Maemo Diablo Reference Manual for maemo 4.1

Using Multimedia Components

December 22, 2008

Contents

Chapter 1

Using Multimedia Components

1.1 Introduction

The following code examples are used in this chapter:

- example_wavlaunch.c
- example_camera.c
- crazyparking

Maemo offers lots of possibilities for multimedia applications. Maemo was built with the Internet Tablet devices in mind - hardware in this form factor provides a big high resolution touch screen, audio input and output, video and image capturing through the camera, fast network connections for streaming and a digital signal processor for efficient audio and video manipulation.

For developers, there are open programming interfaces to make use of these features apart from directly programming the DSP. The preferred way of doing video and audio programming for maemo is using the GStreamer framework. For manipulating images there are several libraries. Interactive multimedia for applications like games can be done using SDL framework.

GStreamer - Multimedia Framework

The main multimedia framework in maemo is GStreamer. It is based on the concept of pipelines, which are made of multiple elements. Elements themselves can be practically anything that do something with a data stream.

Maemo includes a variety of these elements to support audio and video effects, encoding and decoding and interfacing with the device's hardware. Using this approach, a developer does not have to bother with details like low level hardware management, audio and video compression schemes etc. If the elements included with the OS are not enough, more can be either compiled for maemo, or new ones can be developed.

Developer documentation for GStreamer can be found at the project's website[4].

Stream Encoding and Decoding

Maemo devices include a wide arsenal of video and audio codecs, which enable the maemo applications to read and write nearly all commonly used video and audio formats. These are supported also by the GStreamer framework. For many purposes the developer does not even have to specify the used codecs implicitly, since GStreamer automatically detects the format and the codec to use. The playbin[6] GStreamer base plug-in provides a convenient abstraction layer for all audio and video content.

Due to non-technical reasons, most of the codecs are not distributed with the SDK. This is good to keep in mind, when developing applications relying on such features.

Digital Signal Processor

Inside Internet Tablets, there is a dedicated digital signal processor (DSP). Its design is optimized for tasks like stream coding and audio effects. Maemo has a high level programming support for the DSP in form of GStreamer elements, which can decode most of the supported file formats. By using the DSP, the computing load is also taken off from the main processor, greatly enhancing system performance and responsivity.

Audio

For audio programming, maemo has two main APIs: GStreamer and ESound. Usually system sounds, such as sounds for notifying user of an event, e.g. battery low, is played through ESound. More sophisticated operations, e.g. playing music files or recording audio, should be generally performed using GStreamer, which provides better performance and a much more flexible API. Most of maemo's computing intensive GStreamer elements are implemented using the device's DSP, which greatly enhances their performance.

Linux kernel has also two lower level audio interfaces: ALSA and OSS. Of these, ALSA is supported through a plug-in package that is part of the SDK. The legacy API OSS is not supported by the kernel, but ALSA has an OSS emulation system that works for most purposes.

For the audio APIs' documentation, see the GStreamer web site[4], ESound white paper[3] and ALSA project's web site[1].

Video

Although the framework hides much of the implementation and hardware details, it is good for a developer to know what happens beneath the interfaces. Video capturing is performed via Linux kernel's Video4Linux API, and graphics are displayed using the X Window System. Practically all GNU/Linux applications rely on these components for video tasks, so porting existing applications should be quite effortless, and support easy to find.

Hands-on instructions for using capture and output features are given in section 1.3.

Graphics and Images

Maemo includes several libraries for working with images and graphics:

- Cairo is a library for making 2D vector graphics. It features, for example, high quality vector-based graphics, support for importing and exporting

a variety of formats, such as SVG, PNG, PDF and Postscript, and bindings for many programming languages

- GDK and GdkPixbuf are the bitmap graphics libraries that GTK+ is built on. Together, they provide opening and saving for bitmap images in JPEG, PNG, TIFF, ICO and BMP formats, tight integration into GTK+ and tools for drawing with graphic primitives. Also loading of SVG images is supported
- For more detailed control over the different image formats, there are many more specialized libraries, e.g. libpng, libjpeg and libwmf.

Visit Cairo website[2] for guides, tutorials and references. GDK's and GdkPixbuf's documentation can be found at GTK+ project's website[9].

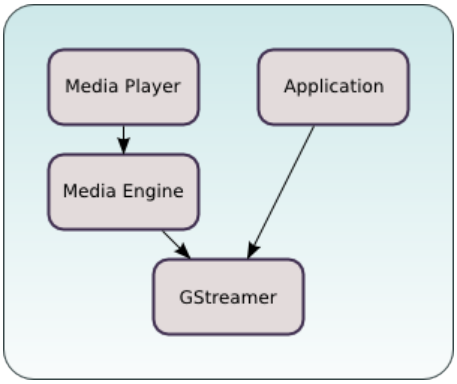
Games

For game developers, maemo offers a common start-up and settings screen, and a framework called *osso-games-startup*. Using this library, game developers and porters can reduce the common start-up screen coding, and concentrate on the real game programming. Osso-games-startup's operating system communication features also offer an interface to make games behave correctly. It eases, for example, the management of full screen mode and exceptional situations, such as battery low notification, in a unified and user friendly manner. See section 1.4 for more in-depth description and API usage.

1.2 Getting Started with Multimedia

This section explains how to get started developing multimedia applications and plug-ins in the maemo SDK. It is recommended to read also GStreamer's documentation.

Here is a diagram of the multimedia architecure from the device's Media Player point of view. It is important, if planning to develop GStreamer plug-ins that can be used by the Media Player.



1.2.1 Simple GStreamer Example

This example demonstrates how to use GStreamer's API to play PCM audio.

