Maemo Diablo LibOSSO
Training Material

February 9, 2009

Contents

Chapter 1

LibOSSO

1.1 Introduction to LibOSSO

LibOSSO is a library that all applications designed for maemo are expected to use. Mainly because it automatically allows the application to survive the task killing process. This task killing is done by the Desktop environment when an application launched from the Task navigator doesn't register the proper D-Bus name on the bus within a certain time limit after the launch. LibOSSO also conveniently isolates the application from possible implementation changes on D-Bus level. D-Bus used to be not API stable before as well, so LibOSSO provided "version isolation" with respect D-Bus. Since D-Bus has reached maturity (1.0), no API changes are expected for the low level library, but the GLib/D-Bus wrapper might still change at some point.

Besides the protection and isolation services, LibOSSO also provides useful utility functions to handle autosaving and state saving features of the platform, process hardware state and device mode changes and other important events happening in Internet Tablets. It also provides convenient utility wrapper functions to send RPC method calls over the D-Bus. The feature set is aimed at covering the most common GUI application needs, and as such, will not be enough in all cases. In these cases it will be necessary to use the GLib/D-Bus wrapper functions (or libdbus directly, which is not recommended).

In this material we will be concentrating on the RPC aspects of LibOSSO (and use an utility function or two as well).

1.2 Using LibOSSO for D-Bus method calls

We'll start by re-implementing the functionality from the libdbus example that we used before, but use LibOSSO functions instead of direct libdbus ones. The new version will use exactly the same D-Bus name-space components to pop up a Note dialog. LibOSSO also contains a function to do all this for you (`osso_system_note_dialog`) which we'll use directly later on. It is however instructive to see what LibOSSO provides in terms of RPC support and using a familiar RPC method is the easiest way to achieve this.

We'll start with the header section of the example program:

```c
#include <libosso.h>

  /*... Listing cut for brevity ...*/

#define SYSNOTE_NAME  "org.freedesktop.Notifications"
#define SYSNOTE_OPATH "/org/freedesktop/Notifications"
#define SYSNOTE_IFACE "org.freedesktop.Notifications"
#define SYSNOTE_NOTE  "SystemNoteDialog"
```

Listing 1.1: Header files for the simple example (libosso-example-sync/libosso-rpc-sync.c)

LibOSSO by itself only requires the libosso.h header file to be included. We'll also use the exact same D-Bus well-known name, object path, interface name and method name as before.

When reading other source code that implements or uses D-Bus services, you might sometimes wonder why the D-Bus interface name is using the same symbolic constant as the well-known name (in the above example `SYSNOTE_IFACE` would be omitted, and `SYSNOTE_NAME` would be used whenever an interface name would be required). If the service in question is not easily reusable or re-implementable, it might make sense to use an interface name that is as unique as the well-known name. This goes against the idea of defining interfaces, but is still quite common (and is the easy way out without bothering with difficult design decisions).

We continue by looking at how LibOSSO contexts are created and how they're eventually released:

```c
int main(int argc, char** argv) {

  /* The LibOSSO context that we need to do RPC. */
  osso_context_t* ossoContext = NULL;

  g_print("Initializing LibOSSO\n");
  /* The program name for registration is communicated from the
     Makefile via a -D preprocessor directive. Since it doesn't
     contain any dots in it, a prefix of "com.nokia." will be added
     to it internally within osso_initialize(). */
  ossoContext = osso_initialize(ProgName, "1.0", FALSE, NULL);
  if (ossoContext == NULL) {
    g_error("Failed to initialize LibOSSO\n");
  }

  g_print("Invoking the method call\n");
  runRPC(ossoContext);

  g_print("Shutting down LibOSSO\n");
  /* Deinitialize LibOSSO. The function doesn't return status code so
     we cannot know whether it succeeded or failed. We assume that it
     always succeeds. */
  osso_deinitialize(ossoContext);
  ossoContext = NULL;

  g_print("Quitting\n");
  return EXIT_SUCCESS;
}
```

Listing 1.2: Initialising and closing LibOSSO (libosso-example-sync/libosso-rpc-sync.c)

An LibOSSO context is a small structure containing the necessary information for the LibOSSO functions to communicate over D-Bus (both session and system buses). When a context is created, you'll need to pass your "application name" to `osso_initialize`. This name is used to register a name on the D-Bus and this will keep the task killer from killing your process later on (assuming the application was started via the Task navigator). If your application name does not contain any dot characters in it, `com.nokia.` will be prepended to it automatically in LibOSSO. The application name is normally not visible to users, so this shouldn't be a big problem. You might run into application name collisions if some other application will use the same name (and also without the dots), so it might be a good idea to provide a proper name based on a DNS domain you own or control. If you plan to implement a service to clients over the D-Bus (with `osso_rpc_set_cb`-functions) you'll need to be extra careful about the application name used here.

The version number is currently still unused, but `1.0` is recommended for now. The second to last parameter is obsolete and has no effect while the last parameter tells LibOSSO which mainloop structure to integrate to. Using `NULL` here means that LibOSSO event processing will integrate to the default GMainLoop object created (which is what you most often want).

Releasing the LibOSSO context will automatically close the connections to the D-Bus buses and release all allocated memory related to the connections and LibOSSO state. If you want to use LibOSSO functions after this, you'll have to initialize a context again.

