Olsonet Communications

# PicOS
## LCD-G interface

Version 0.8
August 2008

## Preamble

This note describes the praxis interface to graphic LCD under PicOS. At the moment, the only LCD with this interface is Nokia 6100 equipped with Philips controller (PCF8833).

The present interface consists of two parts: basic operations provided by the driver and a library of functions implementing simple stacking of displayable objects on the screen (including menus).

## Basic operations

The LCD offers raw access to individual pixels with no built-in text functionality. A single pixel can be in at least two states, possibly more. The pixels are organized into a rectangular display area, with each pixel being (in principle) addressable and settable independently of other pixels. In particular, Nokia 6100 has 132x132 pixels. Each pixel can be set to an arbitrary 12-bit value representing a color. Boundary pixels are deemed unusable, i.e., for all practical purposes, the display consists of 130x130 pixels. The driver ignores the boundary lines of pixels (never touches them) assuming that the display is organized into 130 rows and 130 columns addressed from 0 to 129 (up to down and left to right. Other modes regarding the representation of colors are theoretically available (e.g., 8-bit colors). Those modes have been tried and discarded as not very useful.

Text operations are optional depending on the availability of fonts. To avoid overtaxing the code flash, fonts are stored in EEPROM. Thus, EEPROM is a required configuration option for text operations. The EEPROM must be preloaded with fonts, and the starting address of the font file in EEPROM must be made available to the driver. This is accomplished by setting the constant **`LCDG_FONT_BASE`** (typically in **`board_pins.h`**) to the respective EEPROM address. If this constant is not defined, text operations are not available.

```
void lcdg_on (byte con)
```

The operation switches on the display, i.e., the display will actually show its contents. When the display is off (see below), it will accept content-rendering commands, but the image will not show. For example, if the rendering operation is messy (and you would prefer not to show to the user how the individual pixels are being processed), you may switch the display off, render the image, and then switch the device on when everything is ready and nice.

The argument gives the contrast, with zero standing for the default value of 55. Feel free to try other values.

```
void lcdg_off ()
```

The operation switches the display off (see above). The display is off by default.

```
void lcdg_set (byte xl, byte yl, byte xh, byte yh)
```

The operations sets a bounding rectangle, which is typically intended as a container for some object to be displayed. For example, in order to render a picture or a text area, you have to set its bounding rectangle first. The arguments are coordinates of two points: **`xl`** and **`yl`** stand for the coordinates of the left upper corner, and **`xh`**, **`yh`** are the coordinates of the lower right corner. Note that the x coordinate grows left to right, and the y

coordinate grows top to bottom. This assumption is consistently followed in the rest of this document.

For example,

```
lcdg_set (0, 0, 129, 129);
```

sets the maximum possible bounding rectangle for Nokia 6100.

Two symbolic constants: **LCDG_MAXX** and **LCDG_MAXY** describe the maximum x and y coordinates allowed for the particular LCD device. They are 129/129 for Nokia 6100.

```
void lcdg_get (byte *xl, byte *yl, byte *xh, byte *yh)
```

The function returns via the arguments the current setting of the bounding rectangle. Any of the four pointers can be NULL, in which case the corresponding value will not be returned.

```
void lcdg_setc (byte bgr, byte fgr)
```

The operation sets the background and foreground colors, e.g., for text rendering. This is the list of colors (color numbers) that are currently available:

```
#define COLOR_WHITE     0
#define COLOR_BLACK     1
#define COLOR_RED       2
#define COLOR_GREEN     3
#define COLOR_BLUE      4
#define COLOR_CYAN      5
#define COLOR_MAGENTA   6
#define COLOR_YELLOW    7
#define COLOR_BROWN     8
#define COLOR_ORANGE    9
#define COLOR_PINK      10
```

If an argument to **lcd_setc** is not one of the above numbers, then it will be ignored, i.e., the corresponding setting will not change. The default colors are background = **COLOR_BLACK**, foreground = **COLOR_WHITE**.

```
void lcdg_clear ()
```

The function erases the current bounding rectangle by filling it with the background color.

```
void lcdg_render (byte c, byte r, const byte *pix, word n)
```

The function is normally called in the course of rendering a picture. It fills **n** pixels in the current bounding rectangle starting from the pixel with coordinates **c**, **r** relative to the left upper corner of the rectangle. The values to which those pixels are set are taken from **pix**, which should contain at least **n** values of the proper size. For Nokia 6100, two pixels take three bytes (two packed 12-bit values). If **n** is odd, the lower nibble of the last byte in **pix** is ignored.

Note that the operation automatically wraps around when attempting to cross the right edge of the bounding rectangle.