First, get the sample applications (including build scripts), for instance with the following command:

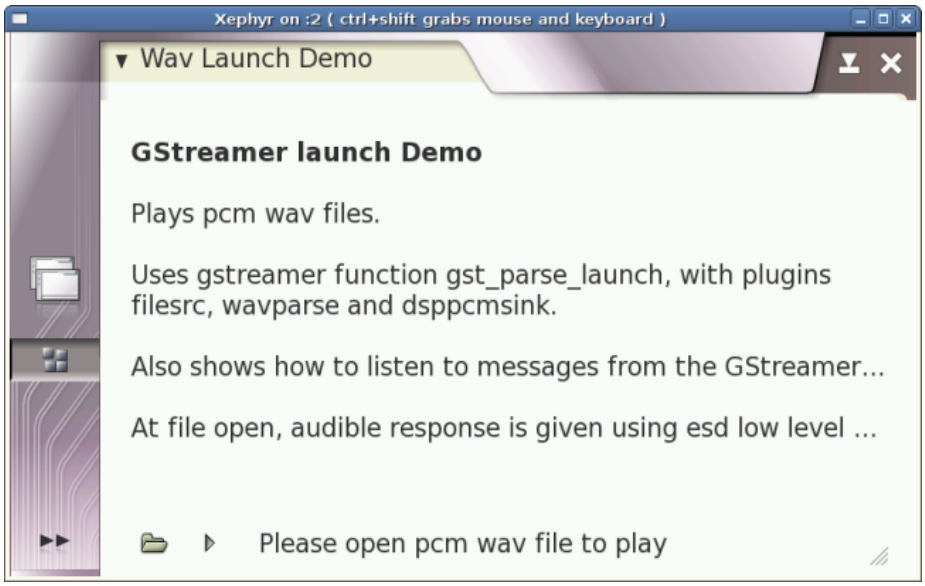
```
svn checkout https://garage.maemo.org/svn/maemoexamples/tags/maemo_4.1/maemo-examples/
```

To compile and run, start Xephyr and execute the following commands:

```
cd maemo-examples
make
af-sb-init.sh start
run-standalone.sh ./example_wavlaunch
```

To use, simply:

- Click Open to open the File Chooser.
- Browse to a wav file and select Open.
- Select Play to start playing.



Many aspects of GStreamer's application development are described in the GStreamer Application Development Manual [5].

For more information on how to use GStreamer's API, see the Core API Reference [8].

1.2.2 Plug-in Development

In order to have support for a new media format in the maemo SDK, it is necessary to compile a codec for the format, and compile the GStreamer plug-in supporting that codec.

Codecs are libraries that enable the use of compression for digital audio and video. GStreamer plug-ins are loadable libraries, which provide GStreamer elements that process video and audio streams.

Some plug-ins have the codec embedded, and therefore do not need an additional library. To play some unusual formats, a demuxer GStreamer plug-in

is needed as well. More information about the internals of GStreamer plug-ins can be found in the Plugin Writers Guide [7].

The list of plug-ins for GStreamer is available here.

To add support for a new media format:

- Get the codec from the codec's manufacturer.
- Get the GStreamer plug-ins from the GStreamer website.
- Extract packages to the Scratchbox environment and follow the compiling instructions for the codec and the GStreamer plug-ins package

1.2.3 Installing OGG Vorbis

To enable playback for ogg-vorbis audio files on the platform, take the following steps:

1. Get the integer-only implementation from here.

```
svn co http://svn.xiph.org/trunk/Tremor/
```

2. Build and install in Scratchbox.

```
./autogen.sh --prefix=/usr
make install
```

3. Get gst-plugins-bad-0.10.5.
4. Build and install in Scratchbox:

```
./configure --prefix=/usr
make -C ext/ivorbis install
```

5. Check that it is there (if you have gstreamer-tools):

```
gst-inspect-0.10 tremor
```

6. If you want to try it:

```
gst-launch-0.10 filesrc location=test.ogg ! application/ogg ! tremor ! alsasink
```

1.2.4 Deployment

Copy files from scratchbox to the device (into the same directory):

- `/usr/lib/libvorbisidec.so* -> /usr/lib/`
- `/usr/lib/gstreamer-0.10/libgstivorbis.so -> /usr/lib/gstreamer-0.10`
- If you want to try it:

```
gst-launch-0.10 filesrc location=test.ogg ! application/ogg ! tremor ! dsppcmsink
```

The next step is to tell the file manager and the media player about the new format.

1. Edit /usr/share/mime/packages/ogg-vorbis.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<mime-info xmlns="http://www.freedesktop.org/standards/shared-mime
    -info">
  <mime-type type="audio/x-vorbis">
    <glob pattern="*.ogg"/>
    <magic priority="50">
      <match type="string" value="OggS" offset="0"/>
    </magic>
    <comment>OGG Vorbis audio</comment>
  </mime-type>
</mime-info>
```

N.B. The mime type needs to start with "audio/" to be properly recognized. The same applies to video codecs ("video/").

Update the MIME database:

```
update-mime-database /usr/share/mime
```

2. Add "audio/x-vorbis" to /usr/share/applications/hildon/mp_ui.desktop

   Update the Desktop database:

   ```
   update-desktop-database
   ```

3. Add ogg to the libmetalayer configuration /usr/share/libmetalayer/metadata_lib.conf

   ```
   ogg libmtext_gst
   ```

   Reload metalayer crawler

   ```
   /etc/init.d/metalayer-crawler0 restart
   ```

N.B.

This information may contain references to third-party information, software and/or services. Such third-party references in this material do not imply endorsement by Nokia of the third party in any way. Your use of third party software shall be governed by and subject to you agreeing to the terms of separate software licenses and Nokia shall not be responsible or liable for any part of such dealings.

1.3 Camera API Usage

This section explains how to use the Camera API to access the camera hardware that is present in some models of Nokia Internet Tablets.

1.3.1 Camera Hardware and Linux

The Linux operating system supports live video and audio hardware, such as webcams, TV tuners, video capture cards, FM radio tuners, video *output* devices etc. The primary API for the applications to access those devices is Video4Linux.

Video4Linux is a *kernel* API, so there must be kernel drivers for each supported device. At the user level, device access is standardized via device files. In the case of video capture devices like cameras, which are the focus of this material, the files would be */dev/video0*, */dev/video1*, etc., as many as there are connected devices.

Data exchanged between the device file and user-level application has a standardized format for each device class. This allows the application to be instantly compatible with every video capture device that has a driver for Linux.

The built-in camera present in some Nokia Internet Tablet devices is compatible with Video-4-Linux version 2 API [10]. In principle, any application compatible with this API is easily portable to the maemo platform.

Since the maemo platform delegates all multimedia handling to the GStreamer framework, applications that need access to the built-in camera should employ GStreamer for this, instead of directly accessing Video4Linux devices, via the *v4l2src* GStreamer module.

Thanks to the flexibility of GStreamer, a developer can fully test any given application in a regular desktop PC with a connected webcam, and then perform the final test in the Internet Tablet itself, without a single change in the source code, since GStreamer refers to modules as text names.

One important note about the camera in the Tablet: only one application can use it at any given time. So, while using the camera in an application, other tasks that could possibly make use of it (e.g. a video call) will be blocked.

To demonstrate how the camera manipulation is performed, an example application is provided and discussed.

1.3.2 Camera Manipulation in C Language

This C application allows to use the Internet Tablet as a "mirror" (i.e. showing the camera input in the screen), as well as allows to take pictures to be saved as JPEG files (which illustrates the video frame buffer manipulation).

In this example, the function initialize_pipeline() is most interesting, since it is responsible for creating the GStreamer pipeline, sourcing data from Video4Linux and sinking it to a *xvimagesink* (which is an optimized X framebuffer). The pipeline scheme is as follows:

```
/* Initialize the the Gstreamer pipeline. Below is a diagram
 * of the pipeline that will be created:
 *
 *                              |Screen|   |Screen|
 *                            ->|queue |->|sink   |-> Display
 * |Camera|   |CSP    |   |Tee|/
 * |src    |->|Filter|->|    |\   |Image|    |Image |   |Image|
 *                            ->|queue|-> |filter|->|sink |-> JPEG file
 */
```

Listing 1.1: example_camera.c

Between the source and sinks, there are two *ffmpegcolorspace* filters, one to configure the camera frame rate and the picture size expected by the JPEG encoder, and the second to satisfy the video sink. Capabilities ("caps") are employed to tell which format the data needs to have when exiting the filter.

The second filter is necessary, since the video sink may have different requirements (bit depth, color space) from the JPEG encoder. Those requirements can vary also according to the hardware.

Because there are two sinks, the queues are important, since they guarantee that each pipeline segment operates on its own thread downstream the queue. This ensures that the different sinks can synchronize without waiting for each other.

This sample application is not different from other GStreamer applications, be it Linux-generic or maemo-specific apps:

```
static gboolean initialize_pipeline(AppData *appdata,
    int *argc, char ***argv)
{
  GstElement *pipeline, *camera_src, *screen_sink, *image_sink;
  GstElement *screen_queue, *image_queue;
  GstElement *csp_filter, *image_filter, *tee;
  GstCaps *caps;
  GstBus *bus;


  /* Initialize Gstreamer */
  gst_init(argc, argv);

  /* Create pipeline and attach a callback to it's
   * message bus */
  pipeline = gst_pipeline_new("test-camera");

  bus = gst_pipeline_get_bus(GST_PIPELINE(pipeline));
  gst_bus_add_watch(bus, (GstBusFunc)bus_callback, appdata);
  gst_object_unref(GST_OBJECT(bus));

  /* Save pipeline to the AppData structure */
  appdata->pipeline = pipeline;

  /* Create elements */
  /* Camera video stream comes from a Video4Linux driver */
  camera_src = gst_element_factory_make(VIDEO_SRC, "camera_src");
  /* Colorspace filter is needed to make sure that sinks understands
   * the stream coming from the camera */
  csp_filter = gst_element_factory_make("ffmpegcolorspace", "csp_filter
      ");
  /* Tee that copies the stream to multiple outputs */
  tee = gst_element_factory_make("tee", "tee");
  /* Queue creates new thread for the stream */
  screen_queue = gst_element_factory_make("queue", "screen_queue");
  /* Sink that shows the image on screen. Xephyr doesn't support XVideo
   * extension, so it needs to use ximagesink, but the device uses
   * xvimagesink */
  screen_sink = gst_element_factory_make(VIDEO_SINK, "screen_sink");
  /* Creates separate thread for the stream from which the image
   * is captured */
  image_queue = gst_element_factory_make("queue", "image_queue");
  /* Filter to convert stream to use format that the gdkpixbuf library
   * can use */
  image_filter = gst_element_factory_make("ffmpegcolorspace", "
      image_filter");
  /* A dummy sink for the image stream. Goes to bitheaven */
  image_sink = gst_element_factory_make("fakesink", "image_sink");

  /* Check that elements are correctly initialized */
  if(!(pipeline && camera_src && screen_sink && csp_filter &&
```

```
    screen_queue
  && image_queue && image_filter && image_sink))
{
  g_critical("Couldn't create pipeline elements");
  return FALSE;
}

/* Set image sink to emit handoff-signal before throwing away
 * it's buffer */
g_object_set(G_OBJECT(image_sink),
    "signal-handoffs", TRUE, NULL);

/* Add elements to the pipeline. This has to be done prior to
 * linking them */
gst_bin_add_many(GST_BIN(pipeline), camera_src, csp_filter,
    tee, screen_queue, screen_sink, image_queue,
    image_filter, image_sink, NULL);

/* Specify what kind of video is wanted from the camera */
caps = gst_caps_new_simple("video/x-raw-rgb",
    "width", G_TYPE_INT, 640,
    "height", G_TYPE_INT, 480,
    NULL);


/* Link the camera source and colorspace filter using capabilities
 * specified */
if(!gst_element_link_filtered(camera_src, csp_filter, caps))
{
  return FALSE;
}
gst_caps_unref(caps);

/* Connect Colorspace Filter -> Tee -> Screen Queue -> Screen Sink
 * This finalizes the initialization of the screen-part of the
     pipeline */
if(!gst_element_link_many(csp_filter, tee, screen_queue, screen_sink,
     NULL))
{
  return FALSE;
}

/* gdkpixbuf requires 8 bits per sample which is 24 bits per
 * pixel */
caps = gst_caps_new_simple("video/x-raw-rgb",
    "width", G_TYPE_INT, 640,
    "height", G_TYPE_INT, 480,
    "bpp", G_TYPE_INT, 24,
    "depth", G_TYPE_INT, 24,
    "framerate", GST_TYPE_FRACTION, 15, 1,
    NULL);

