Olsonet Communications

PicOS
LCD-G interface

Version 0.81
October 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 56-byte “chunks” that need not occur in any particular order.[1] Each chunk contains its number, which implicitly specifies a pair of coordinates (describing the location of the starting pixel) and 54 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, at least once.

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 current 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 of the bounding rectangle. 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, e.g., if **nl** happens to be larger than the rectangle's capacity. Similar to **lcdg_wl**, it fails silently and does nothing if no font is defined.

Display manager

The functions discussed in this section (described in files **lcdg_dispman.c** and **lcdg_dispman.h** in **PicOS/Libs/LibLCDG**) 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 "lcdg_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 **lcdg_dispman.h**).

lcdg_dm_img_t	a picture
lcdg_dm_tex_t	a piece of text
lcdg_dm_men_t	a menu

These types are structures sharing a common initial chunk of data represented by type **lcdg_dm_obj_t**. This means that it makes sense to cast any of the three specific object types to type **lcdg_dm_obj_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;
lcdg_dm_obj_t   *next;
address         Extras;
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 **LCDG_DMTYPE_IMAGE**, **LCDG_DMTYPE_MENU**, **LCDG_DMTYPE_TEXT** identifying the actual object type (and determining the layout of the remaining portion of the object structure). The **Extras** attribute **is reserved for the praxis** and can be used to store a pointer to some auxiliary data related to 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 **LCDG_DM_HEAD** (a global variable of type **lcdg_dm_obj_t***). This list can be empty, in which case **LCDG_DM_HEAD** contains NULL. Another global variable of the same type, **LCDG_DM_TOP**, points to the last object on the display list, which is called the *top* (or *current*) object. **LCDG_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, **lcdg_dm_newtop** (see below) is called after an object creation to show the object on the screen and make it the current (top) object.

```
lcdg_dm_obj_t *lcdg_dm_newimage (   word ihandle,
                                    byte x,
                                    byte y);
```

The function creates an image (picture) object. The first argument is the **image handle**, which can be returned, e.g., by a call to **lcdg_im_find** (as described further in this document). The last two arguments specify the left top corner of the picture's bounding rectangle. The coordinates of the right bottom corner of the bounding rectangle will be determined from the picture's dimensions.

Note: all the functions creating displayable objects assume that the positions of those objects are specified at the moment of creation. This may not be such a terrific idea. Perhaps we need a special function to set the position of the object afterwards, with 0, 0 being the default? Then, what to do with other attributes, like fonts and colors, width, height?

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 **LCDG_DM_STATUS** of type **byte**. Generally, this variable can take the following values (not all of them are applicable to pictures):

0	no problem (last operation has succeeded)
LCDG_DMERR_NOMEM	memory allocation (**umalloc**) failure
LCDG_DMERR_GARBAGE	format error (bad image handle)
LCDG_DMERR_NOFONT	font unavailable
LCDG_DMERR_RECT	bounding rectangle out of screen

For example, 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 **LCDG_DM_STATUS** will be set to **LCDG_DMERR_RECT**.

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

```
lcdg_dm_obj_t *lcdg_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 **LCDG_DMERR_NOMEM**, **LCDG_DMERR_NOFONT** (the specified font is not available), and **LCDG_DMERR_RECT** (the resultant bounding rectangle does not entirely fit on the screen).

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

```
lcdg_dm_obj_t *lcdg_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 NULL-byte-terminated 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 **lcdg_m_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.

```
byte lcdg_dm_newtop (lcdg_dm_obj_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 lcdg_dm_display (lcdg_dm_obj_t *o);
```

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

The function sets **LCDG_DM_STATUS** to zero (upon success) or to an error code (see above). The setting of **LCDG_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 **LCDG_DMERR_NOMEM**. Such errors are generally intermittent and they may never occur under sane circumstances. The memory requirements of **lcdg_dm_display** 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 **lcdg_dm_display** may fail in a confusing way, e.g., signaling **LCDG_DMERR_NOFONT** or **LCDG_DMERR_GARBAGE**.

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 **lcdg_dm_newtop** (as well as its setting of **LCDG_DM_STATUS**) are those of **lcdg_dm_display** invoked to do the actual rendering.