In the Seawolf project (for which the present interface has been implemented) pictures are stored (in EEPROM) as sequences of 54-byte "chunks" that need not occur in any particular order.[1] Each chunk specifies a pair of coordinates (two bytes describing the location of the starting pixel) and 52 packed bytes representing the values of up to 36 pixels. Such a chunk is naturally displayed by a simple call to **lcdg_render**.

```
word lcdg_font (byte f)
```

The function sets the font for a subsequent text rendering operation. The argument represents the font index, and its range corresponds to the number of fonts available in the font file in EEPROM. The operation fails when the specified font index is out of range, or if the font file doesn't exist. There are three fonts at present (or, rather, one font with three different sizes [sort of] labeled 0, 1, 2).

The present implementation only allows fixed-size fonts.

Note: there is no default font. If you want to render text, you have to issue **lcdg_font** beforehand.

The function returns zero on success and nonzero on failure.

```
byte lcdg_cwidth ()
byte lcdg_cheight ()
```

The functions return the width and height (in pixels) of the current font (note that all characters are the same width and height). The size includes spacing.

```
word lcdg_sett (byte x, byte y, byte nc, byte nl)
```

The operation can be viewed as a variant of **lcdg_set** to set up a bounding rectangle for a text. It requires a font definition, so it must be preceded by a call to **lcdg_font**. As for **lcdg_set**, the first two arguments are the coordinates of the left upper corner of the bounding rectangle. The remaining two arguments specify the number of characters in one line (**nc**) and the number of lines (**nl**). Based on the character size for the current font, the function calculates the pixel coordinates of the right lower corner and issues a pertinent call to **lcdg_set**. If needed, those coordinates can be then obtained by a call to **lcdg_get** (see above).

The function returns zero on success. It fails (returning nonzero) if the rectangle cannot be determined (because there is no font definition), or if the rectangle is too large (i.e., it doesn't entirely fit on the screen).

```
void lcdg_wl (const char *s, word sh, byte cx, byte cy)
```

The operation displays one line of text (described by string **s**) in the current bounding rectangle (e.g., created with a preceding call to **lcdg_sett**). The second argument is the *shift* count indicating the number of characters in **s** to be skipped from the left. The last two arguments are the *character* coordinates of the first displayed character, e.g., **cx** = 3, **cy** = 5 means that the first character will appear in the fourth character position counting from the left edge of the bounding rectangle, and in the sixth row (line) of the rectangle. Any characters that fall outside the rectangle are clipped out.

[1] This is described in a separate document.



If **sh** is greater than or equal to the number of characters in **s**, the operation is void. The role of this argument is to provide for horizontal scrolling whereby some initial portion of the line has been shifted out off the left edge of the bounding rectangle.

The displayed characters are rendered in the current foreground color (as defined by **lcdg_setc**) against the current background color.

Note that the function returns no value. If the text cannot be rendered (because no font is defined), its action is void.

```
void lcdg_ec (byte cx, byte cy, byte nc)
```

The function erases (i.e., fills with the background color) **nc** characters in the current bounding rectangle starting at character position **cx**, **cy** (coordinates as for **lcdg_wl**). The operation only affects one line of the area, i.e., it doesn't wrap past the right edge. Similar to **lcdg_wl**, it fails silently and does nothing if no font is defined.

```
void lcdg_el (byte cy, byte nl)
```

The function erases (i.e., fills with the background color) **nl** entire lines in the current bounding rectangle starting at line number **cy**. The operation never affects any pixels below the lower edge of the bounding rectangle, even if **nl** is larger than the rectangle's capacity. Similar to **lcdg_wl**, it fails silently and does nothing if no font is defined.

## Library functions

The functions discussed in this section allow you to represent a simple hierarchy of displayable objects, including pictures, text areas, as well as scrollable and selectable menus. You have to insert:

```
#include "dispman.h"
```

in the header section of your program to make them available to the praxis.

A displayable object is described by one of these types (declared in **dispman.h** in directory **PicOS/Libs/Lib**).

| | |
|---|---|
| **dimage_t** | a picture |
| **dtext_t** | a piece of text |
| **dmenu_t** | a menu |

These types are structures sharing a common initial chunk of data represented by type **dobject_t**. This means that it makes sense to cast any of the three specific object types to type **dobject_t** as long as you are only interested in the shared attributes of all displayable objects. Those shared attributes are:

```
byte            XL, YL, XH, YH;
dobject_t       *next;
byte            Type
```

The first four numbers describe the bounding rectangle of the object, attribute **next** is needed to organize multiple objects into a list, and **Type** can be one of **DOTYPE_IMAGE**, **DOTYPE_TEXT**, **DOTYPE_MENU** identifying the actual object type (and determining the layout of the remaining portion of the object).