/* Link the image-branch of the pipeline. The pipeline is
 * ready after this */
if(!gst_element_link_many(tee, image_queue, image_filter, NULL))
    return FALSE;
if(!gst_element_link_filtered(image_filter, image_sink, caps)) return
     FALSE;

gst_caps_unref(caps);

/* As soon as screen is exposed, window ID will be advised to the
```

```
      sink */
  g_signal_connect(appdata->screen, "expose-event", G_CALLBACK(
      expose_cb),
       screen_sink);

  gst_element_set_state(pipeline, GST_STATE_PLAYING);

  return TRUE;
}
```

Listing 1.2: example_camera.c

The following function is called back when the user has pressed the "Take photo" button, and the image sink has data. It will forward the image buffer to *create_jpeg()*:

```
/* This callback will be registered to the image sink
 * after user requests a photo */
static gboolean buffer_probe_callback(
    GstElement *image_sink,
    GstBuffer *buffer, GstPad *pad, AppData *appdata)
{
  GstMessage *message;
  gchar *message_name;
  /* This is the raw RGB-data that image sink is about
   * to discard */
  unsigned char *data_photo =
      (unsigned char *) GST_BUFFER_DATA(buffer);

  /* Create a JPEG of the data and check the status */
  if(!create_jpeg(data_photo))
    message_name = "photo-failed";
  else
    message_name = "photo-taken";

  /* Disconnect the handler so no more photos
   * are taken */
  g_signal_handler_disconnect(G_OBJECT(image_sink),
      appdata->buffer_cb_id);

  /* Create and send an application message which will be
   * catched in the bus watcher function. This has to be
   * sent as a message because this callback is called in
   * a gstreamer thread and calling GUI-functions here would
   * lead to X-server synchronization problems */
  message = gst_message_new_application(GST_OBJECT(appdata->pipeline),
      gst_structure_new(message_name, NULL));
  gst_element_post_message(appdata->pipeline, message);

  /* Returning TRUE means that the buffer can is OK to be
   * sent forward. When using fakesink this doesn't really
   * matter because the data is discarded anyway */
  return TRUE;
}
```

Listing 1.3: example_camera.c

The *xvimagesink* GStreamer module will normally create a new window just for itself. Since the video is supposed to be shown inside the main application window, the X-Window window ID needs to be passed to the module, as soon as the ID exists:

```
/* Callback to be called when the screen-widget is exposed */
static gboolean expose_cb(GtkWidget * widget, GdkEventExpose * event,
    gpointer data)
{

  /* Tell the xvimagesink/ximagesink the x-window-id of the screen
   * widget in which the video is shown. After this the video
   * is shown in the correct widget */
  gst_x_overlay_set_xwindow_id(GST_X_OVERLAY(data),
            GDK_WINDOW_XWINDOW(widget->window));
  return FALSE;
}
```

Listing 1.4: example_camera.c

For the sake of completeness, it follows the JPEG encoding function. It is worthwhile to mention that the buffer that came from GStreamer is a simple linear framebuffer:

```
/* Creates a jpeg file from the buffer's raw image data */
static gboolean create_jpeg(unsigned char *data)
{
  GdkPixbuf *pixbuf = NULL;
  GError *error = NULL;
  guint height, width, bpp;
  const gchar *directory;
  GString *filename;
  guint base_len, i;
  struct stat statbuf;

  width = 640; height = 480; bpp = 24;

  /* Define the save folder */
  directory = SAVE_FOLDER_DEFAULT;
  if(directory == NULL)
  {
    directory = g_get_tmp_dir();
  }


  /* Create an unique file name */
  filename = g_string_new(g_build_filename(directory,
      PHOTO_NAME_DEFAULT, NULL));
  base_len = filename->len;
  g_string_append(filename, PHOTO_NAME_SUFFIX_DEFAULT);
  for(i = 1; !stat(filename->str, &statbuf); ++i)
  {
    g_string_truncate(filename, base_len);
    g_string_append_printf(filename, "%d%s", i,
        PHOTO_NAME_SUFFIX_DEFAULT);
  }


  /* Create a pixbuf object from the data */
  pixbuf = gdk_pixbuf_new_from_data(data,
      GDK_COLORSPACE_RGB, /* RGB-colorspace */
      FALSE, /* No alpha-channel */
      bpp/3, /* Bits per RGB-component */
      width, height, /* Dimensions */
      3*width, /* Number of bytes between lines (ie stride) */
      NULL, NULL); /* Callbacks */
```

```
  /* Save the pixbuf content's in to a jpeg file and check for
   * errors */
  if(!gdk_pixbuf_save(pixbuf, filename->str, "jpeg", &error, NULL))
  {
    g_warning("%s\n", error->message);
    g_error_free(error);
    gdk_pixbuf_unref(pixbuf);
    g_string_free(filename, TRUE);
    return FALSE;
  }