The following snippet shows the RPC call using LibOSSO, and also contains the code suitable for using to deal with possible errors in the launch as well as the result of the RPC. The `ossoErrorStr` function is covered shortly as is the utility function to print out the result structure.

```
/**
 * Do the RPC call.
 *
 * Note that this function will block until the method call either
 * succeeds, or fails. If the method call would take a long time to
 * run, this would block the GUI of the program (which we don't have).
 *
 * Needs the LibOSSO state to do the launch.
 */
static void runRPC(osso_context_t* ctx) {

  /* Message to display. */
  const char* dispMsg = "Hello SystemNote!\nVia LibOSSO/sync.";
  /* Icon type to use. */
  gint iconType = OSSO_GN_ERROR;
  /* Button label text to use, "" means leaving the defaults. */
  const char* labelText = "";

  /* Will hold the result from the RPC invocation function. */
  osso_return_t result;
  /* Will hold the result of the method call (or error). */
  osso_rpc_t methodResult = {};

  g_print("runRPC called\n");

  g_assert(ctx != NULL);
```

```
  /* Compared to the libdbus functions, LibOSSO provides conveniently
     a function that will do the dispatch and also allows us to pass
     the arguments all with one call.

     The arguments for the "SystemNoteDialog" are the same as in
     dbus-example.c (since it is the same service). You might also
     notice that even if LibOSSO provides some convenience, it does
     not completely isolate us from libdbus. We still supply the
     argument types using D-Bus constants.

     NOTE Do not pass the argument values by pointers as with libdbus,
          instead pass them by value (as below). */
  result = osso_rpc_run(ctx,
                        SYSNOTE_NAME,      /* well-known name */
                        SYSNOTE_OPATH,         /* object path */
                        SYSNOTE_IFACE,           /* interface */
                        SYSNOTE_NOTE,          /* method name */
                        &methodResult, /* method return value */
                        /* The arguments for the RPC. The types
                           are unchanged, but instead of passing
                           them via pointers, they're passed by
                           "value" instead. */
                        DBUS_TYPE_STRING, dispMsg,
                        DBUS_TYPE_UINT32, iconType,
                        DBUS_TYPE_STRING, labelText,
                        DBUS_TYPE_INVALID);
  /* Check whether launching the RPC succeeded. */
  if (result != OSSO_OK) {
    g_error("Error launching the RPC (%s)\n",
            ossoErrorStr(result));
    /* We also terminate right away since there's nothing to do. */
  }
  g_print("RPC launched successfully\n");

  /* Now decode the return data from the method call.
     NOTE: If there is an error during RPC delivery, the return value
           will be a string. It is not possible to differentiate that
           condition from an RPC call that returns a string.

     If a method returns "void", the type-field in the methodResult
     will be set to DBUS_TYPE_INVALID. This is not an error. */
  g_print("Method returns: ");
  printOssoValue(&methodResult);
  g_print("\n");

  g_print("runRPC ending\n");
}
```

Listing 1.3: Issuing the method call and waiting for response (libosso-example-sync/libosso-rpc-sync.c)

It is important to note that `osso_rpc_run` is a synchronous (blocking) call which will wait for either the response from the method call, a timeout or an error. In our case the method will be handled quickly so it's not a big problem, but in many cases the methods take some time to execute (and might require loading external resources) so you should keep this in mind. Asynchronous LibOSSO RPC functions will be covered shortly.

If your method call will return more than one return value (this is possible in D-Bus), LibOSSO currently does not provide a mechanism to return all of them (it will return the first value only).

Decoding the result code from the LibOSSO RPC functions is pretty straight forward and is done in a separate utility:

```
/**
 * Utility to return a pointer to a statically allocated string giving
 * the textural representation of LibOSSO errors. Has no internal
 * state (safe to use from threads).
 *
 * LibOSSO does not come with a function for this, so we define one
 * ourselves.
 */
static const gchar* ossoErrorStr(osso_return_t errCode) {

  switch (errCode) {
    case OSSO_OK:
      return "No error (OSSO_OK)";
    case OSSO_ERROR:
      return "Some kind of error occurred (OSSO_ERROR)";
    case OSSO_INVALID:
      return "At least one parameter is invalid (OSSO_INVALID)";
    case OSSO_RPC_ERROR:
      return "Osso RPC method returned an error (OSSO_RPC_ERROR)";
    case OSSO_ERROR_NAME:
      return "(undocumented error) (OSSO_ERROR_NAME)";
    case OSSO_ERROR_NO_STATE:
      return "No state file found to read (OSSO_ERROR_NO_STATE)";
    case OSSO_ERROR_STATE_SIZE:
      return "Size of state file unexpected (OSSO_ERROR_STATE_SIZE)";
    default:
      return "Unknown/Undefined";
  }
}
```

Listing 1.4: Decoding LibOSSO errors (libosso-example-sync/libosso-rpc-sync.c)

Decoding the RPC return value is however slightly more complex, as the return value is a structure which contains a typed union (type is encoded in the `type` field of the structure):

```
/**
 * Utility to print out the type and content of given osso_rpc_t.
 * It also demonstrates the types available when using LibOSSO for
 * the RPC. Most simple types are available, but arrays are not
 * (unfortunately).
 */
static void printOssoValue(const osso_rpc_t* val) {

  g_assert(val != NULL);

  switch (val->type) {
    case DBUS_TYPE_BOOLEAN:
      g_print("boolean:%s", (val->value.b == TRUE)?"TRUE":"FALSE");
      break;
    case DBUS_TYPE_DOUBLE:
      g_print("double:%.3f", val->value.d);
      break;
    case DBUS_TYPE_INT32:
      g_print("int32:%d", val->value.i);
      break;
    case DBUS_TYPE_UINT32:
```

```
      g_print("uint32:%u", val->value.u);
      break;
    case DBUS_TYPE_STRING:
      g_print("string:'%s'", val->value.s);
      break;
    case DBUS_TYPE_INVALID:
      g_print("invalid/void");
      break;
    default:
      g_print("unknown(type=%d)", val->type);
      break;
  }
}
```

Listing 1.5: Decoding RPC results (libosso-example-sync/libosso-rpc-sync.c)

Note that LibOSSO RPC functions do not support array parameters either, so you're restricted to use method calls that only have simple parameters.

We build the example and then run it. The end result is the now familiar Note dialog.

```
[sbox-DIABLO_X86: ~/libosso-example-sync] > run-standalone.sh ./libosso-rpc-sync
Initializing LibOSSO
Invoking the method call
runRPC called
/dev/dsp: No such file or directory
RPC launched successfully
Method returns: uint32:8
runRPC ending
Shutting down LibOSSO
Quitting
```

Running the example

The only difference is the location of the audio device error message. It will now appear before `runRPC` returns, since `runRPC` waits for RPC completion. You never should rely on this kind of ordering, because the RPC execution could also be delayed (and the message might appear at a later location when you try this program).