At any time, the display manager knows the current hierarchy of objects to be shown on the screen. Those objects are described by a list (we shall call it the *display list*) headed at **DM_HEAD** (a global variable of type **dobject_t***). This list can be empty, in which case **DM_HEAD** contains NULL. Another global variable of the same type, **DM_TOP**, points to the last object on the display list, which is called the *top* (or *current*) object. **DM_TOP** is NULL if the display list happens to be empty.

The following three functions are used to create objects of the specific types. Note that a created object is not automatically displayed. Typically, **dm_top** (see below) is called after an object creation to show the object on the screen and make it the current (top) object.

```
dobject_t *dm_newimage (const char *lbl, byte x, byte y);
```

The function creates an image (picture) object. The first argument can be viewed as an identifier of the image: it is a string looked up in the image label. The function scans through the pictures stored in EEPROM and selects the first one whose label includes **lbl** as a substring. Then it creates an image object representing that picture. The last two arguments specify the left top corner of the picture's bounding rectangle.

The function returns the new object or, possibly, NULL indicating a failure. In the latter case, the reason for the failure is stored in global variable **DM_STATUS** of type **word**. This variable can take the following values (not all of them are applicable to pictures):

| | |
|---|---|
| **0** | no problem (last operation has succeeded) |
| **DERR_MEM** | memory allocation (umalloc) failure |
| **DERR_FMT** | format error |
| **DERR_NFN** | not found |
| **DERR_FNT** | font unavailable |
| **DERR_REC** | bounding rectangle out of screen |

For example, if there is no picture in EEPROM matching the specified label string, the function will return NULL, and **DM_STATUS** will be set to **DERR_NFN**. Similarly, if the sum of one of the corner coordinates (**x** or **y**) and the respective dimension of the picture (as stored in EEPROM) turns out to be greater than the screen size, the function will return NULL, and **DM_STATUS** will show **DERR_REC**.

Note that **umalloc** failures (**DERR_MEM**), if they happen at all, are usually intermittent, which means that it makes sense to retry the same operation later. Other failures are fatal and indicative of more fundamental problems.

```
dobject_t *dm_newtext (const char *str, byte font,
                byte bgr, byte fgr, byte x, byte y, byte w)
```

The function creates a text object. The first argument is the character string to be shown on the screen. Newlines, as well as any non-printable characters, are converted to blanks, i.e., the string is effectively treated as a single line. The next three arguments identify the font and two colors: background and foreground.

The left upper corner of the bounding rectangle for the text object is described by **x** and **y**. The width of that rectangle in characters is **w**, and its height accounts for the number of lines resulting from wrapping the specified string (as many times as needed) at the rectangle's right edge. For example, assuming that **w** is 12, this string “a quick brown fox jumps over the lazy dog” will produce 4 lines rendered like this:



```
a quick brown
fox jumps ove
r the lazy do
g
```

The possible failure codes for this operation are **DERR_MEM**, **DERR_FNT** (the specified font is not available), and **DERR_REC** (the resultant bounding rectangle does not fit on the screen).

The string passed to **dm_newtext** must not be deallocated for as long as the text object is supposed to leave (the object will point to the original rather than creating its private copy).

```
dobject_t *dm_newmenu (const char *lines, word nl, byte font,
          byte bgr, byte fgr, byte x, byte y, byte w, byte h)
```

The function creates a menu. The first argument is an array of character strings constituting the list of menu entries. The number of strings (lines) in this array should be equal to **nl**. This array will not be replicated, i.e., it should remain untouched for as long as the menu object exists. The next six arguments have the same meaning as for **dm_newtext**. Note that the first character position (column) of a menu is reserved for the pointer to current selection; thus, the actual entries will be displayed starting from position 1. The width specification (**w**) stands for the total character width of the object's bounding rectangle. Also, **h** specifies the rectangle's height in characters.

In contrast to a text object, a menu line is never wrapped at the end of the bounding rectangle, but scrolled out (truncated). Also, it is quite common for a menu to contain more lines than can simultaneously fit into the rectangle (**nl** > **h**). Scrolling operations are available (see below) to make different portions of the menu visible within the rectangle.

Here are the operations that render objects on the screen.

```
word dm_top (dobject_t *o)
```

The function displays the object pointed to by the argument and makes it the new top object. If the object already happens to be the top object, its image is just refreshed. If the object is not present on the display list, it is added as a new object (the list grows). If the object occurs somewhere on the list (but it isn't the top object), it is moved to the top.

The operation invokes this function, which is also available for general use:

```
word dm_dobject (dobject_t *o)
```

Its responsibility is to render a single object pointed to by its argument, which need not be present on the display list.

The function sets **DM_STATUS** to zero (upon success) or to an error code (see above). The setting of **DM_STATUS** is also returned as the function value.

