The EFL Manual

Draft 0.1 - 17th December 2010

Notes

Existing documentation cannibalised and reworked to produce this version

Red text in document is draft text not yet worked on.

This document is intended to be paired withto others to create a complete ? guide.

- **The Edje User Manual** (to be written)
- Source code API docs (doxygen) [with overviews removed]

Edited:
Kim Lester

Authors:
various (to be attributed)
Kim Lester

What is EFL?

EFL is a collection of libraries that are independent or may build on top of each-other to provide useful features that complement an OS's existing environment, rather than wrap and abstract it, trying to be their own environment and OS in its entirety. This means that it expects you to use other system libraries and API's in conjunction with EFL libraries, to provide a whole working application or library, simply using EFL as a set of convenient pre-made libraries to accomplish a whole host of complex or painful tasks for you.
One thing that has been important to EFL is efficiency. That is in both speed and size. The core EFL libraries even with Elementary are about half the size of the equivalent "small stack" of GTK+ that things like GNOME use. It is in the realm of one quarter the size of Qt. Of course these are numbers that can be argued over as to what constitutes and equivalent measurement. EFL is low on actual memory usage at runtime with memory footprints a fraction the size of those in the GTK+ and Qt worlds. In addition EFL is fast. For what it does. Some libraries claim to be very fast - but then they also don't "do much". It's easy to be fast when you don't tackle the more complex rendering problems involving alpha blending, interpolated scaling and transforms with dithering etc. EFL tackles these, and more.]

EFL consists of multiple C libraries with bindings to several languages. The libraries are primarily intended to be compiled in a POSIX environment (??) and are supported primarily on GNU/Linux although Windows and other embedded ports are under development.

The library functions may be roughly categorised into the following areas:

- general utilities including communications, image management, configuration
- low level graphics
- widgets
- themes

Why EFL ?

There are already a number of graphics and widget toolkits around so why EFL?
Most of these toolkits work in a similar manner - the application code around every graphical object responsible for drawing and redrawing that object every time a window redraw event occurs. The window itself knows nothing about its drawn contents. In addition the use of 3D window effects, transparency, textures etc needs to be managed by a GPU.

Whilst this paradigm works quite well it does have limitations. For example:

- Drawing and special effects are an increasingly heavy burden on the CPU/GPU. Whilst graphics hardware manufacturers may rub their hands in glee at the horsepower required to get a modern window manager out of bed, embedded systems (which are actually a bigger market than desktops) are left out in the cold.
- There is no real separation between graphics functionality and graphics design. Developers are not generally good at graphics design and graphics designers are not generally good at coding. When one's graphic design and gui functionality are embedded in same code it is usually the graphic design that suffers.
- Themes are traditionally limited to a handful of window decoration colour schemes and a desktop background and despite the recent popularity of compositing window managers, transparent GUI components are relatively rare in the standard toolkits.
- GUI animation is genrally not practical unless embedded into all the widgets at development time which generally does not occur.

To overcome these limitations one typically has to fall back to lower level libraries (OpenGL, Xlib, device hardware etc).

EFL has addressed the above limitations (and more). It provides an envrionment with the following key features:

- high power-to-weight ratio - stunning graphics effects without a GPU.
- supports 2.5D graphics (2D objects, object layering, 2D matrix transforms equivalent to 2D objects oriented in 3D space projected back onto the 2D screen).
- cross platform

- supports embedded processors like ARM cores.
- GUI library
- multimedia playback libraries
- cross platform application support utilities
- theme engine

For the desktop user, EFL's benefit is its superbly flexible thememing engine, allowing a desktop environment or application's appearance to be radically altered by graphics designers without traditional coding.

For the embedded developer, EFL's benefit is its ability to deliver high quality, responsive graphical interfaces on moderate to low end hardware in a small resource footprint and without a GPU.

To understand more of the details see the section Understanding EVAS.

A Little History

EFL libraries grew out of the Enlightenment Window Manager developed in the 1990s. After version 16 (1999) it was decided to rebuild the system from the ground up. Ten years of spare-time later the EFL libraries are maturing to the point where they are now ready to be used by the world at large.
A new window manager (or more accurately "desktop shell") is under development (E version 17) and the embedded system side of things has taken off with both Samsung and Canonical (Ubuntu) using the EFL libraries in various products.

Obtaining EFL

EFL can be found at **www.enlightenment.org**

- Getting binary packages for your Linux distribution.
- Downloading source snapshots of libraries. (URL ????)
- Downloading directly from SVN. (URL ????)
- Running/installing the E-live CD (URL ?????)

Using EFL

There are three ways to approach EFL:

- as a high level application developer - wanting a flexible GUI environment
- as am embedded platform developer - wanting to use EFL in an embedded system
- as a graphic artist - wanting to create cool themes.

xxxxxx

EFL Concepts

EFL Concepts - Event Model

EFL brings a few different paradigms to the table which provide major benefits if understood and used correctly. One of these major features is the rendering and canvas model, but to understand it we first need to discuss EFL's general event and work-flow model.

Why is multi-tasking required within a GUI application?

Traditional console-based programs are straightforward to program when it comes to parallelism. The application is in control and the user normally waits for execution to finish. Interaction happens either before the applicaion starts (command line arguments) or at well defined points during runtime where the application stops and asks the user for required input.

In GUI applications the control situation is reversed. The user is now in control and the application must respond to user actions (which come in the form of events). Since a GUI application should keep its display windows updated and respond to button presses whilst it is processing user requests, a form of parallelism is needed. The GUI application must appear to do several things at once - there is nothing more irritating than a GUI application whose interface freezes whilst it performs some calculation.

Thread Driven vs Event Driven Programming

Thread support is common in many graphics toolkit support libraries to provide multi-tasking. One thread is manages the GUI whilst one or more are processing in the background. The thread model is well understood however thread programming is difficult in real-world cases. When threads have to share common resources the programmer must be careful to avoid race conditions and ensure the use of mutexes/locking to avoid data corruption whilst also avoiding deadlocks. An imprefectly written threaded application can have serious problems ranging from poor performance to deadlock. Threads need also special libraries for implementation and special debugging support from the run-time system.