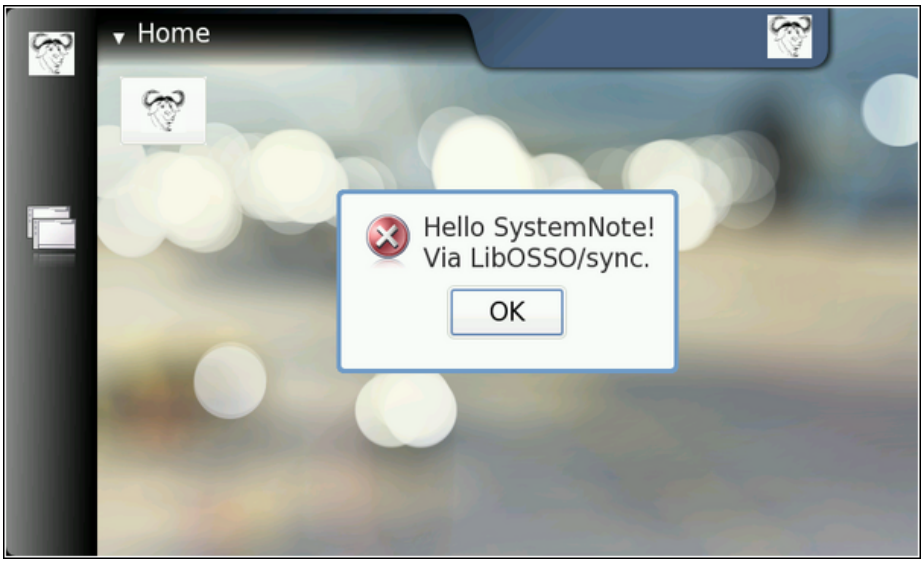


Figure 1.1: The end result

One point of interest in the Makefile is the mechanism by which the `ProgName` define is set. It is often useful to separate the program name related information outside the source code, so that the code fragment may then be re-used more easily. In this case, we control the application name that is used when LibOSSO is initialised from the Makefile.

```
# define a list of pkg-config packages we want to use
pkg_packages := glib-2.0 libosso

# ... Listing cut for brevity ...

libosso-rpc-sync: libosso-rpc-sync.c
  $(CC) $(CFLAGS) -DProgName=\"LibOSSOExample\" \
    $< -o $@ $(LDFLAGS)
```

Listing 1.6: Integrating LibOSSO into Makefiles (libosso-example-sync/Makefile)

1.3 Asynchronous method calls with LibOSSO

Sometimes the method call will take long time to run (or you're not sure whether it might take long time to run). In these cases you should use the asynchronous RPC utility functions in LibOSSO instead of the synchronous ones. The biggest difference is that the method call will be split into two parts: launching of the RPC and handling its result in a callback function. The same limitations with respect to method parameter types and the number of return values still apply.

In order for the callback to use LibOSSO functions and control the mainloop object, we'll need to create a small application state. The state will be passed to the callback when necessary.

```
/**
 * Small application state so that we can pass both LibOSSO context
 * and the mainloop around to the callbacks.
 */
typedef struct {
  /* A mainloop object that will "drive" our example. */
  GMainLoop* mainloop;
  /* The LibOSSO context which we use to do RPC. */
  osso_context_t* ossoContext;
} ApplicationState;
```

Listing 1.7: Application state for the example (libosso-example-async/libosso-rpc-async.c)

The `osso_rpc_async_run` function is used to launch the method call and it will normally return immediately. If it returns an error, it will be probably a client-side error (since the RPC method hasn't returned by then). The callback function to handle the RPC response will be registered with the function, as will the name-space related parameters and the method call arguments:

```
/**
 * We launch the RPC call from within a timer callback in order to
 * make sure that a mainloop object will be running when the RPC will
 * return (to avoid a nasty race condition).
 *
 * So, in essence this is a one-shot timer callback.
 *
 * In order to launch the RPC, it will need to get a valid LibOSSO
 * context (which is carried via the userData/application state
 * parameter).
 */
static gboolean launchRPC(gpointer userData) {

  ApplicationState* state = (ApplicationState*)userData;
  /* Message to display. */
  const char* dispMsg = "Hello SystemNote!\nVia LibOSSO/async.";
  /* Icon type to use. */
  gint iconType = OSSO_GN_ERROR;
  /* Button label text to use. */
  const char* labelText = "Execute!";

  /* Will hold the result from the RPC launch call. */
  osso_return_t result;

  g_print("launchRPC called\n");

  g_assert(state != NULL);

  /*... Listing cut for brevity ...*/

  /* The only difference compared to the synchronous version is the
     addition of the callback function parameter, and the user-data
     parameter for data that will be passed to the callback. */
  result = osso_rpc_async_run(state->ossoContext,
                              SYSNOTE_NAME,     /* well-known name */
                              SYSNOTE_OPATH,      /* object path */
                              SYSNOTE_IFACE,        /* interface */
                              SYSNOTE_NOTE,       /* method name */
                              rpcCompletedCallback,    /* async cb */
                              state,          /* user-data for cb */
```

```
                                /* The arguments for the RPC. */
                                DBUS_TYPE_STRING, dispMsg,
                                DBUS_TYPE_UINT32, iconType,
                                DBUS_TYPE_STRING, labelText,
                                DBUS_TYPE_INVALID);
  /* Check whether launching the RPC succeeded (we don't know the
     result from the RPC itself). */
  if (result != OSSO_OK) {
    g_error("Error launching the RPC (%s)\n",
            ossoErrorStr(result));
    /* We also terminate right away since there's nothing to do. */
  }
  g_print("RPC launched successfully\n");

  g_print("launchRPC ending\n");

  /* We only want to be called once, so ask the caller to remove this
     callback from the timer launch list by returning FALSE. */
  return FALSE;
}
```

Listing 1.8: Starting the RPC call (libosso-example-async/libosso-rpc-async.c)

Handling the return from the RPC method is handled by a simple callback function that will need to always use the same parameter prototype. It will receive the return value as well as the interface and method names. The latter two are useful as you can then use the same callback function to handle returns from multiple different (and simultaneous) RPC method calls.

The return value structure is allocated by LibOSSO and will be freed once your callback will return, so you don't need to handle that manually.

```
/**
 * Will be called from LibOSSO when the RPC return data is available.
 * Will print out the result, and return. Note that it must not free
 * the value, since it does not own it.
 *
 * The prototype (for reference) must be osso_rpc_async_f().
 *
 * The parameters for the callback are the D-Bus interface and method
 * names (note that object path and well-known name are NOT
 * communicated). The idea is that you can then reuse the same
 * callback to process completions from multiple simple RPC calls.
 */
static void rpcCompletedCallback(const gchar* interface,
                                 const gchar* method,
                                 osso_rpc_t* retVal,
                                 gpointer userData) {

  ApplicationState* state = (ApplicationState*)userData;

  g_print("rpcCompletedCallback called\n");

  g_assert(interface != NULL);
  g_assert(method != NULL);
  g_assert(retVal != NULL);
  g_assert(state != NULL);

  g_print(" interface: %s\n", interface);
  g_print(" method: %s\n", method);
  /* NOTE If there is an error in the RPC delivery, the return value
          will be a string. This is unfortunate if your RPC call is
```

```
          supposed to return a string as well, since it is not
          possible to differentiate between the two cases.

          If a method returns "void", the type-field in the retVal
          will be set to DBUS_TYPE_INVALID (it's not an error). */
  g_print(" result: ");
  printOssoValue(retVal);
  g_print("\n");

  /* Tell the main loop to terminate. */
  g_main_loop_quit(state->mainloop);

  g_print("rpcCompletedCallback done\n");
}
```

Listing 1.9: Handling the end of the RPC call (libosso-example-async/libosso-rpc-async.c)

In our case, receiving the response to the method call will cause the main program to be terminated.