Note that, by using **lcdg_dm_display** 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 **lcdg_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:

```
byte lcdg_dm_refresh ();
```

which scans through all objects from the display list, starting from **LCDG_DM_DHEAD**, and renders them in turn (by calling **lcdg_dm_display**). 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 **lcdg_dm_refresh** upon initialization.

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

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

```
lcdg_dm_obj_t *lcdg_dm_remove (lcdg_dm_obj_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 from the display list is deleted, then the preceding object moves to the top. That new top object is then displayed (refreshed with **lcdg_dm_display**) 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 **lcdg_dm_display** implicitly invoked by **lcdg_dm_remove** to bring the new top object to the front can be verified by reading **LCDG_DM_STATUS**.

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

```
Boolean lcdg_dm_shown (const lcdg_dm_obj_t *o);
```

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

```
byte lcdg_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 **lcdg_dm_display**, if the object displayed by the latter function is not on the display list (and **lcdg_dm_remove** will not automatically revert to the previous top object).

The function returns the status of **lcdg_dm_display** invoked to render the 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 **lcdg_dm_men_t** (see **lcdg_dispman.h**). Its value can be read, but it should not be set directly. For purists, this function (macro) can be used to reference the SE attribute in a less explicit way:

```
#define  lcdg_dm_menu_c(m)        (((lcdg_dm_men_t*)(m))->SE)
```

The value of **SE** 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 provided to be 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 lcdg_dm_menu_d (const lcdg_dm_men_t *m)
void lcdg_dm_menu_u (const lcdg_dm_men_t *m)
void lcdg_dm_menu_l (const lcdg_dm_men_t *m)
void lcdg_dm_menu_r (const lcdg_dm_men_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 **lcdg_dm_menu_u**, or at the last entry for **lcdg_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 displayed within the bounding rectangle contains lines longer than the rectangle's width. Thus, **lcdg_dm_menu_l** shifts the view one position to the left, while **lcdg_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 affect the **SE** attribute.

Here are two more functions that appear generally useful in the context of menus:

```
char **lcdg_dm_asa (word n);
```

This function allocates an array to store **n** strings and returns its pointer or NULL, if **umalloc** fails. In the latter case, **LCDG_DM_STATUS** is set to **LCDG_DMERR_NULL**. If the array has been successfully allocated, it is filled with NULL pointers. Such an array can be subsequently filled with string pointers and used for a dynamically created list of menu items, i.e., the first argument of **lcdg_dm_newmenu**.

```
void lcdg_dm_csa (char **lines, word n);
```

This one deallocates the specified array of strings of size **n**, assuming that all the strings as well as the array itself were previously allocated with **umalloc**. Any entries in **lines** that are NULL are not deallocated. For many menus, the array of lines is created dynamically by the praxis, so it must be cleaned up when the menu is destroyed.

The image handling module

The functions pertaining to image processing are contained in files **lcdg_images.c** and **lcdg_images.h** in directory **PicOS/Libs/LibLCDG**. They cover image lookup in EEPROM, rendering, as well as OEP hooks for image transmission over unreliable links.

Images are stored in EEPROM in such a way that one picture occupies an entire number of 8KB pages. Note that those pages are logical, i.e., the 8KB size is not necessarily related to any physical properties of the EEPROM, although, of course, the fact that that size is a nice power of 2 is not without a significance. The pages constitute a (somewhat crude) unit of storage allocation for images, and the 8KB size of that unit reflects a tradeoff between fragmentation and the ease of lookup. The image handling module must be initialized by a call to this function:

```
void lcdg_im_init (word fp, word np);
```

where **fp** is the number of the first EEPROM page (8KB chunk) to be used for image storage, and **np** is the total number of pages used for this purpose. If **np** is zero or extends beyond the EEPROM size, the image area will comprise the entire trailing portion of EEPROM starting at address **fp** x 8KB. On the hand, if **fp** falls beyond the EEPROM size, the function will trigger a system error.

