTTON data for buttons, TEXTLINE data for textlines, etc)

- A pointer to the function handler of the widget (0 if no touch event is associated with the widget)
- A pointer to the function handler argument

Listing 1: Widget Data Structure

```
1 typedef struct _widget {
2     WIDGET_TYPE     wtype;
3     TOUCH_AREA  bounds;
4     uint8_t     id;
5     void        *wdata;
6     HANDLER     handler;
7     void        *handler_arg;
8 } WIDGET;
```

As already stated, in addition to this general fields, every type of widget needs to store particular informations. Hence, for each type of widget, a data structure has been defined to contain them.

1.1.1 LCD Page

The informations stored for this type of widget are:

- The background type (color or image)
- An union that contains either the background color or a pointer to the background image

Listing 2: LCD Page Data Structure

```
typedef union _background
{
    uint16_t color;
    uint8_t* image;
} BACKGROUND;

typedef enum _backgroundType{
    BACKGROUND_COLOR,
    BACKGROUND_IMAGE,
    NUM_BACKGROUND_TYPES
}BACKGROUND_TYPE;

typedef struct lcd_page_background {
    BACKGROUND      background;
    BACKGROUND_TYPE    background_type;
} PAGE_BACKGROUND;

typedef struct lcd_page {
    PAGE_BACKGROUND    backgroundObj;
    BACKGROUND_TYPE    background_type;
} LCD_PAGE;
```

1.1.2 Button

As far the button widget is concerned, it is necessary to store the pointer to the **widget icon**. If the icon is not specified, the button becomes only a touch area linked to a function handler (if defined).

Listing 3: Button Data Structure

```
typedef struct touch_button {
    uint8_t* icon;
} BUTTON;
```

1.1.3 Textline

A textline is a widget that allows to print and modify a text on the screen. It is composed by a background rectangle and a text. The information to store for this widget are:

- Rectangle background color
- Rectangle border color
- Text color
- Length (expressed as maximum number of characters) of the textline
- Font object pointer
- Text pointer (the array of characters has to be allocated by the user)

Listing 4: Textline Data Structure

```
1 typedef struct _textline{
2     uint16_t background_color;
3     uint16_t border_color;
4     uint16_t text_color;
5     uint16_t textline_len;
6     sFONT   *font;
7     char    *textline;
8 } TEXT_LINE;
```

1.1.4 Label

A label is intended as a line with fixed text. Differently from a textline, a label does not have a background. Hence, the structure associated to this kind of widget includes:

- Text color
- Font object pointer
- Text

Listing 5: Label Data Structure

```
1 typedef struct _label{
2     uint16_t text_color;
3     sFONT   *font;
4     char    text[MAX_LABEL_LEN];
5 } LABEL;
```

1.2 Touch Management

As far the touchscreen is concerned, the main functions used to obtain a screen cordinate, whenever a touch event is detected, are taken from a previous version of this library, implemented by me in another course (Embedded System Design) of the Master Degree in Embedded Computing System. The touchscreen is managed with the **polling mechanism**. Differently from the previous version, in which the detected events were immediately handled once detected by calling the associated function handler (if any), in this case the detected touch events can be enqueue in a **circular queue** and dispatched later. As stated in the introduction, the library has been designed in order to be used together with the **Erika Operating System** Hence, the polling and the dispatch procedure can be separated in two separated tasks or, as it has been done in the final example, the polling procedure can be demanded to the background activity of the main program, whereas the dispatch procedure can be managed by a specific task. The data structure used to represent the *circular queue* is shown in Listing. 6.

Listing 6: Events Queue Data Structure

```
1 typedef struct _event_queue{
2     uint8_t widgetID[MAX_EVENTS_ENQUEUED];
3     uint8_t counter;
4     uint8_t read_index;
5     uint8_t write_index;
6 } EVENT_QUEUE_T;
```

However, it is important to notice that using this implementation the *'polling'* task and the *'dispatcher'* task perform two concurrent activities. For this reason, a *'Resource'* (a mutual exclusion semaphore) provided by the Erika OS has been used to avoid race conditions. In Fig. 3 it is possible to see how the queue is managed by the touchscreen tasks.

In particular, the *enqueueTouchEvent(POINT detected)* takes the point detected by the touch driver as argument and then search in the tree that has the **active page** as root. The tree is explored until the **inner most** object that contains the detected coordinate (and owns an associated function handler) is found. Once the widget id of the right object has been found, it is enqueued in the touch event queue. As far the dispatchTouchEvents() procedure is concerned, it checks whether there are events in the queue and in this case calls the appropriate function handler. In Listing. 7 and 8 is reported the code of these functions.