  /* Free allocated resources and return TRUE which means
   * that the operation was succesful */
  g_string_free(filename, TRUE);
  gdk_pixbuf_unref(pixbuf);
  return TRUE;
}
```

Listing 1.5: example_camera.c

1.4 Using Games Start-up Screen

The osso-games-startup application is a generic game start-up interface, providing a common hildonized user interface view for game start-up control and configuration.

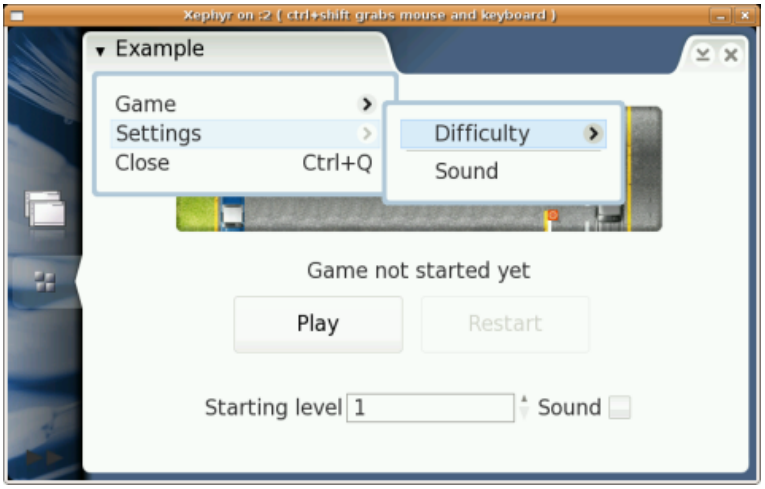


Figure 1.1: Basic osso-games-startup application screen

N.B. The older maemo-games-startup has been replaced by osso-games-startup that is covered in this material.

1.4.1 Application Functionality

To execute the game, the osso-games-startup application should be called. The game configuration file should be passed as an argument. Once loaded, osso-games-startup will create a common interface for all games (see figure 1.1)

and, if needed, will load a specific plug-in for each game. Games are activated through an auto-activating D-BUS message, which tells the game either to start, restart or to continue. In cases where no clean-up routine within the plug-in exists, it can also start the game to clean its state data. In these cases, the game usually does not open its own window, but kills the state data in the background instead.

When the Play button is pressed, the D-BUS service defined in the configuration file will be executed. Games that do not use the glib mainloop must integrate some functionality to their mainloop (for more information about games without a glib mainloop see section 1.4.2).

The service is called every time any communication between osso-games-startup and the game is needed.

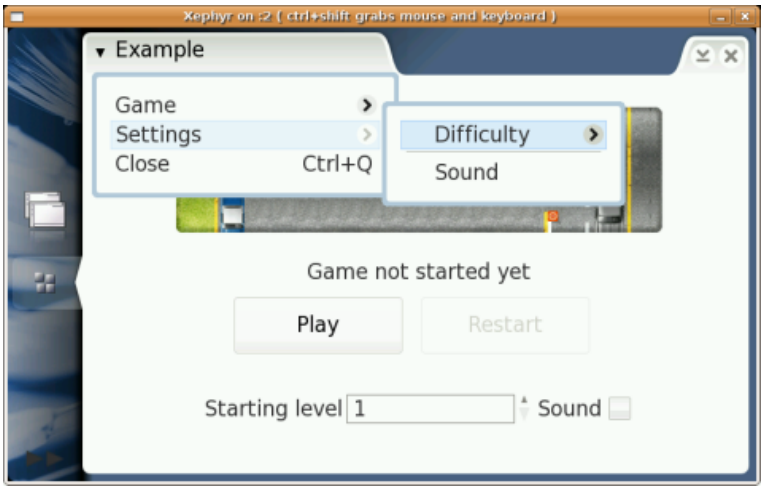


Figure 1.2: The Games Start-up screen with a simple plug-in containing difficulty level and sound configuration

Diagram 1.3 illustrates the operational execution flow of the osso-games-startup application.

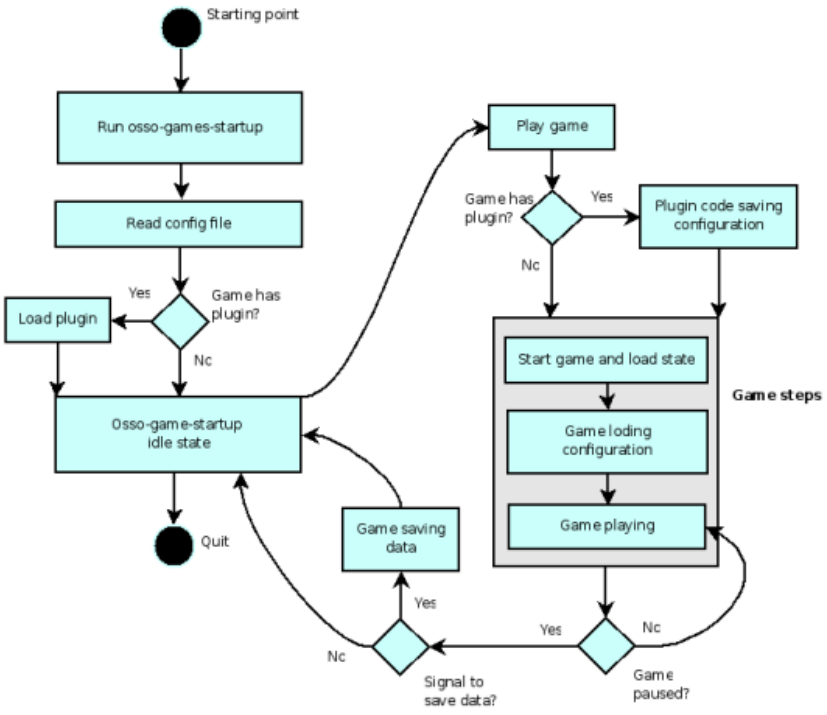


Figure 1.3: Activity Diagram

1.4.2 Integration

The integration tasks depend on the toolkit the game is based on. The following sections describe the integration tasks for:

- Games based on GTK+
- Games based on other toolkits, such as SDL

Integrating GTK+ Games

To integrate games based on GTK+, a configuration file should be created, defining the information necessary for the osso-games-startup application. The following example illustrates the contents of the configuration file:

```
[Startup Entry]
# the name of the application
Name=example
# the current version of the application
Version=0.2.0
# the title that will be used, if not specified by the GettextPackage
Title=Example
# the GettextPackage to be used to locate the title string
GettextPackage=example
# if the TitleId is defined, it will search for it inside  the gettext package
TitleId=example_title
# if the game has a plug-in
PluginPath=/usr/share/example/example_plugin.so
# the games-startup screen image
Image=/usr/share/example/pixmaps/example.png
# the D-BUS service, path and interface names
ServiceName=com.domain.example
PathName=/com/domain/example
InterfaceName=com.domain.example
```

The game should be integrated with libosso to enable it to receive and send D-BUS messages, such as "pause", "restart" and "continue". In order to receive messages from the osso-games-startup application, the game must register the service as defined in the configuration file.

When sending messages to osso-games-startup, the game must use the service, path and interface name defined in the configuration file, but with a "_startup" suffix.

Then a D-BUS service (a .service file) should be created, using the name of the service that executes the game binary, as defined in the configuration file. The following example illustrates the contents of the file:

