PicOS

Seawolf interface

Version 0.11
October 2008

Preamble

This note describes the praxis interface developed for the Seawolf project. It supplements these documents: **`LCDGint.pdf`** (describing the Nokia LCD display interface, the menu system, and the image handling system) and **`OEP.pdf`** (describing the Object Exchange Protocol module). This document comments on the interface specific to the Seawolf praxis, while the modules described in the other documents can be used in other applications in principle unrelated to Seawolf.

Needless to say, everything written below is preliminary. While it refers to existing (already implemented) software, this software is far from being cast in stone.

Building EEPROM images

The present library of functions has been implemented with the intention of facilitating a certain class of applications dealing with people profiles, including pictures, stored in EEPROM (flash memory) at the node. The node is equipped with a Nokia 6100p LCD display, a joystick (4 positions labeled N, E, S, W + push), and two push buttons (labeled B0 and B1).

To simplify the functions (most importantly to reduce the code size), EEPROM has been partitioned into certain zones whose boundaries are static (but can be easily redefined at compile time). Note that the largest objects stored in EEPROM are images – their memory demands by far outweigh the demands of all other objects combined. Consequently, little can be gained by trying to come up with a smart allocation scheme for those other objects. Instead, they can be stored in a fixed (and relatively small) dedicated chunk of EEPROM partitioned in a moderately efficient manner, while the remaining (large) portion of EEPROM is dedicated to images (see **`LCDGint.pdf`**). This way, even if the non-image objects are not handled extremely efficiently (e.g., incur some fragmentation), the impact of that inefficiency on overall EEPROM usage is negligible.

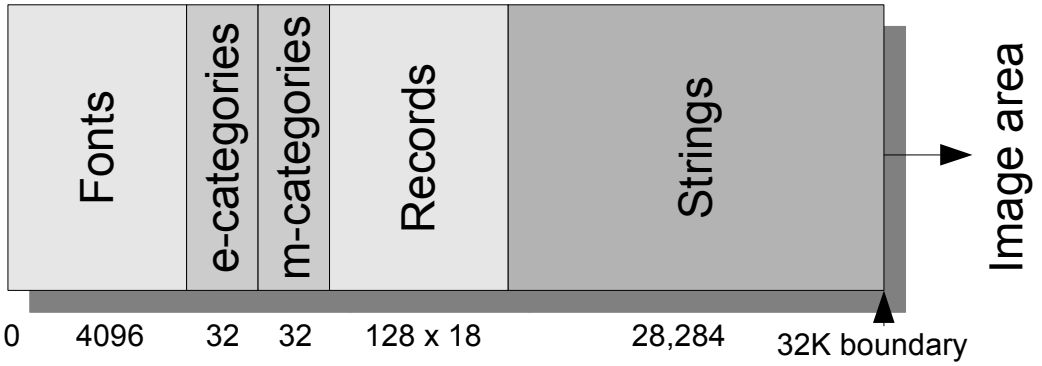

Figure 1: EEPROM layout

The organization of EEPROM is shown in Figure 1. After a preliminary discussion with Wlodek, we have decided to modify this as follows: 1) add a separate area for “Ignore” IDs (they need not look like records, and it makes sense to keep them in a separate table); 2) add a dynamic data area where the praxis can store non-volatile stuff. I will revise this figure when these changes have been implemented. The first 4KB chunk is used by the fonts file needed for rendering texts on the Nokia LCD. At location 4096, we have two word arrays, 16 items each, representing the category lists: event categories and *my* categories. Not all slots have to be filled: the array size (16) determines the limit on the number of categories, with the actual number possibly being smaller. The

category lists are followed by the Records area, which again is an array of fixed-size entries (18 bytes each) representing the profiles of up to 128 people. This part, in turn, is followed by the Strings area, which is a storage for text strings pointed to from records or category lists. That area can expand up to the 32K boundary, which begins the Image zone, i.e., the first page of the image zone has number 4 (image pages are 8KB).