An alternative approach is event driven programming. Here all pending actions (jobs to do) are queued in a single list. The running application looks at this queue, selects one action to process, then comes back, selects another one and so on. There is a single control flow which simplifies programming (no locks/racing conditions). The UNIX system calls poll() and select() which act on file descriptors are actually examples of event handling mechanisms. Event driven programming is not a panacea, its drawbacks can include sub-optimal performance if an individual action in the queue takes too long to process or if the waiting event queue is too large. The following table compares the two different approaches.

Table 5.2. Threads vs Event based programming

	Threads	Events
Programming effort	High	Low
Locks needed	Yes	No
Possible Deadlocks	Yes	No
Debugging	Hard/Impossible	Easy/Trivial
Good for	Many independent workloads	A few intermixed workloads
Bad for	Many shared resources	Many/heavy workloads
Ideal in	Multi processor/core systems	Single processor/core systems

Actually a third technology - message passing is probably the most powerful multi-tasking technology. It has the parallelability of threads with the simplicity of events.

Threads are however a powerful mechanism for taking advantage of true parallel processing ability in a multiple or multi-core CPU system. In a single-CPU/core system threads can help spread the available CPU cycles between competing tasks but true parallelism is not available.

If you still feel that threads are essential for a graphics application it might interest you to know that the X-Server is a *single* user-level process (no ties with the kernel OS) and successfully handles all on screen drawing using events. If the whole X architecture implementation is possible with events then certainly your application can refrain from using threads too.

EFL (via the Ecore library) chooses the Event-driven approach. The Ecore event loop already deals with many events so no additional effort is required to handle common window actions such as moving/resizing/exposing/repainting. A simple application may not need to deal with events at all. Just draw items in the EFL canvas and when the interface changes (values updated, UI elements added/removed) the application will manage the updating by itself. EFL also provides sophisticated addition event loop features allowing complex application functionality to be be developed without resorting to threads.

Having said all that, there are times when threads are useful (or more accurately, a necessary evil). Ecore does provide wrappers for threads, but hopefully it will be seen that threads are not required within the GUI component of an application and should be reserved for special cases where significant processing needs to be offloaded from the main application, particularly in multi-core environments.

EFL Event Loop

wiki EFLOVERVIEW IMAGES

EFL uses the same mainloop concept as GTK+ and many other toolkits. First an application performs some initialisation and then enters an "infinite loop" processing events until the application quits. In some toolkits (not EFL) the main loop typically runs as fast as it can (processing keyboard events, updating state and rendering graphics), however this results in high CPU usage often performing un-necessary checking or rendering. The EFL mainloop is smart, and sleeps, not consuming CPU resources unless an event occurs, except in rare cases where the developer has specifically created a CPU eating Idler callback.

EFL's event loop management library (Ecore) provides many mechanisms for flexible manipulation of the main loop including functionality not found in most other event loop systems. These capabilities include:

- **tasks can be added to the application's event loop** - these will be actioned on a best effort basis, the exact timing cannot be predicted. (Mechanisms such as Timers exist for more precise control.)
- **an idle callback can be specified** - to be exectuted when the application is otherwise waiting for for I/O or user input.
- **callbacks exist for transition events *entry* into and *exit* from idle state**.

The application is deemed in the *active* state by EFL when executing any event callbacks other than idlers. At some point the queue of outstanding events will have been processed and the application enters the idle state again. Whenever the state changes (edge-triggered), the appropriate idler transition callback (`Enterer` or `Exiter`) will be executed. Once any entry idler has completed its task the event loop will sleep until an event occurs.

On entering Idle a render call, evas_render(), is executed on each Evas canvas. It is this call which updates the GUI. A benefit of this approach is there is only one render per state change (ie per stream of events). this ensures that from a gui perspective multiple changes are performed atomically and only need to be rendered once when all the immediate changes are complete. This also helps if for some reason the event loop is delayed, many pending events will be updated in a single "catch-up" redraw.

Note: Since Idler callbacks do not themselves affect the application's state (a logical necessity), any Idler that needs to wake-up the event loop into the active state needs to queue something that would ordinarily wake up the main loop, like a Job, Timer, etc.

EFL Events

Event handlers, timers and animators etc, are typically responsible for moving program logic along by calling functions that modify program state, update the GUI state, work with files, network IO and so on.

What sort of events does EFL support:

- time based (timers and animators)
- file descriptor events
- unix signals

Time-based events are scheduled by Ecore on a "best effort" basis, using the system sleep mechanism (select() with a timeout, epoll() etc.) to suspend process activity until an event (on a file descriptor) or a timeout occurs. Once Ecore wakes up, it will find out what events are ready, eg any Timers or Animators that expired, Fd handlers to read/write etc.

EFL is restricted by any system imposed time granularity, and if several events happen to be scheduled simultaneously, they will be executed one after the other so small additional delays will occur. In general the response is "good enough" for all but the most time critical of activities.

Ecore also provides a very useful feature in that it serialises UNIX signal handlers into the mainloop event queue, so signals like SIGCHLD are presented to the application as normal events.

Queued events are processed one at a time. Each event has a unique type ID. Some event types may have multiple event handlers (chains). Each handler will be called for a given event of that type unless one of the handlers steals the event (by returning FALSE to indicate no further handlers are to be called on that event). Event handler chains allow multiple subsystems to listen in on events and marshal the information off into their respective subsystems to be handled as desired.

The main loop function (ecore_mainloop_begin() or elm_run() if using Elementary) sits in a loop checking for events, handling timers, callbacks etc. The loop function will exit if it has been requested to exit (ecore_main_loop_quit() or elm_exit()).

EFL Concepts - Drawing with Evas

Libraries: Evas, Ecore

Evas is a high quality raster-based drawing canvas library with support for alpha-blending, resampling and anti-aliased text. Evas provides an abstracted interface to a number of different backend graphics subsystems including OpenGL, X11 and software framebuffers. The same Evas drawing commands produce identical output on all supported backends making it suitable for cross-platform use.

Unlike other canvas objects Evas maintains state; it remembers all the *objects* drawn (rather like a display list) and keeps track of which objects have been updated and which need rendering. This is termed *retained mode* drawing, as opposed to the stateless *immediate mode* drawing done by most other graphics library canvases.

A developer using Evas need not be concerned with window expose events, object repainting etc. For example if you tell Evas to draw a rectangle, Evas records the rectangle's size, shape, colour, visiblity etc. The rectangle can then be modified (eg moved or resized) by providing Evas with just the changed properties. The application is freed from having to specifically redraw the rectangle.