Normally, if the EEPROM already contains some images, the praxis should know the bounds of the image area and specify them properly with an initial call to **lcdg_im_init**. If the EEPROM has been physically erased, the area will be automatically recognized as containing zero images, i.e., no specific initialization is required in such a case.

```
word lcdg_im_find (const byte *lbl, byte len, word pn);
```

This function can be used to locate an image by its label. The first two arguments describe the label, which can be any string of bytes (not necessarily ASCII characters) of the specified length (**len**) between 0 and 38. If **len** is less than 38 (which is the total size of the fixed label area of an image), then the first **len** bytes in **lbl** must be identical to the first **len** bytes in the image's label for a match. This also means that if **len** is zero (in which case **lbl** is irrelevant and can be NULL), any image matches the search. The last argument (**pn**) can be used to start the search from a particular place (e.g., where previously stopped), which makes it possible to scan through all the images currently stored in EEPROM.

The function returns either WNONE (**0xFFFF**) or the so-called *image handle*, which is simply the pointer (number) to the image's first page. If the last argument is WNONE, the search will begin from the very first page of the image area in EEPROM. Otherwise, the search will start at page **pn**+1. This way, by setting pn to the previously located handle (other than WNONE), you can start effectively continue the previous search. In particular, here is a loop that scans through all the images:

```
pn = WNONE;
while ((pn = lcdg_im_find (NULL, 0, pn)) != WNONE) {
   ... process the image pointed to by handle pn ...
}
```

The simplicity of the search scheme consists in the fact that it is easy to identify the first page of an image by looking at its first two bytes (the magic code). That first page, among other things, contains pointers to the remaining pages (if any) representing the complete image.

```
byte lcdg_im_hdr (word pn, lcdg_im_hdr_t *hdr);
```

Given a valid image handle (**pn**), the function returns (via **hdr**) the image header, which is a structure with the following layout (declared in **lcdg_images.h**):

```
typedef struct {
   byte X, Y;
   byte Label [38];
}  lcdg_im_hdr_t;
```

where **X** and **Y** are the image dimensions in pixels, and **Label** is the complete label.

The value returned by the function indicates success or failure in a manner similar to the functions described earlier. Thus, zero means success, i.e., the handle was valid, and the structure pointed to by **hdr** has been filled with the image's header. Otherwise, in this case, the value can only be **LCDG_IMGERR_HANDLE**, which means that the handle doesn't seem to point to the first page of a valid image.

```
byte lcdg_im_disp (word pn, byte x, byte y);
```


Given a valid image handle (**pn**), the function renders the image on the display, with **x** and **y** standing for the left top corner coordinates. In particular, the **lcdg_dm_display** function of the display manager (see the previous section) invokes **lcdg_im_disp** in the course of its action.

The function returns zero on success. The possible error codes are:

LCDG_IMGERR_HANDLE	the specified handle doesn't point to a valid image
LCDG_IMGERR_NOMEM	memory allocation failed
LCDG_IMGERR_GRBAGE	the image structure in EEPROM appears damaged

The amount of dynamically allocated memory needed by the function to do its job is about 70 bytes.

Some of the error codes returned by the image-related functions coincide with those returned by functions in related modules. For example **LCDG_IMGERR_NOMEM** and **LCDG_DMERR_NOMEM** translate into the same value (17), which is the same as **OEP_STATUS_NOMEM**. Also, **LCDG_IMGERR_GARBAGE** and **LCDG_DMERR_GARBAGE** are identical (33).

Note that the image rendered by l**cdg_im_disp** will only show up on the screen if the display is turned on (see **lcdg_on** above).

```
word lcdg_im_free ();
```

This function returns the number of free pages available in the declared image area of EEPROM.

```
byte lcdg_im_purge (word pn);
```

This function removes the image pointed to by the specified handle reclaiming its pages (marking them as free). The values returned are the same as for **lcdg_dm_disp**.

```
byte lcdg_im_purge (word pn);
```

This function removes the image pointed to by the specified handle reclaiming its pages (marking them as free). The values returned are the same as for **lcdg_dm_disp**.

OEP functions for exchanging images

The following functions implement image transfer over OEP. Please consult the OEP document before reading this part.