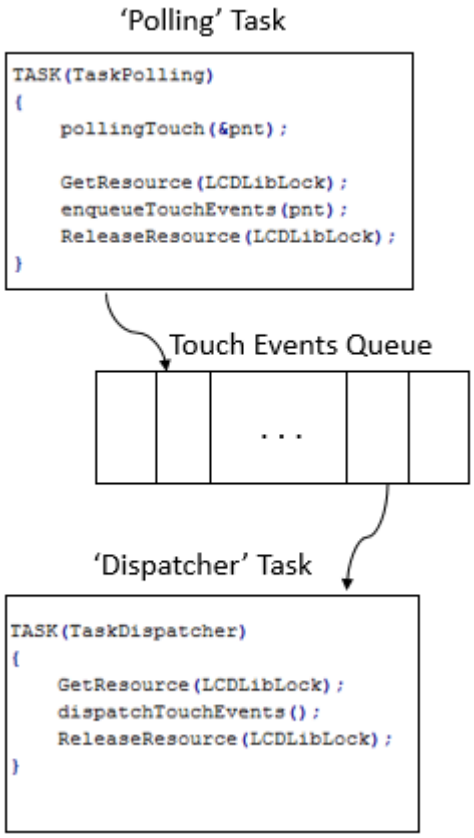

Figure 3: Touch Event Queue Management

Listing 7: enqueueTouchEvents()

```
1
2 void enqueueTouchEvents(POINT detected){
3       WIDGET *widget;
4       WIDGET_DESC *desc;
5
6       desc = findPressedWidget(&detected);
7
8       if(desc != 0){
9           widget = desc->data;
10          if(eventQueue.counter != MAX_EVENTS_ENQUEUED){
11              eventQueue.widgetID[eventQueue.write_index] =
                    widget->id;
12              eventQueue.write_index = (eventQueue.write_index
                    + 1) % MAX_EVENTS_ENQUEUED;
13              eventQueue.counter++;
14          }
15      }
16 }
```

Listing 8: dispatchTouchEvents()

```
1 void dispatchTouchEvents(){
2    uint8_t id;
3    if(eventQueue.counter != 0){
4        id = eventQueue.widgetID[eventQueue.read_index];
5        eventQueue.read_index = (eventQueue.read_index + 1) %
             MAX_EVENTS_ENQUEUED;
6        eventQueue.counter--;
7        if(widgetList[id].data->handler != 0)
8            (widgetList[id].data->handler)(widgetList[id].data->
                 handler_arg);
9    }
10 }
```

1.3 LCD Library Page Creation Example

In this section an example of how to create a page with the LCD Touch Library is reported. More detailed informations about how to use the implemented functions are reported in the **doxygen documentation**. As it is possibile to see in Listing 9 in order to create a widget the appropriate **'new'** method has to be called. Since multiple pages are allowed, the creation of a widget does not draws the widget on the screen. In order to draw a page, (or in general a widget), the **drawWidget()** method must be called (or the **drawCurrentPage()**, after having been set the correct page with the **setPage(PAGE_ID)** method). In each widget creational method it is possible to specify the 'father' object, intended as the child object in the page tree. If it is not specified, the father object is automatically detected from the widget coordinate, starting from the **current page tree**.
Hence, it is important to set the correct page before creating objects.

Listing 9: Page Creation Example

```
1   newPage(&insert_pinPageWidget, RGB_2_565(66, 65, 74));
2
3   setPage(INSERT_PINPAGE);
4
5   KEY1_button_arg = KEY1_BUTTON_TYPE,
6   newButton(&KEY1_button, 12, 62, 64, 36, (uint8_t *)
        button_black, insert_pinPage_buttonsHandler, (void *)&
        KEY1_button_arg, 0);
7
8   newLabel(&one_label, 36, 67, RGB_2_565(255, 255, 255), &
        Font16x24, "1", 0);
9
10  newTextLine(&pin_textline, 15, 21, 114, 33, LCD_LIB_BLACK,
        RGB_2_565(255, 255, 255), RGB_2_565(255, 255, 255), &
        Font8x12, text_pin_input, TEXT_PIN_LEN, 0);
```

1.4 LCD Library Scalability : How to add a new widget

First of all, it is important to summarize from which files the library is composed:

- **LCD_Lib_Types.h**, that contains all the data types needed in the library.
- **LCD_Lib_HWParam.h**, that contains the parameters related to the touchscreen calibration
- **LCD_Colors.h**, that contains the definition as symbolic name of some colors in the bitmap 565 format (that is used by the Discovery Board)
- **LCD_Lib_Conf.h**, tha contains the definition of the number of pages and widget (for each type)
- **LCD_Lib_Draw.h/c**, that contains the functions to draw single widgets (without taking care of the widget tree)
- **LCD_Lib_Layers.h/c**, that contains the functions that manages the widget tree, both for the LCD and the touch
- **LCD_Touch_Lib.h/c**, tha contains the higher level function that will be used directly by the library user

After having been introduced the library files, it is simply to understand the extend the library is very easy. In order to do this, it is necessary to:

- Create a new C struct that contains the specific data of the new widget type, and insert it in the LCD_Lib_Types.h file
- Create a new constant that express the maximum number of elements of the new widget type and insert it in LCD_Lib_Conf.h file
- Create a draw*NewWidgetTypeName*() method to draw the new widget type in LCD_Lib_Draw.h/c files
- Add a case in the switch/case statement contained in the drawSingleWidget() function, in LCD_Lib_Draw.h/c, after having been extended the WIDGET_TYPE enumeration in LCD_Lib_Types.h with the new type

- Create a new new*NewWidgetTypeName*() to create an object of the new widget type in LCD_Touch_Lib.h

The drawSingleWidget() function is reported in Listing 10.

Listing 10: DrawSingleWidget()

```
1  void drawSingleWidget(WIDGET_DESC* widget){
2     switch(widget->data->wtype){
3        case LCD_PAGE_TYPE:   drawLCDPage(widget);
4                              break;
5        case BUTTON_TYPE:  drawButton(widget);
6                              break;
7        case LABEL_TYPE:   drawLabel(widget);
8                              break;
9        case TEXTLINE_TYPE:   drawTextLine(widget);
10                             break;
11    }
12 }
```