As Evas keeps track of high-level objects rather than just pixels it is able to optimise the rendering pipeline to minimise redrawing activity, for example completely obscured objects need not be rendered at all. In addition Evas defers as much processing as it can until the canvas is ready to be drawn, in this way multiple load font/image commands may be collapsed into single I/O actions. These actions save processor effort and avoids the need for complex application level performance optimisations. Evas's technology is the key to running sophisticated looking graphics on embedded hardware that has no graphics acceleration. But even on accelerated hardware Evas can improve performance significantly, providing more headroom for other processing.

Evas handles a range of image formats including gif, jpeg, png, tiff and xpm. Evas can also output rendered images into jpeg, png and tiff formats.

Evas, like other EFL components can be compiled with certain features omitted as required for platform specific or embedded use.

What Evas is not?

Having said what Evas does do it is important to understand what Evas does not do.

Evas is not:

- a widget set or widget toolkit, however it is their base. Elementary is the EFL widget toolkit.
- dependent on, or aware of main loops, input or output systems. Input should be polled from various sources and fed to Evas.
- able to create windows or report windows updates to your system. It just draws pixels and reports the changed areas
- aware of time and cannot, by itself, animate objects.

Many of the things that Evas doesn't do are provided by other EFL components (for example Ecore).

The Evas Drawing Paradigm

Evas can be seen as a display system that stands somewhere between an immediate mode display system and a widget set. It retains basic display logic, but does little high-level logic like drawing scrollbars, push buttons etc. Having an object aware, stateful canvas requires a significant, but easy, mental shift by developers. Instead of thinking in terms of "I must draw', the mindset becomes "I create and manipulate". This is very different from most lower and mid-level API's that people are used to including Xlib, Cairo, GDI to OpenGL which generally work on the "draw and forget" model.

With Evas the programmer simply creates and manages display objects which are passed to Evas. As a result of this higher level state management, Evas has enough knowledge to handle all the complex optimisations required to handle redraw events with the minimum of computation.

By embracing the Evas drawing paradigm the developers work will be reduced and code will be much simpler with good separaton of program logic and GUI (akin to the Model-View-Controller paradigm). Whilst it is possible to drive Evas in the traditional "I draw now" paradigm, much of the benefit of using Evas will be lost and the effort required will be greater.

The cost of most Evas operations, like evas_object_move(), evas_object_resize(), evas_object_show() etc are approximately zero cost at the time they are executed as they just update coordinates within an object and set a changed flag. The function calls do not render anything themselves.

Setting the file on an image object via evas_object_image_file_set() results in only the image header being loaded to provide image dimensions and alpha channel information. Image loading and decoding is delayed until the image is visible and need to be rendered. If one is sure the image data will be required it can be asynchronously preloaded (ie via a background thread, yes thread!) if it is important to avoid any delays when the image is first rendered.

Immediate Mode Drawing

Immediate mode display systems retain very little, or no state. A program will execute a series of commands, as in the pseudo code:

```
draw line from position (0, 0) to position (100, 200);
draw rectangle from position (10, 30) to position (50, 500);
bitmap_handle = create_bitmap();
scale bitmap_handle to size 100 x 100;
draw image bitmap_handle at position (10, 30);
```

This series of commands is executed by the graphics system and the results are displayed. The graphics system forgets each command as soon as it is executed and the pixels rendered, thus it has little or no idea of how to reproduce the resultant image. As a consequence whenever an expose or other redraw event occurs the application must be asked to redraw sections of the window/screen as needed. Applications typically just send redraw commands for the whole window but region clipping is used to minimise the actual commands sent to the graphics pipeline, nevertheless a lot of needless activity occurs drawing items and then overdrawing them with other items etc to build up the final picture.

The advantage of such a system is that it is simple, and gives a program tight control over how something looks and is drawn. Given the increasing complexity of displays and demands by users to have better looking interfaces, more and more complexity is required in widget set internals, custom drawing code etc. not only to support the additional "bling" but also to minimise unnecessary redraws to keep the application responsive. Programmers not very familiar with graphics programming will often make mistakes at this level and produce code that is sub-optimal. Those familiar with this area generally don't enjoy rewriting the same code each time.

For example, if in the above scene, the windowing system requires the application to redraw the area from 0, 0 to 50, 50 (also referred as "expose event"), then a programmer either sends the entire window drawing commands to the graphics subsystem which then clips them to the relevant region needing redraw or else the programmer must calcualte the updated region and send just the neccessary commands. Either way the end result is:

```
Redraw from position (0, 0) to position (50, 50):

// what was in area (0, 0, 50, 50)?
// 1. intersection part of line (0, 0) to (100, 200)?
      draw line from position (0, 0) to position (25, 50);
// 2. intersection part of rectangle (10, 30) to (50, 500)?
      draw rectangle from position (10, 30) to position (50, 50)
// 3. intersection part of image at (10, 30), size 100 x 100?
      bitmap_subimage = subregion from position (0, 0) to position (40, 20)
      draw image bitmap_subimage at position (10, 30);
```

The alert reader might have noticed that, if all elements in the above scene are opaque, then the system is doing useless paints: part of the line is behind the rectangle, and part of the rectangle is behind the image. These useless paints tends to be very costly, as pixels tend to be 4 bytes in size, thus an overlapping region of 100 x 100 pixels is around 40,000 useless writes. The developer could write code to calculate the overlapping areas and avoid painting then, but then it needs to be integrated with the "expose event" handling and the problem rapidly becomes complex and error prone.

An example of Evas pseudo code is:

```
line_handle = create_line();
set line_handle from position (0, 0) to
position (100, 200);
show line_handle;

rectangle_handle = create_rectangle();
move rectangle_handle to position (10, 30);
resize rectangle_handle to size 40 x 470;
show rectangle_handle;

bitmap_handle = create_bitmap();
scale bitmap_handle to size 100 x 100;
move bitmap_handle to position (10, 30);
show bitmap_handle;

render scene;
```

Whilst the initial setup code above is slightly longer, when the display needs to be changed, the programmer only moves, resizes, shows, hides the objects that need to change. When the display needs to be redrawn (eg from an expose event) the programmer does nothing. Thus the programmer thinks at the more natural object management level, and the canvas software does the rest of the work, figuring out what actually changed in the canvas since it was last drawn, how to most efficiently redraw the canvas and its contents to reflect the current state.