The application setup logic is covered next:

```
int main(int argc, char** argv) {

  /* Keep the application state in main's stack. */
  ApplicationState state = {};
  /* Keeps the results from LibOSSO functions for decoding. */
  osso_return_t result;
  /* Default timeout for RPC calls in LibOSSO. */
  gint rpcTimeout;

  g_print("Initializing LibOSSO\n");
  state.ossoContext = osso_initialize(ProgName, "1.0", FALSE, NULL);
  if (state.ossoContext == NULL) {
    g_error("Failed to initialize LibOSSO\n");
  }

  /* Print out the default timeout value (which we don't change, but
     could, with osso_rpc_set_timeout()). */
  result = osso_rpc_get_timeout(state.ossoContext, &rpcTimeout);
  if (result != OSSO_OK) {
    g_error("Error getting default RPC timeout (%s)\n",
            ossoErrorStr(result));
  }
  /* Interestingly the timeout seems to be -1, but is something else
     (by default). -1 probably then means that "no timeout has been
     set". */
  g_print("Default RPC timeout is %d (units)\n", rpcTimeout);

  g_print("Creating a mainloop object\n");
  /* Create a GMainLoop with default context and initial condition of
     not running (FALSE). */
  state.mainloop = g_main_loop_new(NULL, FALSE);
  if (state.mainloop == NULL) {
    g_error("Failed to create a GMainLoop\n");
  }

  g_print("Adding timeout to launch the RPC in one second\n");
  /* This could be replaced by g_idle_add(cb, &state), in order to
     guarantee that the RPC would be launched only after the mainloop
     has started. We opt for a timeout here (for no particular
     reason). */
```

```
  g_timeout_add(1000, (GSourceFunc)launchRPC, &state);

  g_print("Starting mainloop processing\n");
  g_main_loop_run(state.mainloop);

  g_print("Out of mainloop, shutting down LibOSSO\n");
  /* Deinitialize LibOSSO. */
  osso_deinitialize(state.ossoContext);
  state.ossoContext = NULL;

  /* Free GMainLoop as well. */
  g_main_loop_unref(state.mainloop);
  state.mainloop = NULL;

  g_print("Quitting\n");
  return EXIT_SUCCESS;
}
```

Listing 1.10: Application setup timeout registration main loop and finalisation (libosso-example-async/libosso-rpc-async.c)

The code includes an example how to query the method call timeout value as well, however timeout values are left unchanged in our program.

The RPC method call is launched in a slightly unorthodox way, via a timeout call that will launch one second after mainloop processing starts. One could just as easily use `g_idle_add`, as long as the launching itself will be done after mainloop processing starts. Since the method return value callback will terminate the mainloop, the mainloop needs to be active at that point. The only way to guarantee this is to launch the RPC after the mainloop is active.

Testing the program yields little surprises (other than the default timeout value being -1):

```
[sbox-DIABLO_X86: ~/libosso-example-async] > run-standalone.sh ./libosso-rpc-async
Initializing LibOSSO
Default RPC timeout is -1 (units)
Creating a mainloop object
Adding timeout to launch the RPC in one second
Starting mainloop processing
launchRPC called
RPC launched successfully
launchRPC ending
rpcCompletedCallback called
 interface: org.freedesktop.Notifications
 method: SystemNoteDialog
 result: uint32:10
rpcCompletedCallback done
Out of mainloop, shutting down LibOSSO
Quitting
/dev/dsp: No such file or directory
```

You might notice another shift in the audio device error string. It is displayed now after all other messages (similar to the libdbus example). It seems that the audio playback is started "long after" the dialog itself is displayed, or maybe the method returns before SystemNote starts the dialog display. Again, one should not rely on exact timing when dealing with D-Bus remote method calls.

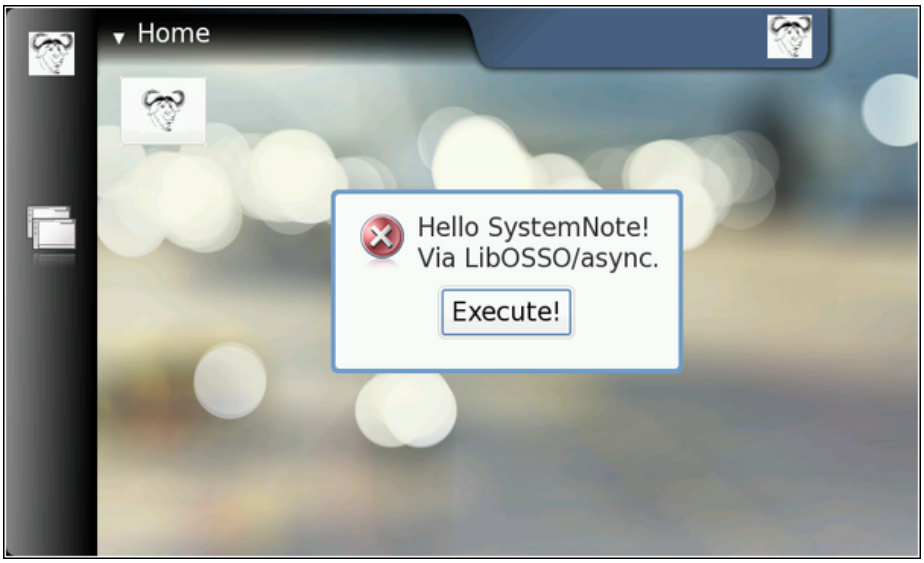


Figure 1.2: End result of the async example (no surprises)

The label text was modified slightly, to test out that non-default labels will work.

The Makefile for this example doesn't contain anything new nor special.

1.4 Device state and mode notifications

Since Internet Tablets are mobile devices, you should expect people to use them (and your software) while on the move, and also on airplanes and other places which might restrict network connectivity. If your program uses network connectivity, or wants to adapt to the conditions in the device better, you'll have to handle changes between the different devices states. The change between the states are normally initiated by the user of the device (when boarding an aircraft for example).

In order to demonstrate handling of the most important device state, we'll next implement a small utility program that will combine various utility functions from LibOSSO as well as handle the changes in the device state. The state that we're particularly interested in is the "flight" mode. This mode is initiated by the user by switching the device into "Offline"-mode. The internal name for this state however is "flight". Curiously enough there is also a mode called "offline" internally.

Our application is a simple utility program that keeps the backlight of the device turned on by periodically asking the system to delay the automatic display dimming functionality. Normally the backlight is turned off after short periods of inactivity, although this setting can be changed by the user. It is the goal of the application then to request a postponement of this mechanism (by 60 seconds at a time). We choose 45 seconds as an internal timer frequency so that we can always extend the time by 60 seconds (and be sure that we don't miss our opportunity by using a lower frequency than the maximum).

