Maemo Diablo Source code for flashlight

Training Material

February 9, 2009

Contents

1 Source code for flashlight **2**
1.1 libosso-flashlight/flashlight.c 2
1.2 libosso-flashlight/Makefile 11

Chapter 1

Source code for flashlight

1.1 libosso-flashlight/flashlight.c

```
/**
 * A simple LibOSSO non-GUI program.
 *
 * This maemo code example is licensed under a MIT-style license,
 * that can be found in the file called "License" in the same
 * directory as this file.
 * Copyright (c) 2007-2008 Nokia Corporation. All rights reserved.
 *
 * Demonstrates how to utilize LibOSSO for trivial requests as well as
 * how to receive state change notifications from the system.
 *
 * The program will use LibOSSO to enable the backlight on the screen
 * of a device, and then periodically extend the backlight timeout
 * mechanism in the device, in order to keep the backlight on. As
 * such, it can be used a "flashlight"-replacement, but really is just
 * a demonstration program about using LibOSSO.
 *
 * The program will either refuse to run, or stop, when the device is
 * placed into "Flight"-mode. This is achieved by utilizing the
 * LibOSSO device state change callbacks.
 *
 * Other demonstrations include displaying Note dialogs to user from
 * a non-GUI program, utilizing the infoprint component and LibOSSO
 * error and state decoding.
 *
 * If you really want a "flashlight" application, you should consider
 * writing a small Hildon application that will display a fully white
 * GtkCanvas and switch it to run in full screen mode. You might also
 * want to maximize the backlight for the duration of the program. The
 * brightness is controlled via a GConf key (an integer from 1 to 9)
 * /system/osso/dsm/display/display_brightness and changing that key
 * will affect brightness automagically (assuming the backlight
 * dimming timer has not expired, but that is what this program stops
 * from happening).
 *
 * NOTE: Keeping the screen and backlight on without a good reason is
 *       not a good idea as this drains the battery on the device.
 *
 * NOTE: When running this on the SDK, you will need to use the
 *       run-standalone.sh script, otherwise LibOSSO initialization
```

```
 *       will fail. The RPC method calls will all succeed in the
 *       SDK (assuming you have AF running), but since there is no
 *       real "screen blank/backlight dimming" functionality in the
 *       SDK, you won't see any changes. Also, it is not possible to
 *       simulate state change events in the SDK (so you will have to
 *       terminate the program using Ctrl+c).
 */

#include <glib.h>
#include <libosso.h>
#include <stdlib.h> /* EXIT_SUCCESS */
#include <string.h> /* memset */

/* 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;

/**
 * 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";
  }
}

/**
 * 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");
  }
}

/**
 * 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;
}

/* 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));
}

/**
 * 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;
  }
}

/**
 * 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));
  }
}

/**
 * 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");
  /* Parameters for osso_initialize:
     gchar* : Application "name". This is not the name that will be
              visible to user, but rather the name that other programs
              can use to communicate with this program (eventually
              over D-Bus). Note that if your name does _not_ include
              dot, 'com.nokia.' will be prepended to the name
              automatically.
     gchar* : Application version.
     gboolean : Unused / no effect.
     GMainContext* : Context under which LibOSSO will integrate into.
              Leave to NULL in order to use the default context (which
              will be true for programs that use one GMainLoop, from
              the default context). */
  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. */

/* In most cases, you should avoid displaying modal dialogs,
   especially when running without a normal GUI (like this program
   is). Instead, you might want to use osso_system_note_infoprint()
   instead.

   In our case, we really really want the user to realize that
   the program will cause extra battery usage. Hence the warning,
   and a modal note dialog. If you run this on a device, you might
   also hear a special alerting sound. */

/* 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. */

  /* The text must be UTF-8 formatted, and may contain newlines, but
     other markup is not supported (and you should limit the amount of
     text that you will put into the dialog).

     If we'd be interested in the return data from the RPC method,
     we could pass a pointer to a osso_rpc_t where the result would
     be then stored.

     Unfortunately for us, the Note dialog returns "void" and returns
     it immediately. This means that the dialog will be waiting for
     the user to press "Ok", but we won't get notified about that.
     In fact, we'll continue running immediately after asking for the
     dialog. This is common to most of the convenience wrappers in
     LibOSSO.

     So, we pass NULL to tell LibOSSO not to bother with the return
     value. */

  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;
}

/**
 * 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. */
}

/**
 * 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 1.1: libosso-flashlight/flashlight.c

1.2 libosso-flashlight/Makefile

```
#
# Makefile for the simple LibOSSO flashlight program
#

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

PKG_CFLAGS  := $(shell pkg-config --cflags $(pkg_packages))
PKG_LDFLAGS := $(shell pkg-config --libs $(pkg_packages))
# Add warnings and debugging info
ADD_CFLAGS := -g -Wall

# Combine
CFLAGS  := $(PKG_CFLAGS) $(ADD_CFLAGS) $(CFLAGS)
LDFLAGS := $(PKG_LDFLAGS) $(LDFLAGS)

targets = flashlight

all: $(targets)

# For simple one-file programs, combining both compiling and linking
# makes sense.
#
# PROGNAME = name of program to prefix to all printout in program
# ProgName = name of program to use for LibOSSO registration and user
#            visible dialogs/infoprints.
flashlight: flashlight.c
  $(CC) $(CFLAGS) -DPROGNAME=\"$@\" -DProgName=\"FlashLight\" \
    $< -o $@ $(LDFLAGS)

.PHONY: clean all
clean:
  $(RM) $(targets)
```

Listing 1.2: libosso-flashlight/Makefile