The boundaries of EEPROM zones are described in file **`sealists.h`** (directory **`TEST`**) as symbolic constants. Note that if these constants should change, you should also update the corresponding definitions in **`mkeeprom.tcl`** (see below).

EEPROM layouts are described by XML files. Here is an example file which illustrates all the XML elements that can appear in an EEPROM definition:

```
<sealist>
<eventcategories>
        <category>Session on chicken feeding</category>
        <category>Session on beer brewing</category>
        <category>Keynote talk: how to decorate a pub</category>
        <category>Reception: wine tasting</category>
        <category>Vodka bar</category>
        <category>Greek restaurant</category>
        <category>Irish pub</category>
</eventcategories>
<mycategories>
        <category>Likes to drink</category>
        <category>Pal</category>
        <category>Smokes pot</category>
        <category>Smart guy</category>
        <category>Money: buys drinks</category>
        <category>Teetotaler</category>
        <category>Robs people</category>
        <category>Blows up buildings</category>
        <category>Smells badly</category>
</mycategories>
<people>
        <record class="yes">
                <id>0xBACA0001</id>
                <name>Wlodek Olesinski</name>
                <nickname>CEO</nickname>
                <why>0,1,3;4,5,6</why>
                <note>Prefers straight vodka with a touch of grapefruit juice</note>
        </record>
        <record class=”no”>
                <id>0xBACA0002</id>
                <name>Osama Bin Laden</name>
                <nickname>Osama</nickname>
                <why>7,8</why>
                <note>If I meet him there, something must be wrong</note>
        </record>
        <record class=”ignore”>
                <id>0xBACA0003</id>
        </record>
</people>
</sealist>
```


The numbering of categories is determined by the ordering of the items; thus, for example, *Session on chicken feeding* has number 0 (within event categories) and *Teetotaler* is number 5 (within my categories). There is an optional "ord" argument for the <category> item to force a specific number, but it is redundant, confusing, and should be dropped (this is a note to myself). The maximum number of categories in each group is 16. The way they are stored in EEPROM is that each category string is written to the Strings zone, and the corresponding word in the categories array is set to the address of that string, assuming that the first byte in the Strings zone has address 0. Unused entries are filled with **`0xFFFF`** (WNONE). Note that zero is a legitimate string pointer. There can be no holes in a category array. This means that the first WNONE entry marks the end of the respective category list (unless the list uses up all its 16 slots).

A string is stored (within the Strings zone) as the length byte followed by the specified number of bytes (no NULL-byte sentinels). This is more convenient from the viewpoint of the praxis, which, when moving the string to memory, has to known its length to allocate an appropriately sized array.

The program creating EEPROM files identifies multiple occurrences of the same string, and stores only a single copy in the Strings zone (creating multiple pointers as needed). If there is a need to add records dynamically (on-line), the end of the used part of the string zone can be easily detected by the node at startup (e.g., as the first WNONE word).

The Records area is also a straightforward array of fixed-length records with this layout:

network ID
event categories
my categories
class
nickname
name
note
image

Except for the network ID, which is a **`lword`** (4 bytes), each attribute takes exactly 2 bytes (a **`word`**); thus, the record size amounts to 18 bytes. The two categories words are bit maps reflecting the category membership. The leftmost bit corresponds to category number zero (the first one on the respective list), and 1 stands for "present".

Record number zero is "self". It describes the owner of the node and points to his/her picture. There may be some stuff pointed to from record zero which is not applicable to other records. We may want to use two consecutive records for that, i.e., records 0 and 1.

The value of the class word reflects the class attribute of the XML <record> element (see above). Only three values are used at present: 1 = "yes" (meaning a positive association), 2 = "no" (negative association), 3 = "ignore". I didn't understand the role of "ignore", but Wlodek illuminated me on this one. There will be no "ignore" records. IDs to be ignored will be kept in a separate table. Thus, in the above example of a data file, we may need a separate element to describe "ignore" Ids. More values can be added later, if needed.