We also track the device mode, and once the device will enter the flight-mode, the program will terminate. Should the program be started when the device is already in flight-mode, the program will refuse to run.

Since the program has no GUI of its own, we'll also use Note dialogs and the infoprint facility to display status information to the user. The Note is used to remind the user that leaving the program running will exhaust the battery. The infoprints are used when the application will terminate or if it will refuse to run.

Most of the work required will be contained in the application setup logic, which allows us to reduce the code in `main` significantly:

```
/**
 * Main program:
 *
 * 1) Setup application state
 * 2) Start mainloop
 * 3) Release application state & terminate
 */
int main(int argc, char** argv) {

  /* We'll keep one application state in our program and allocate
     space for it from the stack. */
  ApplicationState state = {};

  g_print(PROGNAME ":main Starting\n");
  /* Attempt to setup the application state and if something goes
     wrong, do cleanup here and exit. */
  if (setupAppState(&state) == FALSE) {
    g_print(PROGNAME ":main Setup failed, doing cleanup\n");
    releaseAppState(&state);
    g_print(PROGNAME ":main Terminating with failure\n");
    return EXIT_FAILURE;
  }

  g_print(PROGNAME ":main Starting mainloop processing\n");
  g_main_loop_run(state.mainloop);
  g_print(PROGNAME ":main Out of main loop (shutting down)\n");
  /* We come here when the application state has the running flag set
     to FALSE and the device state changed callback has decided to
     terminate the program. Display a message to the user about
     termination next. */
  displayExitMessage(&state, ProgName " exiting");

  /* Release the state and exit with success. */
  releaseAppState(&state);

  g_print(PROGNAME ":main Quitting\n");
  return EXIT_SUCCESS;
}
```

Listing 1.11: Application main logic (libosso-flashlight/flashlight.c)

In order for the device state callbacks to force a quit of the application, we'll need to pass it the LibOSSO context. We also need access to the mainloop object and utilise a flag to tell when the timer should just quit (since timers cannot be removed externally in GLib).

```
/* Application state.
```

```
   Contains the necessary state to control the application lifetime
   and use LibOSSO. */
typedef struct {
  /* The GMainLoop that will run our code. */
  GMainLoop* mainloop;
  /* LibOSSO context that is necessary to use LibOSSO functions. */
  osso_context_t* ossoContext;
  /* Flag to tell the timer that it should stop running. Also utilized
     to tell the main program that the device is already in Flight-
     mode and the program shouldn't continue startup. */
  gboolean running;
} ApplicationState;
```

Listing 1.12: Application state (libosso-flashlight/flashlight.c)

All of the setup and start logic is implemented in `setupAppState`, and contains a significant number of steps that are all necessary:

```
/**
 * Utility to setup the application state.
 *
 * 1) Initialize LibOSSO (this will connect to D-Bus)
 * 2) Create a mainloop object
 * 3) Register the device state change callback
 *    The callback will be called once immediately on registration.
 *    The callback will reset the state->running to FALSE when the
 *    program needs to terminate so we'll know whether the program
 *    should run at all. If not, display an error dialog.
 *    (This is the case if the device will be in "Flight"-mode when
 *    the program starts.)
 * 4) Register the timer callback (which will keep the screen from
 *    blanking).
 * 5) Un-blank the screen.
 * 6) Display a dialog to the user (on the background) warning about
 *    battery drain.
 * 7) Send the first "delay backlight dimming" command.
 *
 * Returns TRUE when everything went ok, FALSE when caller should call
 * releaseAppState and terminate. The code below will print out the
 * errors if necessary.
 */
static gboolean setupAppState(ApplicationState* state) {

  osso_return_t result;

  g_assert(state != NULL);

  g_print(PROGNAME ":setupAppState starting\n");

  /* Zero out the state. Has the benefit of setting all pointers to
     NULLs and all gbooleans to FALSE. This is useful when we'll need
     to determine what to release later. */
  memset(state, 0, sizeof(ApplicationState));

  g_print(PROGNAME ":setupAppState Initializing LibOSSO\n");

  /*... Listing cut for brevity ...*/

  state->ossoContext = osso_initialize(ProgName, "1.0", FALSE, NULL);
  if (state->ossoContext == NULL) {
    g_printerr(PROGNAME ": Failed to initialize LibOSSO\n");
    return FALSE;
```

```
  }

  g_print(PROGNAME ":setupAppState Creating a GMainLoop object\n");
  /* Create a new GMainLoop object, with default context (NULL) and
     initial "running"-state set to FALSE. */
  state->mainloop = g_main_loop_new(NULL, FALSE);
  if (state->mainloop == NULL) {
    g_printerr(PROGNAME ": Failed to create a GMainLoop\n");
    return FALSE;
  }

  g_print(PROGNAME
          ":setupAddState Adding hw-state change callback.\n");
  /* The callback will be called immediately with the state, so we
     need to know whether we're in offline mode to start with. If so,
     the callback will set the running-member to FALSE (and we'll
     check it below). */
  state->running = TRUE;
  /* In order to receive information about device state and changes
     in it, we register our callback here.

     Parameters for the osso_hw_set_event_cb():
       osso_context_t* :  LibOSSO context object to use.
       osso_hw_state_t* : Pointer to a device state type that we're
                          interested in. NULL for "all states".
       osso_hw_cb_f* :    Function to call on state changes.
       gpointer :         User-data passed to callback. */
  result = osso_hw_set_event_cb(state->ossoContext,
                                NULL, /* We're interested in all. */
                                deviceStateChanged,
                                state);
  if (result != OSSO_OK) {
    g_printerr(PROGNAME
               ":setupAppState Failed to get state change CB\n");
    /* Since we cannot reliably know when to terminate later on
       without state information, we will refuse to run because of the
       error. */
    return FALSE;
  }
  /* We're in "Flight" mode? */
  if (state->running == FALSE) {
    g_print(PROGNAME ":setupAppState In offline, not continuing.\n");
    displayExitMessage(state, ProgName " not available in Offline mode"
        );
    return FALSE;
  }

  g_print(PROGNAME ":setupAppState Adding blanking delay timer.\n");
  if (g_timeout_add(45000,
                    (GSourceFunc)delayBlankingCallback,
                    state) == 0) {
    /* If g_timeout_add returns 0, it signifies an invalid event
       source id. This means that adding the timer failed. */
    g_printerr(PROGNAME ": Failed to create a new timer callback\n");
    return FALSE;
  }

  /* Un-blank the display (will always succeed in the SDK). */
  g_print(PROGNAME ":setupAppState Unblanking the display\n");
  result = osso_display_state_on(state->ossoContext);
  if (result != OSSO_OK) {
    g_printerr(PROGNAME ": Failed in osso_display_state_on (%s)\n",
```

```
                ossoErrorStr(result));
    /* If the RPC call fails, odds are that nothing else will work
       either, so we decide to quit instead. */
    return FALSE;
  }

  /* Display a "Note"-dialog with a WARNING icon.
     The Dialog is MODAL, so user cannot do anything with the stylus
     until the Ok is selected, or the Back-key is pressed. */

  /*... Listing cut for brevity ...*/

  /* Other icons available:
      OSSO_GN_NOTICE: For general notices.
     OSSO_GN_WARNING: For warning messages.
       OSSO_GN_ERROR: For error messages.
        OSSO_GN_WAIT: For messages about "delaying" for something (an
                      hourglass icon is displayed).
                   5: Animated progress indicator. */

  /*... Listing cut for brevity ...*/

  g_print(PROGNAME ":setupAppState Displaying Note dialog\n");
  result = osso_system_note_dialog(state->ossoContext,
                                   /* UTF-8 text into the dialog */
                                   "Started " ProgName ".\n"
                       "Please remember to stop it when you're done, "
                                "in order to conserve battery power.",
                                   /* Icon to use */
                                   OSSO_GN_WARNING,
                                   /* We're not interested in the RPC
                                      return value. */
                                   NULL);
  if (result != OSSO_OK) {
    g_error(PROGNAME ": Error displaying Note dialog (%s)\n",
            ossoErrorStr(result));
  }
  g_print(PROGNAME ":setupAppState Requested for the dialog\n");

  /* Then delay the blanking timeout so that our timer callback has a
     chance to run before that. */
  delayDisplayBlanking(state);

  g_print(PROGNAME ":setupAppState Completed\n");

  /* State set up. */
  return TRUE;
}
```