Evas's cross platform portability provides a significant added bonus in hiding platform specific drawing details without sacrificing performance from simple wrapping of immediate mode drawing commands.

Evas Primitives

The core Evas datatype is Evas_Object. This is a C opaque type which represents a generic visual element evas handles and draws. Evas_Objects have a type associated with them, so that specialization is possible. Evas has a set of built-in object types:

- rectangle
- line
- polygon
- text (and multi-line text)
- gradient (two types)
- image

Rectangles, text and images are the most used objects, allowing one to create 2D interfaces of arbitrary complexity with ease.

The above set may be extended to include new objects of arbitrary complexity using *Smart Objects*. Once the developer has taught Evas how to draw a new type of object and provided the standard object functions (add, delete, move, resize, show etc) the object may be used as like any other object, and appear to the application developer as standard Evas_Objects.

Smart Objects support drawing object hierarchies. They can have other objects as children and operations (eg resize) on the parent object can be modified to act differently on the children if needed.

Widget developers can benefit from Smart Objects because it allows them to present widgets in a unified manner as first class drawing objects (along with lines and text etc).
Application developers benefit from a unified drawing interface that includes primitives and widgets. In addition Edje based Smart Objects are fully themable (see Edje later on) without special coding.

Evas comes with three built-in smart object:

- **Box Objects -** A box container is intended to have child elements, displaying them in sequential order. The API provides an extensible mechanism for child element layout algorithms.

- **Table Objects -** A table container lays out its children in a table (or grid). Several predefined layout algorithms are provided (no customisation is available).
- **Edje Objects** This is how Edje and Evas integrate internally. Visual elements whose appearance and behavior are controlled by Edje are just Evas smart objects. Edje encapsulates all the instantiation steps of these objects. An Edje Object must have an EDJ file associated with it. Edje Objects are also containers and may have any Evas objects as children, this is a powerful aggregation mechanism.

Elementary's widgets, as one might expect, are also evas smart objects.

Any Evas_Object may have *hint fields* set. Their objective is to guide the e-libs' functionality in some way during object rendering and layout (similar concept to font hinting). Hints may affect, for example:

- object size (both horizontal and vertically)
- object alignment inside a container
- object padding

Hints are not always guaranteed to be honoured (they are after all just hints). An object's actual size is a property separate from any size hints. Size hints are never used in calculations involving and object's size.

Smart Objects may implement any custom behavior for the values they get from hint fields. For example, Box Objects use alignment hints to align its lines/columns inside its container, padding hints to set the padding between each individual child, etc. Edje also uses size hints to layout its objects. We'll see later on that edje exposes some of these hint fields in the EDC language.

Removed from EO description too detailed: *This operation also requires that a group2, declared inside that EDJ, is associated with this object. This group tells about all of the child elements this edje object has. These can be any evas object, including the smart ones (note the recursive aggregation possibilities, here).*

Understanding the Evas Canvas

(concepts.pdf) 8 pages incl. images

Compiling with Evas (part of Auto docs)

EFL Concepts - Themes

Libraries: Edje, Evas

Edje provides EFL with a graphical design and layout library. The Edje Layout Engine is a significant departure from previous layout engines by providing an abstraction layer separating application code from GUI look and feel (*theme*). An application is thus split into two parts; a graphical part which knows nothing about C code and applicaton code which knows nothing about the GUI layout. A contract is established between the program's backend and its interface (via *signal* definitions, explained below). Once this is done developers can easily change the back-end's functionality independently of the GUI and vice-versa for interface designers.

Edje is not a widget set, but it supplies key capabilities in designing widgets (Refer to Elementary for an EFL-based widget set), supporting the creation and layout of visual elements such as window borders, buttons etc and the ability to animate visual elements. Edje consists of two parts, a development-time compiler and a run-time engine.

Edje supports extremely flexible dynamic layouts and animations which are created in the form of an Edje Data Collection (EDC), which is a text configuraton file detailing the interface look, feel and behaviour. This concept is termed a *declarative user interface*.

The Edje compiler, `edje_cc`, is used to translate these text descriptions (`.edc` files) into low-level layout instructions, and bundles these instructions along with all the required images and fonts into a single binary *theme file* (`.edj`) that is distributed along with the application executable. Edge suppports multiple layout collections in one file, sharing the same image and font database and thus allowing a whole theme to be conveniently packaged into single file.

At runtime the compiled files are parsed and the GUI is built. New widgets may also be designed and coded as needed and are able to seamlessly and dynamically inherit the look and feel of a theme in the same was as existing widgets.

Edje implements a geometry state machine that contains a state graph of object visibility, location, size, colour etc. When declaring an interface visual element, the designer describes one or more states that element can be at. These states can differ in many parameters, for example:

- object's position and size
- object's color and opacity
- object's visibility
- object's response to input events, etc.

In addition to look and feel, Edje allows specified actions and state transitions to be triggered by events either coming from the application or as a result of input to a visual element including mouse-overs, clicks etc. Complex interactions can be built up for example clicking a button can invoke actions that might change the visibility of other visual elements. Actions can be quite sophisticated and indeed have their own simple programming language (refer to Embryo).

State transitions typically involve some visual animation. Predefined transition timeline functions exist for more complex transitions that take an object from current state to target state over a period of time. Transition functions include: linear, accelerated and sinusoidal.

This powerful event driven themable interface can be used to produce almost any look and feel for the basic visual elements. At some point in complexity however however the developer must decide when sufficiently complex GUI element would be better off developed as a C-based widget using Edge for the themeing.

Binary `.edj` files may be vieed and edited usind the following applications:

- Editje (editor)
- Edje_viewer (viewer)

Edje Scripting

Powerful and complex themes may be built with just the basic Edje file syntax. The ability to include simple logic within a theme file to control actions is however a powerful concept. To this end Edje supports the optional use of Embryo sctipting language within Edje configuration files, adding an extra dimension to themeability.

Refer to the Embryo section for more details.

Edj files contain an image and font database, used by all the parts in all the collections to source graphical data. It has a directory of logical part names pointing to the part collection entry ID in the file (thus allowing for multiple logical names to point to the same part collection, allowing for the sharing of data between display elements).