With the present way of looking up entries in the Records array, an empty record slot is recognized by its class equal WNONE. The next three fields: nickname, name, and note, are word-size pointers into the the Strings zone (as described above). The image word

contains the image handle (see **LCDGint.pdf**) pointing to the first page of an image in EEPROM, or WNONE, if no image is associated with the record.

It is assumed that the name field for a nonempty record is set (the pointer is not WNONE)~~, except when class is "ignore", in which case only the network ID must be set (the remaining fields are optional).~~ The program creating EEPROM files makes sure that, as long as name is set, nickname is set as well: nickname defaults to name (the same string pointer is used), if not set explicitly.

Here is how you can generate EEPROM files:

```
mkeeprom.tcl fontfile xmldata output
```

where ***fontfile*** is a file containing fonts for the LCDG driver, ***xmldata*** is an XML file with the layout described above, and ***output*** is the target file where the resulting EEPROM image is to be written. The way to prepare a font file is to go to directory **TEST/FONTS** and do this:

```
gcc fontfile.c
a.exe > ../fonts.bin
```

The present set of fonts (which will be written to **fonts.bin** with the above sequence) consists of three "standard" fonts (numbered 0-2) + one special font (number 3) used for rendering the "meters" representing graphically the category membership for a record.

The script will build the EEPROM image by placing the contents of the fonts file at location zero, and then setting up the categories arrays, Records and Strings zones, based on the XML description. When processing a record, the scripts will check whether the current directory includes a file named ***image_xxxxxxxx.nok***, where **xxxxxxxx** is the lower-case hex digit content of the network ID attribute. For example, if the ID (as specified in the record's <id> element) is **0xBACA0001**, the file name will be **image_baca0001.nok**. If such a file is found, it is interpreted as the image file (picture) to be associated with the record. Otherwise, the record will have no associated picture (its image handle will be WNONE). The way to convert pictures to the format acceptable by **mkeeprom.tcl** is decribed in **LCDGint.pdf**.

It is possible to indicate to the script that the picture files should be sought in a specific directory (not necessarily the current one), e.g.,

```
mkeeprom.tcl -i MY/PICS FONTS/fonts.bin data.xml eeprom.bin
```

The **-i** parameter must be followed by a directory path and, if present, must precede all other arguments.

The output file generated by **mkeeprom.tcl** stores the absolute content of EEPROM starting at location 0. Its length depends on the amount of EEPROM filled by the data, most notably, on the number of images. There is an alternative (more compact) way of representing the EEPROM contents as a list of chunks. This feature has been programmed into the script, but is disabled at present (the test praxis that loads EEPROM images into nodes doesn't recognize it).

Extra functions provided in **sealists.c**

By including **sealists.h** in the praxis's header, you get access to a few functions that will allow you to extract data from EEPROM set up according to the layout described in

the previous section. These functions are described below. To make sense of them, you have to read **LCDGint.pdf** first.

```
lcdg_dm_obj_t *seal_mkcmenu (Boolean ev);
```

The function builds a menu object from the categories array, which is either event categories (if **ev** is YES) or my categories (if **ev** is NO). The menu lines are category names.

Note that the menu object (whose pointer is returned by the function), as well as its **Lines** attribute, are allocated by **umalloc**. Consequently, the proper way to deallocate this object (when it is no longer needed) should include deallocating the **Lines** array, e.g., according to this scheme (see **LCDGint.pdf**):

```
dobj = seal_mkcmenu (YES);
...
...
lcdg_dm_csa (dobj->Lines, dobj->NL);
ufree (dobj);
```

The function invokes **lcdg_dm_newmenu** to create the menu object. The location of the menu on the screen, its geometry, colors, and the font, are fixed (meaning that the menu will always show up in the same position on the screen. The values of all those parameters are described by symbolic constants in **sealists.h**, which can be changed to taste.