Listing 1.13: Setting up of the application state (libosso-flashlight/flashlight.c)

The callback to handle device state changes is registered with the `osso_hw_set_event_cb` and we also see how to force the backlight on (which is necessary so that the backlight dimming delay will accomplish something). We also register the timer callback which then will start firing away after 45 seconds and will keep delaying the backlight dimming and do the first delay so that the backlight isn't dimmed right away.

The callback function will always receive the new "hardware state" as well as the user-data. It is also somewhat interesting to note that just by registering the callback, it will be triggered immediately. This will happen even before we

have started our mainloop in order to tell the application the initial state of the device when the application starts. We utilise this to determine whether the device is already in flight-mode and refuse to start if it is. Since we don't always know whether the mainloop is active or not (the callback can be triggered later on as well), we also utilise an additional flag to communicate the timer callback that it should quit (eventually).

```
/**
 * This callback will be called by LibOSSO whenever the device state
 * changes. It will also be called right after registration to inform
 * the current device state.
 *
 * The device state structure contains flags telling about conditions
 * that might affect applications, as well as the mode of the device
 * (for example telling whether the device is in "in-flight"-mode).
 */
static void deviceStateChanged(osso_hw_state_t* hwState,
                               gpointer data) {
  ApplicationState* state = (ApplicationState*)data;

  g_print(PROGNAME ":deviceStateChanged Starting\n");

  printDeviceState(hwState);

  /* If device is in/going into "flight-mode" (called "Offline" on
     some devices), we stop our operation automatically. Obviously
     this makes flashlight useless (as an application) if someone gets
     stuck in a dark cargo bay of a plane with snakes.. But we still
     need a way to shut down the application and react to device
     changes, and this is the easiest state to test with.

     Note that since offline mode will terminate network connections,
     you will need to test this on the device itself, not over ssh. */
  if (hwState->sig_device_mode_ind == OSSO_DEVMODE_FLIGHT) {
    g_print(PROGNAME ":deviceStateChanged In/going into offline.\n");

    /* Terminate the mainloop.
       NOTE: Since this callback is executed immediately on
             registration, the mainloop object is not yet "running",
             hence calling quit on it will be ineffective! _quit only
             works when the mainloop is running. */
    g_main_loop_quit(state->mainloop);
    /* We also set the running to correct state to fix the above
       problem. */
    state->running = FALSE;
  }
}
```

Listing 1.14: Handling device state changes (libosso-flashlight/flashlight.c)

The `printDeviceState` is an utility function to decode the device state structure that the callback will be invoked with. The state contains the device mode, but also `gboolean` flags which tell the application to adapt to the environment in other ways (like memory pressure and other indicators):

```
/* Small macro to return "YES" or "no" based on given parameter.
   Used in printDeviceState below. YES is in capital letters in order
   for it to "stand out" in the program output (since it's much
   rarer). */
#define BOOLSTR(p) ((p)?"YES":"no")
```

```
/**
 * Utility to decode the hwstate structure and print it out in human
 * readable format. Mainly useful for debugging on the device.
 *
 * The mode constants unfortunately are not documented in LibOSSO.
 */
static void printDeviceState(osso_hw_state_t* hwState) {

  gchar* modeStr = "Unknown";

  g_assert(hwState != NULL);

  switch(hwState->sig_device_mode_ind) {
    case OSSO_DEVMODE_NORMAL:
      /* Non-flight-mode. */
      modeStr = "Normal";
      break;
    case OSSO_DEVMODE_FLIGHT:
      /* Power button -> "Offline mode". */
      modeStr = "Flight";
      break;
    case OSSO_DEVMODE_OFFLINE:
      /* Unknown. Even if all connections are severed, this mode will
         not be triggered. */
      modeStr = "Offline";
      break;
    case OSSO_DEVMODE_INVALID:
      /* Unknown. */
      modeStr = "Invalid(?)";
      break;
    default:
      /* Leave at "Unknown". */
      break;
  }
  g_print(
    "Mode: %s, Shutdown: %s, Save: %s, MemLow: %s, RedAct: %s\n",
    modeStr,
    /* Set to TRUE if the device is shutting down. */
    BOOLSTR(hwState->shutdown_ind),
    /* Set to TRUE if our program has registered for autosave and
       now is the moment to save user data. */
    BOOLSTR(hwState->save_unsaved_data_ind),
    /* Set to TRUE when device is running low on memory. If possible,
       memory should be freed by our program. */
    BOOLSTR(hwState->memory_low_ind),
    /* Set to TRUE when system wants us to be less active. */
    BOOLSTR(hwState->system_inactivity_ind));
}
```

Listing 1.15: Decoding the device state structure (libosso-flashlight/flashlight.c)