The application using Edje will then create an object in its Evas canvas and set the bundle file to use, specifying the **group** name to use. Edje will load such information and create all the required children objects with the specified properties as defined in each **part** of the given group.

Annotated example:

c file and edj file examples

EFL Concepts - Graphics Callbacks

Libraries: Evas, Ecore

EFL (Ecore-Evas in particular) handles the marshaling of many lower level raw events like mouse moves, presses, etc. and feeds the appropriate events to the Evas canvase(s). Evas then figures out what objects they belong to and will automatically call the appropriate callbacks configured on those objects.

In general at the Evas level the developer just creates objects, sets up callbacks and waits for the callbacks to executed. Callbacks react to all manner of logic transitions and user input, adjusting program state more by creating/destroying/modifying other objects.

xxxxx

EFL Concepts - Widgets

Libraries: Elementary, Edje, (Evas, Ecore)

Elementary is an easy to use basic widget set library built on top of other EFL components. It was originally designed with mobile touch-screen devices in mind, but scales well to the desktop envionment, although it does not yet have a full complement of common widgets.

User interface componets (eg widgets) may be created at any level, from Evas through to Edje. Elementary however provides a set of useful widgets such as:

- icons
- buttons
- scroll bars
- labels, etc.

Being built on top of Edje, the Elementary widgets have the benefit of being fully themable.

Elementary also provides higher level abstractions and control. For example, the library enables operations which affect the whole application's window, with functions to lower it, raise it, maximize it, rotate it and the like. Also, with elementary the user may set the "finger size" (finger-clickable widgets' sizes) intended for use on a touchscreen application, affecting all the widgets with finger interaction.

Common Application Functionality using EFL

Basic Drawing

Using Evas (and higher level) but not GUI Elements

xxxxx

Basic GUI

Elements ...

Application Configuration

Libraries: Ecore

Any non-trivial application needs manage configuration information. Configuration data may be changed by a user and the new settings should be saved to a file for loading next time the applicaiton is run.

Configuration File Formats

There are three common groups of file formats used in modern software:

- text-based, generally line oriented configuration in key-value format.
- XML
- proprietary binary formats
- a centralised configuration repository (ala Windows Registry).

The traditional way to store configuration in a UNIX system is well known. Each application reads and writes a human readable text file, typically line-oriented key-value pairs.. Some applications have much more complex text configuration files (or whole directories) that use a mini configuraion language (eg some web servers, mail and FTP software).

Text files have the benefit that they can be viewed and altered with any text editor and are cross platform. Text files also have several drawbacks depending on the situation:

- modern "desktop" users are generally not comfortable editing text files and prefer GUI dialogues for configuration parameters.
- There are quite a few subtly different configuration file formats, often with irritatingly minor but critical syntactic differences, eg use of #, % or ! as the comment character. key=value versus key: value etc.
- Text files are not suitable for binary data (if required by an application) although they can refer to external data files.
- Development of one's own format requires careful construction of a text parser and for all but the most trivial cases, use of tools like lex, yacc which can however be obscure to those who haven't studied formal computer grammers.
- updating of application state from a changed configuration file whilst the application is running is handled traditionally by sending a SIGHUP to the application. However the mechanism of implementing updates is left entirely to the developer.

XML is often touted as the solution to all problems including configuration files. While XML has definite uses, being edited by humans is not one of them. Humans have difficulty editing XML without making errors and a single character error often renders an entire file unparseable. In addition computer parsing of such files is non-trivial (although many libraries exist to ease the burden). Compare this to the traditional (generally line-based) UNIX configuration file in which any lines not understood can be skipped and one still has a reasonable chance that the application won't keel over.

Commercial closed-source world often uses propietary binary configuration files whose format is known only to the manufacturer. Application updates often require converters to change the configuration from one format to another and these formats are generally not backwards compatible, preventing easy use of previous software versions.

Centralised respositories are generally obscure and hard to manage, although they do have some benefits. They may be treated as either text based or binary based depending on the native file format.

Ecore uses an architecture independent binary format, whilst this may go against conventional unix wisdom it provides several benefits in the EFL context which benefits from combining multiple binary objects into a single file.
EFL uses the EFL EET library (and file format) to store it's configuation data.
Application configuration is normally stored in the user's home directory in the path:

$HOME/.e/apps/*myapp*/config.eet.

Refer to EET for more information.

Ecore provides functions used to store primitives on disk without specifying any additional details of file location or format.

Table - Ecore configuration primitives

Primitive	**Description**
Integer	A simple number
Float	Floating point number
String	String-based value
Colour	RGB description of a colour
Theme	Definition of a theme
Boolean	Binary (true/false) value

Notice that ones does not need to write any code about how to store and load these values. All the low-level code is handled by Ecore.

Notice the .eet extension. Ecore uses the Eet Storage library for configuration which is the same one used by Edje for themes (and Enlightenment for backgrounds, eapps, themes, splash screens e.t.c). The only thing that matters to you as a programmer is the name of the value that you can use to load and store information. Everything else is abstracted away by Ecore.

So why this approach is better? First of all binary files are (as expected) faster that text files. Secondly you don't need to deal with any text parsing at all. Your application does not contain any low level I/O code at all. It just links to the Ecore_Config library. You need of course to provide some GUI for your users to change the values if that is required.

The fact that all configuration files are actually Eet files gives applications two advantages. Eet is designed so that its files are architecture independent. Despite being binary Eet files can be freely moved around on different systems. Additionally it is trivial to program a command line application which will read/dump the values stored on *any* configuration file. Therefore universality is accomplished in a way too. Expert users who want to bypass the GUI and directly edit in the command line can do so, since their Eet reading application will be compatible with all configuration files that use Ecore for storage. The Eet library is open source so the binary format does not need reverse engineering.

Last thing to notice is that Ecore gives you the capability of Configuration Listeners. These Listeners register to configuration parameters and when they are changed they trigger the callback function you have specified. This means that you don't need special "restart" code. Your application can be informed during run-time on configuration changes and act accordingly.

In summary, Ecore provides a unified way to store configuration values which makes things easier for the programmer. It may feel a bit strange to know that your application stores configuration in binary but you should quickly see the advantages.