The function may fail, in which case it will return NULL. The reason for failure can be determined by examining **LCDG_DM_STATUS**, which, in this case, can receive one of these values:

LCDG_DMERR_GARBAGE	the corresponding categories list is empty
LCDG_DMERR_NOMEM	**umalloc** failure (not enough memory)

If the symbolic constants describing the parameters of the categories menu (in **sealists.h**) have malicious values (or/and the font file in EEPROM is broken), additional error codes are possible, i.e., those corresponding to the possible failures of **lcdg_dm_newmenu**.

```
lcdg_dm_obj_t *seal_mkrmenu (word class);
```

The function builds a menu object from all nonempty records (in the Records zone) whose class matches the specified value. The lines of this menu consist of the nicknames of the selected records. Wlodek says this function is useless in its present form and suggests something like this:

```
lcdg_dm_obj_t *seal_mkrmenu (Boolean (*qual)(sea_rec_t*));
```

with **qual** acting as the record qualifier predicate. The rules are similar to those for **seal_mkcmenu**. Note that in this case the numbering of menu entries need not coincide with the numbering of records in the Records zone. Thus, the menu object uses its **Extras** attribute to store record pointers. **Extras[i]** can be viewed as a handle to the record represented by the **i**'th menu entry. When deallocating the complete object, you have to remember to free the **Extras** array (in addition to freeing **Lines**).

```
sea_rec_t *seal_getrec (word handle);
```

The function allows you to get hold of the complete record pointed to by the handle, e.g., taken from the **Extras** array of a records menu (see above). The function allocates (and returns a pointer to) the following structure (declared in **sealists.h**):

```
typedef struct {
      lword NID;
      word  ECats, MCats, CL, IM;
      char  *NI, *NM, *NT;
} sea_rec_t;
```

The attributes are: **NID** – the network ID, **ECats**, **MCats** – the category membership flags (events, my), **CL** – class, **IM** – image handle, **NI**, **NM**, **NT**, strings representing the nickname, the name, and the note.

~~Records with no nickname (this is only possible for some ignore-class records – see above) are never included in record menus, even if the argument of **seal_mkrmenu** is 0 (for ignore). Also, **seal_getrec** will ignore such a record, returning NULL (instead of creating a **sea_rec_t** structure with **NI** equal NULL).~~ This paragraph is no longer relevant.

~~Technically, **seal_getrec** admits a record with absent name but present nickname, although **mkeeprom.tcl** will never create such records.~~ An absent string (like **NT**, which can be legitimately absent) is represented by a NULL pointer. When moved to RAM, the strings become classical C strings terminated with NULL bytes.

The function will fail, returning NULL and setting **LCDG_DM_STATUS** to nonzero in the following circumstances:

- The nickname pointer of the record represented by the handle is WNONE; **LCDG_DM_STATUS** returns **LCDG_DMERR_GARBAGE**.
- There is not enough memory to allocate **sea_rec_t**; **LCDG_DM_STATUS** returns **LCDG_DMERR_NOMEM**.

You should remember to deallocate the structure created by **seal_getrec** when it is no longer needed. This also involves deallocating any non-NULL strings. Here is the function that you can use for this purpose:

```
void seal_freerec (sea_rec_t *rec);
```

It will deallocate all the strings as well as the record structure itself.

```
word seal_findrec (lword ID);
```

This function searches the Records zone for a record whose network ID matches the specified value. If such a record is found, the function returns its handle; otherwise, the function returns WNONE. Note that **seal_findrec** does not check whether the nickname field is present in the record. Consequently, **seal_getrec** may legitimately fail on a handle returned by **seal_findrec**. Irrelevant: we will make sure that all records do have nicknames.

```
byte seal_meter (word f0, word f1);
```


This function displays a “meter” at the bottom of screen based on the contents of two arguments, which are viewed as sets of flags, e.g., representing categories memberships. For illustration, here is sample code that can be executed in response to a button being pressed within a records menu:

```
word rh;
sea_rec_t *rec;
…
// Acquire a handle to the currently selected record
rh = LCDG_DM_TOP-> Extras [lcdg_dm_menu_c (LCDG_DM_TOP)]);
if ((rec = seal_getrec (rh)) != NULL) {
      // We have the record
      if (rec->IM != WNONE)
            // Display the picture
            lcdg_im_disp (rec->IM, 0, 0);
      // Display the meter
      seal_meter (rec->ECats, rec->MCats);
      // Free the record
      seal_freerec (rec);
}
```

The above code displays the image associated with the current record (if the image handle is not WNONE) and then renders the meter. The latter takes the entire width of the screen. Its Y coordinate, as well as the colors, can be changed by redefining some constants in **sealists.h**.