```
[D-BUS Service]
Name=com.domain.example
Exec=/usr/games/wrapper/games/wrapper.l
```

Finally, the game must create a shell script that calls the osso-games-startup application executable, and passes the game configuration file as an argument. The following example illustrates the contents of the script:

```
/usr/bin/osso_games_startup /usr/share/example/example.conf
```

Integrating Non-GTK+ Games

In order to integrate non-GTK+ applications to osso-games-startup, the only thing different is the actual game implementation. The same steps should be followed to create osso desktop files and registering dbus services as for GTK+ based games; the only difference is that Nokia is providing a hildon-games-wrapper library to deal with dbus messaging without the use of libosso.

To initialize game application to receive events from osso-games-startup, the developer can use following approach:

```
#include <hgw/hgw.h>
/* snip */
HgwContext *hgw;
hgw = hgw_context_init();
```

If and when hgw pointer is not null, the game is ready to receive dbus messages that are intended for the game application. The next step is to check, what state the game was started in. This happens by calling hgw_context_get_start_command(). The return values indicate the state the game should be going into. This is

necessery, for example, when the game has been paused previously, the state of the game has been saved and the player wishes to continue the current game or restart the on-going game. Again, check the header file for HgwStartCommand for possible start values.

In the game's event loop, the user has to check if events are available by calling hgw_msg_check_incoming(). It will return HGW_ERR_COMMUNICATION if there are messages available, and other values if there are not or if there was error. (see hgw/hgw.h for more details about HgwError enum) Example:

```
/* in game event polling */
HgwMessage *msg;
HgwMessageFlags flags;
if ( hgw_msg_check_incoming(hgw,msg,flags) == HGW_ERR_COMMUNICATION ) {
    /* Message Incoming, process msg */
}
```

Pointer *msg will then hold information about the received dbus message and the game application should then be able to process and act accordingly. Actual information about the received events is stored in msg->e_val which is an enumeration mapped against HgwCallback.

When the game ends or the game is requested to pause or quit, a call to hgw_context_destroy() should be made with initialized hgw as a parameter.

Also, Hildon-games-wrapper library provides an interface to gconf. Gconf should be used to store persistent data like the level of difficulty, sounds on/off etc. This way is preferred, because the data stored in gconf is automatically backed up and restored, when the user so chooses (from desktop, back-up manager). However, this interface is read-only, as writing should be happening in the game-specific plug-in in the osso-games-startup screen.

1.4.3 Creating Game-Specific Plug-in

Each game can create a plug-in for specific settings, such as sound control or difficulty level (see figure 1.2). Basically, each plug-in must implement some pre-defined functions that are executed by the osso-games-startup application.

The functions that can be implemented by each plug-in are:

- static GtkWidget *load_plugin (void) This function creates the game-specific plug-in.
- static void unload_plugin (void) This function destroys global variables, if necessary.
- static void write_config (void) This function saves the game configuration chosen by the player using the plug-in options.

If the game needs a submenu in the osso-games-startup screen main menu (see figure 1.1), the following functions must be used:

- static GtkWidget **load_menu (guint *) This function creates the game-specific submenu that is added to the osso-games-startup main menu.
- static void update_menu (void) This function updates the game-specific menu.

The struct below must be filled and sent to the osso-games-startup application. It specifies which plug-in functions the start-up must call.

```
static StartupPluginInfo plugin_info = {
 GtkWidget *  (* load)         (void);
 void         (* unload)       (void);
 void         (* write_config) (void);
 GtkWidget ** (* load_menu)    (guint *);
 void         (* update_menu)  (void);
 void         (* plugin_cb)    (GtkWidget *menu_item, gpointer cb_data)
     ;
};
```

The last item on the struct is necessary only if the game plug-in requires items such as "save", "save as" or "open" to be a submenu in the default Game submenu.

Each plug-in must contain a reference to the osso-games-startup info. The reference is given when STARTUP_INIT_PLUGIN is called. The following example illustrates a reference to osso-games-startup:

```
static GameStartupInfo gs;
```

The following example illustrates the STARTUP_INIT_PLUGIN that initializes the plug-in. The parameters, in the order they are shown, are:

- Pointer for plug-in information (see the struct shown above)
- GameStartupInfo
- Definition on whether the osso-games-startup should send a D-BUS message on closing
- Definition on whether the osso-games-startup menu has open/save game options

```
STARTUP_INIT_PLUGIN(StartupPluginInfo, GameStartupInfo, gboolean,
    gboolean)
```

In order to inform osso-games-startup that the game has a plug-in, the .conf configuration file of the game must include the PluginPath entry, such as PluginPath=datadir/game01/game01_plugin.so.

Building Example Plug-in for CrazyParking

This section illustrates the plug-in for CrazyParking implementation. The plug-in allows the player to set the initial level of the game, and to define whether the game uses sounds.

Games Start-up screen with CrazyParking plug-in's level and sound configuration:

The following code examples illustrate how the Games Start-up screen works, but the best way to learn to use it is to tinker with the game itself. The source code can be used as wished: as a basic skeleton for the game, or simply to gain a better understanding of the game start-up.

Since the plug-in and osso-games-startup are written with GTK+-2.0, they must include startup_plugin.h from osso-games-startup. In addition, Gconf is used to save the user settings.