EFL Development Advice

- It is normally inadvisable to call evas_render() yourself, if using EFL as this will work against the EFL event model and optimizations.
- Use the right primitive for the right task. Do not use Timers as Animators.
- Use Animators to maintain synchronized and artifact-free rendering as they are all called in-sync and key off a single frame clock in an application.
- Pollers are very rough polling mechanisms and recommended for checking things regularly but without the need for precise timing. Pollers are grouped together on power-of-two boundaries to minimize wakeups and improve power consumption.
- Make maximum use of EFL's Jobs as these allow you to defer work until later in the event queue handling, in case events will cause something expensive in CPU-time to be done multiple times when it can be avoided and done once only per wakeup event (before rendering).
- Remember that EFL considers the system to be a state machine with a forward-moving "infinite" event loop, where everything you need can be inserted into the timeline via the constructs above, and then produce an efficient and clean application or library. It is strongly recommended you work within this flexible model.
- EFL itself is **NOT** threadsafe. The mainloop and all EFL calls are meant to be run from a single thread only. Otherwise the work of moving forward through time with a state machine becomes needlessly complex.

  This does not preclude the use of threads to move work to other CPU cores. Ecore has a mechanism for just this. Ecore_Thread is a way to divide the work between heavy function that will run in another process and some callback that will be from the mainloop. Making thread use easy with the EFL. If you must spread your work across multiple CPU's for performance, then divide and conquer. split it up into many worker threads that do the bulk of the work without EFL and then marshal back results to the mainloop to update the application state (and GUI) accordingly.

EFL Library Details

EFL Library Overview

These are the current main libraries

Library Name	Description
Eina	Data types and general utility functions
Evas	
Ecore	
EET	
Edje	
Embryo	
E_Dbus	
Efreet	
EEze	
Eio	
Ethumb	
Elementary	

It is recommended that the libraries be used as a reference and not studied to learn EFL. Many common needs are met through using a combination of functions from different libraries. Therefore the following sections focus on functional goals not individual libraries. A Summary of library contents is provided at the end.

Evas

Supported Platforms

Platform	Support Level
Linux	Full
BSD	?
Mac	?
Solaris	?
Windows XP	?
Windows CE	?

EINA

Eina is a library that provides a number of efficiently implemented datatypes as well as come convenience functions for dynamic library maangement, error management, type conversion, time accounting and memory pool.

Supported Platforms

Platform	Support Level
Linux	Full
BSD	Full
Mac	Full
Solaris	Full
Windows XP	Full
Windows CE	Full

The data types that are available are (see **Data types.**):

Type	Container ?	Description
Array	Yes	standard array of void* data.
Hash Table	Yes	standard hash of void* data.
Inline List	Yes	list with nodes inlined into user type.
List	Yes	standard list of void* data.
Sparse Matrix	Yes	sparse matrix of void* data.
Red-Black tree	Yes	red-black tree with nodes inlined into user type.
String Buffer	No	mutable string to prepend, insert or append strings to a buffer.
Stringshare	No	saves memory by sharing read-only string references.
Tiler	Special	split, merge and navigates into 2D tiled regions.
Trash	Yes	container of unused but allocated data.

The tools that are available are (see **Tools**):

Name	Description
Benchmark	helper to write benchmarks.
Convert	faster conversion from strings to integers, double, etc.
Counter	measures number of calls and their time.
Error	error identifiers.
File	simple file list and path split.
Lazy allocator	simple lazy allocator.
Log	full-featured logging system.
Magic	provides runtime type checking.
Memory Pool	abstraction for various memory allocators.
Module	lists, loads and share modules using Eina_Module standard.
Rectangle	rectangle structure and standard manipulation methods.
Safety Checks	extra checks that will report unexpected conditions and can be disabled at compile time.
String	a set of functions that manages C strings.

Ecore

Ecore is a glue library providing higher-level interfaces to various EFL functions as well as general usage convenience functions such as configuration, IPC mechanisms, simple networking, timers and other common helper functions including the default main loop for most EFL based GUI applications.

Ecore is similar to Glib from GTK+, however there are differences. Whilst Glib can be used without GTK+, GTK+ cannot be used without Glib. Evas on the other hand is self-contained can be used without Ecore if desired.

Ecore is also able to be used by non-graphical applications (eg command line utilities).

There are a number of modules within Ecore. If some of them are not required (or not supported on a particular platform, eg DBUS) the modules may be omitted at link time. In this way Ecore can be compiled as a general purpose utility library for any non-graphical application.

Supported Platforms

Platform	Support Level
Linux	Full
BSD	?
Mac	?
Solaris	?
Windows XP	?
Windows CE	?

Ecore Library Components

This section is a general list of Ecore capabilities.

Job Handling	Add and remove jobs (that call your functions) in the event loop of your application. The event loop and what ecore does with it are explained in a separate section.
Idle Handlers	You can define what your application does in its idle time. You can even define what it should do *before entering* and *after exiting* an idle state. Again this functionality is explained in a separate section.
Configuration management	Ecore provides a binary configuration file (implemented with an Eet storage library). Callbacks may be invoked if the configuration is changed externally.
Process Spawning	An abstraction over UNIX fork, popen, exec, getpid system calls. Also provides convenience functions to send UNIX signals to spawned processes.
Data structures	A hash table, a linked list and a tree data structure are offered. A doubly linked list is available too. Notice that these implementations do not require Evas to be present (as we said before Evas includes code for some simple data structures too). --> Eina ??
File monitoring	the ability to monitor a file descriptor and have a callback function when there is activity on the file managed by this descriptor (e.g. reading/writing).
Search a file	Ecore allows a user-defined set of paths to be searched for a desired file (rather like the various PATH environment variables in Unix)
Ecore plugins	Loading additional Ecore plugins during run-time (ie dlopen).
Timers	Programmable timers. Complete with interval changing *after* the timer has already started.
Network Connections	An abstraction over BSD sockets. Can also use local UNIX sockets for localhost communication.
Framebuffer Utilities	helper functions for framebuffer devices. Calibration, Backlight/Display and

	double click intervals.
Generic IPC	Inter Process Communication. An abstraction layer over UNIX IPC.
Ecore X	An abstraction layer over X. Display, windows, titles, properties, synchronization/flushing, pixmap, geometry etc
Ecore dbus	Dbus bindings. Dbus in an emerging technology offered as a freedesktop.org standard. It is already used by Gnome and recently adopted by KDE. Its objective is system-wide IPC. ? vs E_Dbus ??