The TEST praxis

The praxis in **TEST/app.c**, in collaboration with the **oss.tcl** script, illustrates the usage of most of the library functions provided by these modules: **lcdg_images.[ch]**, **lcdg_dispman.[ch]** (see **LCDGInt.pdf**), **sealists.[ch]**, as well as **oep.[ch]** (see **OEP.pdf**) and **ab.[ch]** (see **Serial.pdf**). It also provides a way for ~~uploading~~ downloading files (e.g., generated by **mkeeprom.tcl** described above) to EEPROM. Here is its functional description.

We have discussed with Wlodek the question of what should be called “uploading” and what “downloading. My (system) perspective was confusing, as I would call “uploading” an act of loading some stuff from the PC to the node, even though, within the bigger picture, that stuff would correspond to something that possibly originated at the (Noombuzz) server to be downloaded from there into the node. Thus, to avoid confusion, anything that gets stored in the node is **downloaded** there. By the same token, anything copied from the node to the PC (or to the server) is **uploaded**.

The praxis starts by initializing the UART PHY for the so-called External Reliable Scheme (XRS) described in **Serial.pdf**. Then, it initializes the image handler (**lcdg_im_init**) and the OEP module (**oep_init**). The UART will be switched between XRS (providing for standard command-line communication with the OSS program) and OEP (during transmission of data chunks, e.g. EEPROM content).

Having started the praxis on a node connected to the PC via a UART, you should execute on the PC the **oss.tcl** script present in the **TEST** directory specifying the UART device as the only argument. It is enough to provide the **COM** number (or **ttyUSB** number on Linux systems), although the full device name (e.g., **COM8:** or **/dev/ttyUSB0**) will work as well. The remaining parameters required by XRS are

hardwired into the script and match those of the praxis (bit rate = 115200 bps and maximum packet length = 60 bytes). The latter equals the maximum packet length (excluding the checksum) of CC1100, which is the same as the maximum packet length used by OEP.

Note: at the end of the **oss.tcl** script you will see this line:

```
u_settrace 7 dump.txt
```

In the above form, it will dump all packets exchanged by the script and the praxis to file **dump.txt** in the current (**TEST**) directory. You can remove this line (or change the argument to 0) to switch this off. If you want to see those packets as things proceed, open a separate window and execute

```
tail -f dump.txt
```

in it.

Normally, any line you type as input to the script will be transformed into an XRS packet and send to the praxis a command, which you would traditionally type into a terminal emulator. A line starting with **!** (the exclamation mark) represents something that should be processed by the script itself. This may be a command initiating an OEP exchange between the script and the praxis, e.g., a transfer of EEPROM chunk or a picture. **!!** provides a shortcut for the last issued command that was rejected. A rejection may happen when the previous command hasn't been confirmed yet by the praxis (no XRS ACK has been received for it yet). You will see the message “board busy” in such circumstances, and the command will have to be issued again, e.g., by entering **!!**. While the script could easily buffer commands and issue them automatically as previous commands are confirmed, that could lead to a confusion, especially as some of those commands may result in XRS being temporarily switched off and OEP taking over. Consequently, at most one command can be outstanding at a time.

Here is the list of direct commands (interpreted directly by the praxis). Remember that each of them must fit into one line, with the line length limited (truncated) to 56 characters.

```
off
on [u]
or
```

These commands affect the LCD: turn off, turn on, and refresh. For **on**, the optional argument specifies the contrast. By refresh, we mean redisplaying the current hierarchy of objects by calling **lcdg_dm_refresh**.

```
ci n h [x [y]]
cm n nl fo bg fg x y w h
ct n fo bg fg x y w
```

These commands directly create displayable objects: image, menu, text, from manually entered data. In all cases, the first argument is a number (index) under which the object can be accessed by subsequent commands. The index value must be between 0 and 31, inclusively. If an index is already taken by some object, the old object is removed (its memory freed).