As explained in the OEP document, the praxis should first negotiate the exchange of an image in its private way. On the sender side, the image (which must be stored in local EEPROM) is represented by its handle, e.g., obtained by invoking **lcdg_im_find** (see above). The sender will also learn from the receiver the requisite values of LID (link ID) and RQN (the request number). The sender must in turn communicate to the receiver the parameters of the image, which come as the contents of the header structure (**lcdg_im_hdr_t**). The receiver will calculate the number of chunks comprising the image from the **X**, **Y** dimensions stored in the header.

Thus, in order to receive an image, the receiver calls this function:

```
byte oep_im_rcv (const lcdg_im_hdr_t *hdr, byte xd, byte yd);
```


where **hdr** points to the image header, and the remaining two parameters are optional. They indicate the coordinates of the left upper corner on the screen for rendering the image simultaneously with its reception (and storage in EEPROM). If the image is just to be received and stored (without being displayed at the same time), **xd** should be BNONE (0xFF). The value of **yd** is then irrelevant.

For its end, the sender should invoke:

```
byte oep_im_snd (word ylid, byte yrqn, word pn);
```

where **ylid** and **yrqn** are link ID and request number obtained from the receiver, and **pn** is the image handle.

Each of the two functions will build some data structures needed by the chunk handlers during the exchange and invoke **oep_rcv**/**oep_snd** to do the job. The value returned by **oep_im_rcv** or **oep_im_snd** should be interpreted in the same way as the status value returned by **oep_rcv** or **oep_snd**. In particular, zero means that the operation has been successfully started and the praxis should now await is completion with **oep_wait**. Any nonzero value means failure. In addition to the OEP failure codes, the above functions can also return these values: **LCDG_IMGERR_HANDLE** (the handle for **oep_im_snd** is not valid), **LCDG_IMGERR_NOMEM** (failed memory allocation, both functions), **LCDG_IMGERR_GARBAGE** (corrupted image file for **oep_im_snd**, or illegal **X**, **Y** values in the header for **oep_im_rcv**), **LCDG_IMGERR_NOSPACE** (insufficient number of free pages to accommodate the image for **oep_im_rcv**).

Once the given end of the transfer has been successfully started (the corresponding function has returned zero), the praxis should use **oep_wait** (see above) to await its (successful or failed) completion. When that happen, the praxis should call this function:

```
void oep_im_cleanup ();
```

which will clean up after the exchange, i.e., deallocate the dynamic structures used by the respective chunk handler. Note that the cleanup function should only be called after a successful initialization of the exchange. There is nothing to clean up if the initialization has failed (i.e., **oep_im_snd** or **oep_im_rcv** has returned a nonzero status).

Appendix: Image formats

Images acquired are stored in EEPROM in such a way that one image occupies an entire number of 8KB (logical) pages. The image module identifies a contiguous range of 8KB pages to be used for storing images. A page falling within that range can be of one of three kinds:

- Free, if its first two bytes look like 0xFFFF. In particular, immediately after the EEPROM is erased, all pages are free.
- The first page of an image. Such a page has 0x7F00 (little endian) in the first two bytes. When looking for a specific image, or counting images stored in the EEPROM, the praxis only looks at such pages.
- Other. Normally, such a page belongs to an image as one of its subsequent (non-first) pages. It is theoretically conceivable that the image area stores objects other than images. In such a case, the pages of those object must not

start with 0xFFFF or 0x7F00. Then, they will never be used for storing images or interpreted as parts of images.

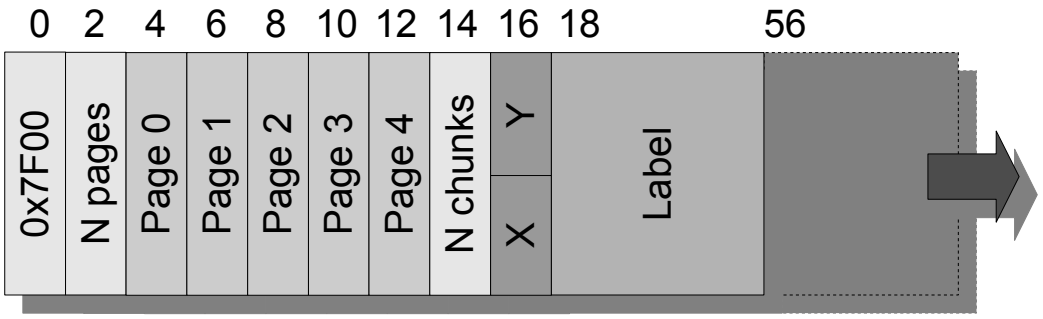

Figure 1: The first page of an image stored in EEPROM.