The delaying of the display blanking is achieved with an utility function of LibOSSO (`osso_display_blanking_pause`) which is implemented in a separate function since it's called from multiple places:

```
/**
 * Utility to ask the device to pause the screen blanking timeout.
 * Does not return success/status.
 */
static void delayDisplayBlanking(ApplicationState* state) {
```

```
  osso_return_t result;

  g_assert(state != NULL);

  result = osso_display_blanking_pause(state->ossoContext);
  if (result != OSSO_OK) {
    g_printerr(PROGNAME ":delayDisplayBlanking. Failed (%s)\n",
               ossoErrorStr(result));
    /* But continue anyway. */
  } else {
    g_print(PROGNAME ":delayDisplayBlanking RPC succeeded\n");
  }
}
```

Listing 1.16: Delaying the backlight blanking (libosso-flashlight/flashlight.c)

The timer callback will normally just ask the blanking delay to be further extended, but will also check whether the program is shutting down (by using the `running` field in the application state). If the application is indeed shutting down, the timer will ask itself to be removed from the timer queue by returning `FALSE`:

```
/**
 * Timer callback that will be called within 45 seconds after
 * installing the callback and will ask the platform to defer any
 * display blanking for another 60 seconds.
 *
 * This will also prevent the device from going into suspend
 * (according to LibOSSO documentation).
 *
 * It will continue doing this until the program is terminated.
 *
 * NOTE: Normally timers shouldn't be abused like this since they
 *       bring the CPU out from power saving state. Because the screen
 *       backlight dimming might be activated again after 60 seconds
 *       of receiving the "pause" message, we need to keep sending the
 *       "pause" messages more often than every 60 seconds. 45 seconds
 *       seems like a safe choice.
 */
static gboolean delayBlankingCallback(gpointer data) {
  ApplicationState* state = (ApplicationState*)data;

  g_print(PROGNAME ":delayBlankingCallback Starting\n");

  g_assert(state != NULL);
  /* If we're not supposed to be running anymore, return immediately
     and ask caller to remove the timeout. */
  if (state->running == FALSE) {
    g_print(PROGNAME ":delayBlankingCallback Removing\n");
    return FALSE;
  }

  /* Delay the blanking for a while still (60 seconds). */
  delayDisplayBlanking(state);

  g_print(PROGNAME ":delayBlankingCallback Done\n");

  /* We want the same callback to be invoked from the timer
     launch again, so we return TRUE. */
  return TRUE;
}
```

Listing 1.17: Application state reactive timer callback (libosso-flashlight/flashlight.c)

We also have a small utility function that will be used to display an exit message (we have two ways of exiting):

```
/**
 * Utility to display an "exiting" message to user using infoprint.
 */
static void displayExitMessage(ApplicationState* state,
                               const gchar* msg) {

  osso_return_t result;

  g_assert(state != NULL);

  g_print(PROGNAME ":displayExitMessage Displaying exit message\n");
  result = osso_system_note_infoprint(state->ossoContext, msg, NULL);
  if (result != OSSO_OK) {
    /* This is rather harsh, since we terminate the whole program if
       the infoprint RPC fails. It is used to display messages at
       program exit anyway, so this isn't a critical issue. */
    g_error(PROGNAME ": Error doing infoprint (%s)\n",
            ossoErrorStr(result));
  }
}
```

Listing 1.18: Using infoprint for non-modal notifications (libosso-flashlight/flashlight.c)

And finally we come to the application state tear-down function which will release all the resources that have been allocated by the setup function.

```
/**
 * Release all resources allocated by setupAppState in reverse order.
 */
static void releaseAppState(ApplicationState* state) {

  g_print(PROGNAME ":releaseAppState starting\n");

  g_assert(state != NULL);

  /* First set the running state to FALSE so that if the timer will
     (for some reason) be launched, it will remove itself from the
     timer call list. This shouldn't be possible since we are running
     only with one thread. */
  state->running = FALSE;

  /* Normally we would also release the timer, but since the only way
     to do that is from the timer callback itself, there's not much we
     can do about it here. */

  /* Remove the device state change callback. It is possible that we
     run this even if the callback was never installed, but it is not
     harmful. */
  if (state->ossoContext != NULL) {
    osso_hw_unset_event_cb(state->ossoContext, NULL);
  }
```

```
  /* Release the mainloop object. */
  if (state->mainloop != NULL) {
    g_print(PROGNAME ":releaseAppState Releasing mainloop object.\n");
    g_main_loop_unref(state->mainloop);
    state->mainloop = NULL;
  }

  /* Lastly, free up the LibOSSO context. */
  if (state->ossoContext != NULL) {
    g_print(PROGNAME ":releaseAppState De-init LibOSSO.\n");
    osso_deinitialize(state->ossoContext);
    state->ossoContext = NULL;
  }
  /* All resources released. */
}
```

Listing 1.19: Releasing the application state (libosso-flashlight/flashlight.c)

The Makefile for this program contains no surprises, so is not shown here. We build the program and then run it in the SDK:

```
[sbox-DIABLO_X86: ~/libosso-flashlight] > run-standalone.sh ./flashlight
flashlight:main Starting
flashlight:setupAppState starting
flashlight:setupAppState Initializing LibOSSO
flashlight:setupAppState Creating a GMainLoop object
flashlight:setupAddState Adding hw-state change callback.
flashlight:deviceStateChanged Starting
Mode: Normal, Shutdown: no, Save: no, MemLow: no, RedAct: no
flashlight:setupAppState Adding blanking delay timer.
flashlight:setupAppState Unblanking the display
flashlight:setupAppState Displaying Note dialog
flashlight:setupAppState Requested for the dialog
flashlight:delayDisplayBlanking RPC succeeded
flashlight:setupAppState Completed
flashlight:main Starting mainloop processing
/dev/dsp: No such file or directory
flashlight:delayBlankingCallback Starting
flashlight:delayDisplayBlanking RPC succeeded
flashlight:delayBlankingCallback Done
...
```

Starting the flashlight application

Figure 1.3: The application will display a modal dialog when it starts, to remind the user of the ramifications.

Since the SDK does not contain indications about backlight operations, you will not "notice" that the application is running in your screen. From the debugging messages you will see that it is. It just takes 45 seconds between each timer callback launch (and for new debug messages to appear).

1.5 Simulating device mode changes

In order to test the flashlight application without requiring a device, it is useful to know how to "simulate" device mode changes in the SDK. From the standpoint of LibOSSO (and programs that use it), it will feel and look exactly like it does on a device. When LibOSSO will receive the D-Bus signal, it will be exactly the same signal as it would be on a device. D-Bus signals are covered more thoroughly later on.