Ecore-evas

Ecore-evas is one of ecore's modules you'll certainly want to use. It is a set of convenience functions around evas, which save you from lots of low level interaction with drawing engines, canvas update and maintenance calls and the like.

All graphic engines evas supports have respective high level functions, found at ecore-evas, to:

- instantiate a canvas, bound to that back-end engine, with a given geometry, and - retrieve the application's window, if the graphic back-end implements windows.

Besides that, ecore-evas gives you generic canvas manipulation functions, regardless of the engine being used. For example, you can:

- set callback functions on canvas events like resize, move, gain/loss of focus, - perform actions on the canvas like move, resize, set the title, set fullscreen mode, etc.

Ecore-evas, finally, integrates the calls of canvas updating (re-calculation of object states and re-renderization) into ecore's main loop, so that the user mustn't deal with it.

Edje

Supported Platforms

Platform	Support Level
Linux	Full
BSD	?
Mac	?
Solaris	?
Windows XP	?
Windows CE	?

Edje History

It's a sequel to "Ebits" which has serviced the needs of Enlightenment development for early version 0.17. The original design parameters under which Ebits came about were a lot more restricted than the resulting use of them, thus Edje was born.

Edje is a more complex layout engine compared to Ebits. It doesn't pretend to do containing and regular layout like a widget set. It still inherits the more simplistic layout ideas behind Ebits, but it now does them a lot more cleanly, allowing for easy expansion, and the ability to cover much more ground than Ebits ever could. For the purposes of Enlightenment 0.17, Edje was conceived to serve all the purposes of creating visual elements (borders of windows, buttons, scrollbars, etc.) and allow the designer the ability to animate, layout and control the look and feel of any program using Edje as its basic GUI constructor.

Unlike Ebits, Edje separates the layout and behavior logic.

(Note **Edje Data Collection reference** in source http://docs.enlightenment.org/auto/edje/edcref.html

An Edje Data Collection, it's a plain text file (normally identified with the .edc extension),consisting of instructions for the Edje Compiler.

The syntax for the edje data collection files follows a simple structure of "blocks { .. }" that can contain "properties: ..", more blocks, or both.

Quick access to block descriptions: ...

EET

Supported Platforms

Platform	Support Level
Linux	Full
BSD	?
Mac	?
Solaris	?
Windows XP	?
Windows CE	?

Eet is a small efficient library designed to store key-vaue data into an optionally compressed binary file and supports fast random access data reads.

.EET files (.eet extension) are suitable for storing data that are written rarely and read frequently. In particular EET is useful if it is desirable to avoid reading the whole file in at once, enabling data items to be accessed if and when needed.

It also can encode and decode data structures in memory, as well as image data for saving to Eet files or sending across the network to other machines, or just writing to arbitrary files on the system. All data is encoded in a platform independent way and can be written and read by any architecture.

Two common use cases for this library are:

- storing configuration data and
- storing themes.

Eet files can be very small and highly compressed, making them very suitable for sending across the internet without explicit archive/compress/decompress steps required.

A simple example using Eet

Here is a simple example on how to use Eet to save a series of strings to a file and load them again. The advantage of using Eet over just fprintf() and fscanf() is that not only can these entries be strings, they need no special parsing to handle delimiter characters or escaping, they can be binary data, image data, data structures containing integers, strings, other data structures, linked lists and much more, without the programmer having to worry about parsing, and best of all, Eet is very fast.

example.

EET File Format

The file format is very simple. Data is stored in network byte order (ie big endian, MSB first, LSB last).

There is a directory block at the start of the file listing entries and offsets into the file where they are stored, their sizes, compression flags etc. followed by all the entry data strung one element after the other.

All Eet files start with a 4 byte magic number.

Table - Eet file format

Offset	Length	Description
0	4	Magic: 0x1ee7ff00
4	4	Number of file entries (signed integer, <0 invalid). 0 is valid. Max value: 0x7fffffff
8	4	Size of directory table (bytes) (signed integer, <0 invalid)
n Directory Entries each entry: 5x4 bytes)		
12+5*n	4	object's file offset from start of file (signed integer, <0 invalid)
16+5*n	4	Flags. Bit 0 set if entry compressed with zlib.
20+5*n	4	size of object as stored in file (bytes) (signed integer, <0 invalid)
24+5*n	4	size of object when loaded (ie uncompressed) (signed integer, <0 invalid) will be same as previous value for uncompressed data.
28+5*n	4	length **S** (bytes) of string identifier (signed integer, <0 invalid) Note: NUL terminator is not counted or stored.
32+5*n	S	String name of entry (no NULL terminator)
block1_off	block1_len	DATA BLOCK 1 (by convention defined by Directory entry 1 but not mandated)
block2_off	block2_len	DATA BLOCK 2
...		
block*n*_off	block*n*_len	DATA BLOCK n

To read an entry from an Eet file, simply find the appropriate entry in the directory table, find its offset and size, and read it into memory. If it is compressed, decompress it using zlib and then use that data.There is no alignment of data, so all data types follow immediately on, one after the other. All compressed data is compressed using the zlib compress2() function, and decompressed using the zlib uncompress() function. Please see zlib documentation for more information as to the encoding of compressed data.

The contents of each entry in an Eet file has no defined format as such. It is an opaque chunk of data, that is up to the application to deocde, unless it is an image, ecoded by Eet, or a data structure encoded by Eet. The data itself for these entries can be encoded and decoded by Eet with extra helper functions in Eet. **eet_data_image_read()** and **eet_data_image_write()** are used to handle reading and writing image data from a known Eet file entry name. **eet_data_read()** and **eet_data_write()** are used to decode and encode program data structures from an Eet file, making the loading and saving of program information stored in data structures a simple 1 function call process.
Please see src/lib/eet_data.c for information on the format of these specially encoded data entries in an Eet file (for now).

EDC files are processed by a binary distributed with edje, the edje compiler, which packs the interface's description and data altogether in the form of an EET file. People have historically named these interface files with the .edj extension. For now on, we'll refer to these specific EET files as EDJ files.
Besides we cite just these two use cases, many more exist for eet. One can store arbitrary data chunks at EET files. There are developers storing javascript scripts in it, for example.