Any page belonging to an image consists of an entire number (up to 146) of 56-byte chunks. As 8192 is not divisible by 56, the last 16 bytes of every page are unused. Their contents are irrelevant, but it makes sense to keep them all ones (0xFF).

All chunks except for the first chunk of the first page (see) contain descriptions of image pixels. One pixel is described with 12 bits packed in such a way that three consecutive bytes represent two consecutive pixels of the image. The first word of a chunk stores the chunk number between 0 and the total number of chunks in the image – 1. The chunk number is required, as the chunks need not be stored in order. As their reception may not be reliable, the received chunks are simply written to the flash in whatever order they arrive (some of them being missing and retransmitted later). This way the access to EEPROM is simpler and faster.

The first chunk of the first page of an image has a special purpose. Its fields are interpreted as follows:

Bytes 0-1	0x7F00 (this is the magic code identifying the first page of an image).
Bytes 2-3	The total number of pages in the image, including the first page.
Bytes 4-13	This is a 5-word array containing the numbers of all pages occupied by the image, starting with this page number. Only the number of entries indicated in bytes 2-3 are valid. A page number is the address of the first byte of the page shifted by 13 bits to he right.
Bytes 14-15	The total number of chunks in the image excluding the first (header chunk), i.e., the total number of pixel chunks.
Byte 16	The X dimension, i.e., the row size (or the image width) in pixels. The maximum legitimate width is 130.
Byte 17	The Y dimension, i.e., the column size (or the image height) in pixels. The maximum legitimate height is 130.
Bytes 18-55	The image label, i.e., an arbitrary sequence of 38 bytes.

The maximum number of pages that a single image may occupy is 5. Note that a continuation (non-first) page will begin with a chunk number, which can never be larger than 4 x 146 = 584. Thus, a continuation page cannot accidentally pose for an empty page or the first page of an image.

In addition to the EEPROM format for images, there is another format in which images can be stored in regular files before being uploaded into EEPROM (or converted to

EEPROM format). The recommended procedure for creating such downloadable images is al follows:

1. Convert the original image (e.g., in JPEG format) to a 130 x 130 pixels standard Windows bitmap format with 24 bits per pixels.

2. Run the bitmap file through a conversion program (script) named cbm.tcl in the following way:

   `cbm.tcl -l label sourcefile.bmp targetfile.lcd`

   where label is the string describing the contents of the label field of the image's first chunks, sourcefile.bmp is the bitmap version of the image and targetfile.lcd represents the target file where the converted copy of the image will be written.

If the **`-l`** argument is absent, then the label field will contain all zeros. Otherwise, **`label`** is interpreted as a character string whose initial 38 characters will be put into the label field. If the string is shorter than 38 characters, then the remaining bytes of the label will be filled with zeros.

An alternative variant of the label argument is **`-ln`** ***`list_of_numbers`***, where list_of_numbers may consist of comma separated integer values, e.g.,

`cbm.tcl – ln "1,2,-10,0xfffccc,999" myphoto.bmp myphoto.lcd`

Each number is interpreted as a 32-bit little endian value to be stored into four consecutive bytes of the label area. As before, any bytes in excess of 38 are ignored, while any leftover trailer of the label area is filled with zeros.

The image format created by **`cbm.tcl`** is even simpler than the EEPROM format (the file is continuous, so there is no need to deal with pages). The file starts with the 16-bit little endian word 0x77AC (used as a magic code), followed by two bytes storing the X and Y dimensions of the image, followed in turn by 38 bytes of the label. This 42-byte header is followed by the pixel codes. The total number of bytes expected in that area is at least X x Y x 12 / 8. They can be cut into 54-byte chunks numbered from zero up. The last chunk need not be full (i.e., the pixel area need not be a multiple of 54 bytes), if the total number of pixels in the image does not warrant a full chunk at the end.