```
#include <stdio.h>
#include <gtk/gtk.h>
#include <startup_plugin.h>
#include <gconf/gconf.h>
#include <gconf/gconf-client.h>

#define MENU_SOUND 15
```

Listing 1.6: crazyparking/src/plugin/plugin.c

The following example illustrates the labels for retrieving information at GConf:

```
#define SETTINGS_LEVEL "/apps/osso/crazyparking/level"
#define SETTINGS_SOUND "/apps/osso/crazyparking/sound"
```

Listing 1.7: crazyparking/src/plugin/plugin.c

The following example illustrates the functions that are implemented:

```
static GtkWidget  *load_plugin          (void);
static void        unload_plugin        (void);
static void        write_config         (void);
static GtkWidget **load_menu            (guint *);
static void        update_menu          (void);
static void        plugin_callback      (GtkWidget *menu_item, gpointer
     data);
static void        settings_callback    (GtkWidget *widget, gpointer
    data);
static void        sound_callback       (GtkWidget *widget, gpointer
    data);
```

The following example illustrates some global variables:

```
GConfClient *gcc = NULL;
GtkWidget *board_box;
GtkWidget *level_1_item;
GtkWidget *level_2_item;
GtkWidget *level_3_item;
GtkWidget *level_4_item;
GtkWidget *level_5_item;
GtkWidget *level_6_item;
GtkWidget *level_7_item;
GtkWidget *level_8_item;
GtkWidget *level_9_item;
GtkWidget *level_10_item;
GtkWidget *level_11_item;
GtkWidget *level_12_item;
GtkWidget *settings_item;
GtkWidget *settings_menu;
GtkWidget *difficulty_item;
GtkWidget *difficulty_menu;
static GameStartupInfo gs;
GtkWidget *menu_items[2];
static int changed = FALSE;
GSList * group = NULL;
GtkWidget *sound_check = NULL;
GtkWidget *sound_item;
```

Listing 1.8: crazyparking/src/plugin/plugin.c

The implemented functions of the plug-in must be sent to osso-games-startup: in this case, a GTK_SPIN_BUTTON and a GTK_CHECK_ITEM. If the plug-in has no specific menu, load_menu and update_menu must be NULL.

```
static StartupPluginInfo plugin_info = {
  load_plugin,
  unload_plugin,
  write_config,
  load_menu,
  update_menu,
  NULL
};
```

Listing 1.9: crazyparking/src/plugin/plugin.c

The following example illustrates the initializing plug-in that informs the application that there is a plug-in:

```
STARTUP_INIT_PLUGIN(plugin_info, gs, FALSE, FALSE);
```

Listing 1.10: crazyparking/src/plugin/plugin.c

The following example illustrates the function that initializes the widgets that localize the osso-games-startup standard buttons:

```
static GtkWidget *load_plugin (void)
{
  int board, enable_sound;
  GtkWidget *game_hbox;
  GtkWidget *board_hbox, *board_label;
  GtkWidget *sound_hbox, *sound_label;

  g_type_init();
  gcc = gconf_client_get_default();
```

```
  board = gconf_client_get_int(gcc, SETTINGS_LEVEL, NULL);
  enable_sound = gconf_client_get_int(gcc, SETTINGS_SOUND, NULL);

  game_hbox = gtk_hbox_new (TRUE, 0);
  g_assert (game_hbox);

  board_hbox = gtk_hbox_new (FALSE, 4);

  board_box = gtk_spin_button_new_with_range (1, 12, 1);
  g_assert (board_box);

  gtk_spin_button_set_value (GTK_SPIN_BUTTON (board_box), board);
  g_signal_connect(G_OBJECT(board_box), "value-changed", G_CALLBACK(
      settings_callback), NULL);
  gtk_box_pack_end (GTK_BOX (board_hbox), board_box, FALSE, FALSE, 0);

  board_label = gtk_label_new ("Starting level");
  g_assert(board_label);

  gtk_box_pack_end (GTK_BOX (board_hbox), board_label, FALSE, FALSE, 0)
      ;

  gtk_box_pack_start (GTK_BOX (game_hbox), board_hbox, FALSE, FALSE, 2)
      ;

  sound_hbox = gtk_hbox_new (FALSE, 4);

  sound_check = gtk_check_button_new();
  g_assert (sound_check);

  gtk_toggle_button_set_active (GTK_TOGGLE_BUTTON(sound_check),
      enable_sound);
  g_signal_connect (G_OBJECT(sound_check), "clicked",
                    G_CALLBACK(sound_callback), NULL);

  gtk_box_pack_end (GTK_BOX (sound_hbox), sound_check, FALSE, FALSE, 0)
      ;

  sound_label = gtk_label_new ("Sound");
  g_assert (sound_label);

  gtk_box_pack_end (GTK_BOX (sound_hbox), sound_label, TRUE, TRUE, 0);

  gtk_box_pack_start (GTK_BOX (game_hbox), sound_hbox, FALSE, FALSE, 2)
      ;

  printf ("%s : %s : %d\n", __FILE__, __FUNCTION__, __LINE__);

  return game_hbox;
}
```

Listing 1.11: crazyparking/src/plugin/plugin.c

The following example illustrates the function that is responsible for using Gconf to store the user preferences (as is recommended, since the back-up application stores the GConf database):

```
static void write_config (void)
{
  int value;

  value = gtk_spin_button_get_value_as_int (GTK_SPIN_BUTTON(board_box))
      ;
```

```
  if (value < 1) value = 1;
  else if (value > 12) value = 12;
  gconf_client_set_int(gcc, SETTINGS_LEVEL, value, NULL);

  gconf_client_set_int(gcc, SETTINGS_SOUND,
        gtk_toggle_button_get_active(GTK_TOGGLE_BUTTON(sound_check)),
            NULL);
}
```

Listing 1.12: crazyparking/src/plugin/plugin.c

The following example illustrates the function that initializes the game-specific plug-in menu, which is between the Game and Close submenus of osso-games-startup:

```
static GtkWidget **load_menu (guint *nitems)
{
  int enable_sound;
  //number of entries in maemo-games-startup main menu for this game
  *nitems = 1;
  settings_item = gtk_menu_item_new_with_label ("Settings");
  settings_menu = gtk_menu_new ();
  menu_items[0] = settings_item;
  gtk_menu_item_set_submenu (GTK_MENU_ITEM (settings_item),
      settings_menu);
  //difficulty settings
  difficulty_menu = gtk_menu_new ();
  difficulty_item = gtk_menu_item_new_with_label ("Difficulty");
  gtk_menu_item_set_submenu (GTK_MENU_ITEM (difficulty_item),
      difficulty_menu);
  gtk_menu_append (GTK_MENU (settings_menu), difficulty_item);
  level_1_item = gtk_radio_menu_item_new_with_label (group, "Level 1");
  gtk_menu_append (GTK_MENU (difficulty_menu), level_1_item);
  group = gtk_radio_menu_item_group(GTK_RADIO_MENU_ITEM(level_1_item));
  level_2_item = gtk_radio_menu_item_new_with_label (group, "Level 2");
  gtk_menu_append (GTK_MENU (difficulty_menu), level_2_item);
  group = gtk_radio_menu_item_group(GTK_RADIO_MENU_ITEM(level_2_item));
  level_3_item = gtk_radio_menu_item_new_with_label (group, "Level 3");
  gtk_menu_append (GTK_MENU (difficulty_menu), level_3_item);
  group = gtk_radio_menu_item_group(GTK_RADIO_MENU_ITEM(level_3_item));
  /* ... Listing cut for brevity ...*/
  level_12_item = gtk_radio_menu_item_new_with_label (group, "Level 12"
      );
  gtk_menu_append (GTK_MENU (difficulty_menu), level_12_item);
  group = gtk_radio_menu_item_group(GTK_RADIO_MENU_ITEM(level_12_item))
      ;
  g_signal_connect (G_OBJECT (level_1_item), "toggled",
                    G_CALLBACK (plugin_callback), (gpointer) LEVEL_1);

  g_signal_connect (G_OBJECT (level_2_item), "toggled",
                    G_CALLBACK (plugin_callback), (gpointer) LEVEL_2);

  g_signal_connect (G_OBJECT (level_3_item), "toggled",
                    G_CALLBACK (plugin_callback), (gpointer) LEVEL_3);
  /* ... Listing cut for brevity ...*/

  g_signal_connect (G_OBJECT (level_12_item), "toggled",
                    G_CALLBACK (plugin_callback), (gpointer) LEVEL_12);

  gtk_menu_append (GTK_MENU (settings_menu), gtk_menu_item_new());

  //sound settings
```

```
  sound_item = gtk_check_menu_item_new_with_label("Sound");
  gtk_menu_append (GTK_MENU (settings_menu), sound_item);
  g_signal_connect (G_OBJECT (sound_item), "toggled",
                    G_CALLBACK (plugin_callback), (gpointer) MENU_SOUND
                        );
  gtk_check_menu_item_set_state (GTK_CHECK_MENU_ITEM(sound_item),
  gtk_toggle_button_get_active (GTK_TOGGLE_BUTTON (sound_check)));

  return menu_items;
}
```

Listing 1.13: crazyparking/src/plugin/plugin.c

The following example illustrates the function that is called when any configuration is performed in the menu. This function updates the GTK_SPIN_BUTTON (level change) or the GTK_CHECK_MENU_ITEM (sound change).

```
static void plugin_callback (GtkWidget *menu_item, gpointer  data)
{
    if (MENU_SOUND == (int) data && !changed){
        changed = TRUE;
        gtk_toggle_button_set_active (GTK_TOGGLE_BUTTON (sound_check),
                                      gtk_check_menu_item_get_active (
                                      GTK_CHECK_MENU_ITEM(sound_item)))
                                          ;
    } else if (!changed) {
        changed = TRUE;
        gtk_spin_button_set_value (GTK_SPIN_BUTTON (board_box), (int)
            data);
    }
    changed = FALSE;
}
```

Listing 1.14: crazyparking/src/plugin/plugin.c

The following example illustrates the function that is called to update the menu level option, when the user chooses the level using the GTK_SPIN_BUTTON:

```
static void settings_callback (GtkWidget *widget, gpointer data)
{
  if (!changed) {
    changed = TRUE;
    gint active = gtk_spin_button_get_value_as_int(GTK_SPIN_BUTTON(
        widget));
    if (active == LEVEL_1) {
      gtk_check_menu_item_set_state(GTK_CHECK_MENU_ITEM(level_1_item),
          TRUE);
    }
    else if (active == LEVEL_2) {
      gtk_check_menu_item_set_state(GTK_CHECK_MENU_ITEM(level_2_item),
          TRUE);
    }
    else if (active == LEVEL_3) {
      gtk_check_menu_item_set_state(GTK_CHECK_MENU_ITEM(level_3_item),
          TRUE);
    }
    /*... Listing cut for brevity ...*/
    else if (active == LEVEL_12) {
      gtk_check_menu_item_set_state(GTK_CHECK_MENU_ITEM(level_12_item),
           TRUE);
    }
  }
```

```
  changed = FALSE;
}
```

Listing 1.15: crazyparking/src/plugin/plugin.c

The following example illustrates the function that handles changes to the sound setup:

```
static void sound_callback (GtkWidget *widget, gpointer data)
{
  if (!changed) {
    changed = TRUE;
    gtk_check_menu_item_set_state (GTK_CHECK_MENU_ITEM(sound_item),
                                   gtk_toggle_button_get_active(
                                   GTK_TOGGLE_BUTTON (widget)));
  }
  changed = FALSE;
}
```

Listing 1.16: crazyparking/src/plugin/plugin.c

The following example illustrates the function that is called by osso-games-startup, if necessary.

```
static void update_menu (void)
{
  settings_callback(board_box, NULL);
  sound_callback(sound_check, NULL);
}
```

Listing 1.17: crazyparking/src/plugin/plugin.c

Bibliography

[1] ALSA projects home page. http://www.alsa-project.org/.

[2] Cairo projects home page. http://cairographics.org/.

[3] Esound white paper. http://developer.gnome.org/doc/whitepapers/esd/.

[4] Gstreamer projects home page. http://gstreamer.freedesktop.org/.

[5] GStreamer Application Development Manual. http://gstreamer.freedesktop.org/data/doc/gstreamer/head/manual/html/index.html.

[6] Gstreamer playbin base plugin. http://gstreamer.freedesktop.org/data/doc/gstreamer/head/gst-plugins-base-plugins/html/gst-plugins-base-plugins-playbin.html.

[7] GStreamer Plugin Writers Guide. http://gstreamer.freedesktop.org/data/doc/gstreamer/head/pwg/html/index.html.

[8] GStreamer Core Reference Manual. http://gstreamer.freedesktop.org/data/doc/gstreamer/head/gstreamer/html/.

[9] Gtk+ projects home page. http://www.gtk.org/.

[10] Video for Linux Two. http://www.thedirks.org/v4l2/.