Theoretically, once an object has been successfully created, its subsequent display attempts can only fail with code **DERR_MEM**. Such errors are generally intermittent and they may never occur in sane circumstances. The memory requirements of



**dm_dobject** are quite modest, but the function does invoke **umalloc**.[2] Note, however, that if the contents of EEPROM are changed after an object has been created (e.g., a font is removed, a picture is replaced), then **dm_object** may fail in a confusing way, e.g., signaling **DERR_FNT** or **DERR_FMT**.

A failure to display an object means that not even a fragment of the object has appeared on the screen. Thus, such a failure will not mess up the display.

The value returned by **dm_top** (as well as its setting of **DM_STATUS**) are those of **dm_dobject** invoked to do the actual rendering.

Note that, by using **dm_dobject** directly, you can display an object from outside the display list. If you subsequently want to revert to the top object from the list, you should invoke **dm_dtop** (see below) which will bring the top object back to the front. While any object can be easily brought to the front, there is no automatic and efficient way to remove a displayed object, such that it disappears from the screen and uncovers the objects that it was obscuring. The only way to clear the screen of garbage is to call this function:

```
word dm_refresh ()
```

which scans through all objects from the display list, starting from **DM_DHEAD**, and renders them in turn (by calling **dm_dobject**). If no object from the display list covers the entire screen, then the screen is cleared (to **COLOR_BLACK**) before this action is started. If the display list is empty, the screen is just cleared. Thus, it makes sense to call **dm_refresh** upon initialization.

Note that **dm_refresh** returns a value. The function assumes that it has failed (and stops scanning the display list) upon the first failure of **dm_dobject** invoked in its course. Then, the value returned by **dm_refresh**, as well as the setting of **DM_STATUS**, are those of the failing **dm_dobject**. If all invocations of **dm_dobject** succeed, the function returns zero.

```
dobject_t *dm_delete (dobject_t *o)
```

The function deletes the indicated object from the list and returns its pointer. Normally, the returned value is the same as the argument, except when the argument is NULL, in which case the function deletes from the list the top object (and returns its pointer). Note that this object can also be NULL (in that case nothing happens, and the function returns NULL).

If the specified object does not occur on the list, the function does nothing and returns the argument.

If the top object on the display list is deleted, then the preceding object moves to the top. That new top object is then displayed (refreshed with **dm_dobject**) overwriting any possible objects obscuring it. If the list becomes empty (there is no new top object), the screen is cleared to **COLOR_BLACK**. The error status of **dm_object** implicitly invoked by **dm_delete** to bring the new top object to the front can be verified by explicitly reading **DM_STATUS**.

[2]Perhaps it doesn't have to. A small buffer area is needed for rendering a picture, which could be declared statically in RAM.



Note that the object deleted from the list by **dm_delete** is not deallocated (freed), which is left to the praxis.

```
Boolean dm_displayed (const dobject_t *o)
```

The function returns YES if the specified object is present on the display list and NO otherwise.

```
Boolean dm_dtop ()
```

The function refreshes (re-displays) the top object. If there is no top object (i.e., the display list is empty), the screen is cleared to **COLOR_BLACK**. For example, it may make sense to call it after **dm_object**, if the object displayed by the latter function is not on the display list (and **dm_delete** will not automatically revert to the previous top object).

Four special operations are available for menu objects. Initially, when a menu object is created, its selection pointer points to the first entry, i.e., line number zero. This pointer is available as attribute **SE** of type **word** declared in **dmenu_t** (see **dispman.h**). Its value can be read, but it should not be set directly. It is interpreted as the index of currently selected line between zero and N-1, where N is the total number of entries in the menu.

The following functions are best called in response to events triggered by pressing buttons or moving a joystick. They assume that the menu object specified as the argument is *active*, i.e., it is the frontmost object on the screen.

```
void dm_menu_d (const dmenu_t *m)
void dm_menu_u (const dmenu_t *m)
void dm_menu_l (const dmenu_t *m)
void dm_menu_r (const dmenu_t *m)
```

The first two functions move the selection pointer one position down and up, respectively. Nothing happens if the pointer is already at the boundary (at entry zero for **dm_menu_u**, or at the last entry for **dm_menu_d**). When the cursor crosses the boundary of the menu's rectangle, and scrolled out entries are present in its direction, new entries are automatically scrolled in. The **SE** attribute is updated to reflect the new selection.

The last two functions are used for horizontal scrolling, which is useful if the current set of entries display within the rectangle contains lines longer than the rectangle's width. Thus, **dm_menu_l** shifts the view one position to the left, while **dm_menu_r** scrolls the rectangle one position to the right. The scrolling stops if no new characters can be exposed. The functions do not change the **SE** attribute.