For **ci**, the remaining arguments specify the image handle (the number of the image's starting page, which can be listed with **li** – see below) and the two coordinates of the image's left top corner on the screen. They are both optional and default to 0, 0.

For **cm**, the parameters describe: the number of lines in the menu, the font number (between 0 and 2), the background color number (0-10), the foreground color, the coordinates of the top left corner, the width in characters, and the height in lines. You will be asked to enter the specified number of lines, which can be any strings up to 56 characters each.

For **ct**, the parameters are: the font number, the background color, the foreground color, the coordinates of the top left corner, and the width in characters. See **LCDGint.pdf** for details. You will be asked to enter one line to complete this command.

lo

The command lists the current set of displayable objects along with their indexes.

li

The command lists all pictures that it can find in EEPROM along with their dimensions, labels, and handles.

da ***n***
dd ***n***

These commands add/delete the specified displayable object (identified by its index) to/from the display list. The addition is carried out by calling **lcdg_dm_newtop**, i.e., the added object is put on the top and is automatically displayed (assuming the LCD is on). The deletion is done with **lcdg_dm_remove**. You should refresh the display (with **or**) to see its effect on the screen.

ee [*from* [*upto*]]

This command erases the EEPROM. The arguments, which both default to zero, are directly passed as arguments to **ee_erase**. Thus, the argument-less variant erases the entire EEPROM. The command takes a while to complete during which the board appears busy to new commands.

ei ***h***

This command erases the image with the specified handle.

m

The command produces three numbers related to memory statistics: current free, minimum free, and minimum stack free (all in words).

fc n [*ev*]

The command creates a category list (a menu object). If ***ev*** is zero (or absent), the list is that of my categories, otherwise, it is event categories. The first argument is the object index, as for **ci**, **cm**, **ct**.

fr n [*c*]

The command creates a record list (a menu object). The second argument is the class identifier (0 = “ignore”, 1 = “yes”, 2 = “no”). The default is 0.

When a menu object is the top displayed object on the scree, the joystick can be used to navigate through the menu. If that menu is a record menu, by pushing the bottom button (B1) you will display the image associated with the currently selected record (if there is one) as well as the meters indicating its category membership. Note that these objects are displayed outside the hierarchy, and **`or`** will remove them.

Four more command: **`si`**, **`se`**, **`ri`**, **`re`** are in principle available in the same way as the ones described above, but you shouldn't execute them directly (you can try, but they won't amount to much). These commands provide a handshake for special commands implemented in the script whose purpose is to transfer chunks of EEPROM and images (pictures) via OEP.

To write a chunk of EEPROM on the device, you can use this script command:

```
!eeput fwa filename
!eeput fwa list_of_values
```

The first variant writes to the EEPROM all the bytes of the specified file starting at EEPROM location ***`fwa`***. The second variant can be used to directly modify a few bytes by hand. The ***`list_of_values`*** should consist of a sequence of integer (possibly hexadecimal) numbers separated by spaces that will be interpreted as bytes to be stored in EEPROM starting at location ***`fwa`***.

To dump a chunk of EEPROM, execute this:

```
!eeget fwa len [filename]
```

The first argument is the starting address in EEPROM, and the second one is the length of the area to be dumped in bytes. If a filename is specified, the EEPROM fragment will be written to that file, otherwise, the bytes will be shown on the screen as hexadecimal values.

Here is how to download a picture:

```
!imput filename
```

where filename should point to a file containing a properly formatted image (see **`LCDGInt.pdf`**). To see the handle of a newly downloaded image, you can use the **`li`** command (see above).

With this command:

```
!imget h filename
```

you can upload the image with the specified handle and store it to the indicated file.