To see how this works, start flashlight in one session and leave it running (you might want to dismiss the modal dialog). Then open another session and send the signal using the system bus that signifies a device mode change (below).

```
[sbox-DIABLO_X86: ~/libosso-flashlight] > run-standalone.sh \
 dbus-send --system --type=signal /com/nokia/mce/signal \
 com.nokia.mce.signal.sig_device_mode_ind string:'flight'
```

Simulating a device mode change signal in the SDK

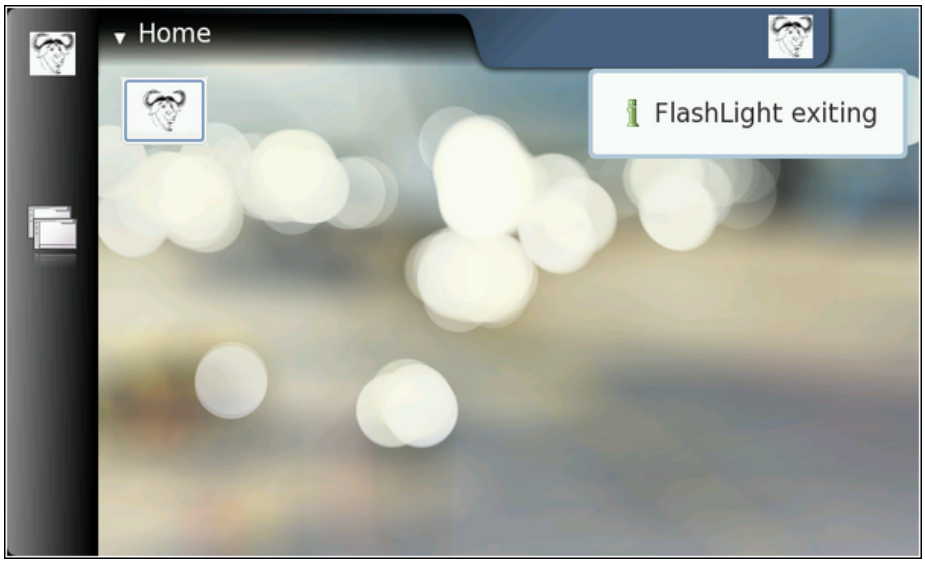
The argument for the signal is a string, stating the new device mode. They are defined (for LibOSSO) in the LibOSSO source code (src/osso-hw.c) which you can get with `apt-get source libosso`. The start of that file also defines other useful D-Bus well-known names, object paths and interfaces as well as method names relating to hardware and system-wide conditions. Here we are

only interested in switching the device mode from 'normal' to 'flight' and then back (see below).

```
flashlight:delayBlankingCallback Starting
flashlight:delayDisplayBlanking RPC succeeded
flashlight:delayBlankingCallback Done
...
flashlight:deviceStateChanged Starting
Mode: Flight, Shutdown: no, Save: no, MemLow: no, RedAct: no
flashlight:deviceStateChanged In/going into offline.
flashlight:main Out of main loop (shutting down)
flashlight:displayExitMessage Displaying exit message
flashlight:releaseAppState starting
flashlight:releaseAppState Releasing mainloop object.
flashlight:releaseAppState De-init LibOSSO.
flashlight:main Quitting
```

Reaction of flashlight to flight mode

Once the signal is sent, it will eventually be converted into a callback call from LibOSSO and our `deviceStateChanged` function gets to run. It will notice that the device mode is now `FLIGHT_MODE` and shutdown flashlight.

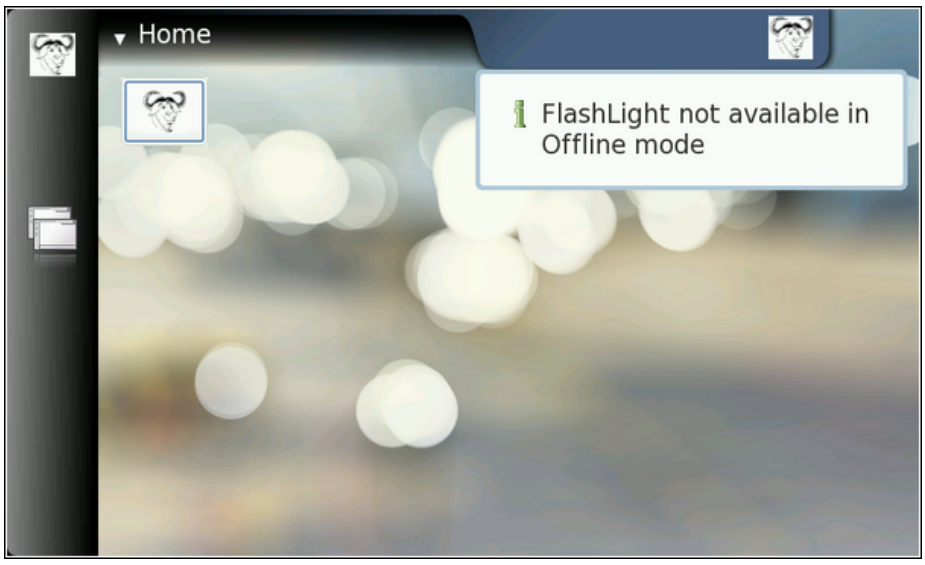


If you now start flashlight again, you will notice something peculiar. It will still see that the "device" is still in flight-mode. How convenient! This allows us to test the rest of the code paths remaining in our application (when it refuses to start if the device is already in flight mode).

```
[sbox-DIABLO_X86: ~/libosso-flashlight] > run-standalone.sh ./flashlight
flashlight:main Starting
flashlight:setupAppState starting
flashlight:setupAppState Initializing LibOSSO
flashlight:setupAppState Creating a GMainLoop object
flashlight:setupAddState Adding hw-state change callback.
flashlight:deviceStateChanged Starting
Mode: Flight, Shutdown: no, Save: no, MemLow: no, RedAct: no
flashlight:deviceStateChanged In/going into offline.
flashlight:setupAppState In offline, not continuing.
flashlight:displayExitMessage Displaying exit message
flashlight:main Setup failed, doing cleanup
flashlight:releaseAppState starting
flashlight:releaseAppState Releasing mainloop object.
flashlight:releaseAppState De-init LibOSSO.
flashlight:main Terminating with failure
```

Flashlight will refuse to start when device is in flight mode

In this case, the user is displayed the cause why flashlight cannot be started since not displaying any feedback to the user would be quite rude.



In order to return the "device" back to normal mode, you'll need to send the same signal as before (`sig_device_mode_ind`) but with the argument `normal`.

```
[sbox-DIABLO_X86: ~/libosso-flashlight] > run-standalone.sh  \
 dbus-send --system --type=signal /com/nokia/mce/signal  \
 com.nokia.mce.signal.sig_device_mode_ind string:'normal'
```

Restoring device mode back to normal