Most themes delivered via the Internet nowadays are rather simple in their structure. Although they appear to be single files, in reality they are a bunch of files compressed using the usual compression methods (e.g. zip or .tar.gz). The application that uses the themes uncompresses them (usually upon installation and not on the fly) and a directory containing all the images of the theme is added to the application directory. Eet takes a completely different approach. The theme file is a single file which contains compressed images and the description of the interface too. This information is read by the Edje layout library to create the interface of your application on the fly. The theme is never uncompressed unless it is modified by a themer (and then compressed again). Moving around themes means moving around files. You can also use Eet as a general storage format for images, text, or even arbitrary data chunks. Eet is best known for storing themes but nothings prevents you from using it for all your storage needs.

Embryo

Embryo is a small library implements a scripting language based on an existing language called Pawn (formerly named "Small"). Pawn and hence Embryo is more-or-less a simplified subset of C (no pointers!).

Supported Platforms

Platform	Support Level
Linux	Full
BSD	?
Mac	?
Solaris	?
Windows XP	?
Windows CE	?

The run-time environment provided by embryo is sandboxed largely protecting the application developer from incorrect or malicious scripts.

Embryo programs are compiled by `embryo_cc`.

Refer to xxxx for an introduction to Pawn. More information about Embryo can be found xxxxx.

Usng Embryo in your own code

Embryo may be used in your own application code. To do so at least the following need to be done:

- Include **Embryo.h**.
- Load the Embryo program using one of the **Program Creation and Destruction Functions**.
- Set up the native calls with **embryo_program_native_call_add**.
- Create a virtual machine with **embryo_program_vm_push**.
- Then run the program with **embryo_program_run**.

Embryo and Edje

Embryo is currently used within Edje and enables simple but powerful action logic to be built into Edje configuration files. Embryo within Edge is limited to acting upon interface elements. Embryo provides some built-in functions and a way of defining small functions inside Edje configuration files.

Besides occurring at group scopes, where message handler functions must be declared, when applicable, script blocks may also occur inside programs. The code snippet which follows illustrates this use.

```
program {
name: "part_name";
signal: "show";
source: "*";
script {
set_int(is_mouse_down, 0);
set_int(enabled, 1);
}
```

set_int is embryo’s way of assigning to a global integer variable, which in this case would have been declared at a group level script block. As you see, embryo facilitates the task of keeping state in the interface..
When a program has such block, it mustn’t have the action one. The functions listed at this script block form the program’s action in this case. Naturally, besides setting (or referencing) global variables, global functions could be used.

Elementary

A simple widget toolkit

Supported Platforms

Platform	Support Level
Linux	Full
BSD	?
Mac	?
Solaris	?
Windows XP	?
Windows CE	?

Available Widgets

Widget	Description
Anchorblock	
Anchorview	
Ctxpopup	
Fileselector	
File Selector Button	
File Selector Entry	
Hoversel	
Scrolled_Entry	
Bg	
Box	
Bubble	
Button	
Calendar	
Check	
Clock	
Conformant	
Diskselector	
Entry	
Flip	
Frame	
Gengrid	
Genlist	
Hover	
Icon	

Image	
Index	
Label	
Layout	
List	
Getting Started	
General	
Selective Widget Scaling	
Styles	
Elementary Config	
Elementary Profile	
Elementary Engine	
Elementary Fonts	
Fingers	
Caches	
Focus	
Scrolling	
Scrollhints	
Widget Tree Navigation	
Debug	
Map	
Mapbuf	
Menu	
notify	
Pager	
Panel	
panes	
Photo	
Photocam	
Progressbar	
Radio	
Scroller	
Separator	
Slider	
slideshow	
Spinner	

Table	
Theme	
Thumb	
Toggle	
Toolbar	
Transit	
Widget	
Win	
Inwin	
Cursors	
Tooltips	
Actionslider	
Animator	
Colorselector	
Flipselector	

E_Dbus

E_Dbus is a wrapper around the Freedesktop.org dbus library, an application message bus system. E_Dbus also implements a set of specifications using dbus as interprocess comunication.

Supported Platforms

Platform	Support Level
Linux	Full
BSD	?
Mac	?
Solaris	?
Windows XP	?
Windows CE	?

Module	Description
EDbus	Wrapper around the dbus library, which implemententent an inter-process communication (IPC) system for software applications to communicate with one another.
EBluez	Implementation of the BlueZ specifications, for wireless communications with Bleutooth devices.
EConnman	Implementation of the connman specifications, which manages internet connections within embedded devices running the Linux operating system.
EHal	Implementation of the HAL specifications, which is a (software) layer between the hardware devices of a computer and the softwares that run on that computer (Hardware Abstraction Layer). HAL is deprecated, in favor of DeviceKit.
ENotify	
EOfono	Implementation of the mobile telephony ofono intercace specifications,
EUkit	Implementation of the DeviceKit specifications, which is, like HAL, an Hardware Abstraction Layer. DeviceKit is a replacement of the deprecated HAL system. It has two submodules: UDisks, which manipulate storage devices, and UPower, which manage power devices.

Efreet

Efreet is a library designed to help applications work with several of the Freedesktop.org standards regarding Icons, Desktop files and Menus.

Supported Platforms

Platform	Support Level
Linux	Full
BSD	?
Mac	?
Solaris	?
Windows XP	?
Windows CE	?

Efreet implements the following specifications:

- XDG Base Directory Specification
- Icon Theme Specification
- Desktop Entry Specification
- Desktop Menu Specification
- FDO URI Specification
- Shared Mime Info Specification
- Trash Specification

Eeze

Eeze is a library for manipulating devices through udev with a simple and fast api.

Supported Platforms

Platform	Support Level
Linux	Full
BSD	?
Mac	?
Solaris	?
Windows XP	?
Windows CE	?

Eeze interfaces directly with libudev, avoiding such middleman daemons as udisks/upower or hal, to immediately gather device information the instant it becomes known to the system. This can be used to determine such things as:

- If a cdrom has a disk inserted
- The temperature of a cpu core
- The remaining power left in a battery
- The current power consumption of various parts
- Monitor in realtime the status of peripheral devices

Each of the above examples can be performed by using only a single eeze function, as one of the primary focuses of the library is to reduce the complexity of managing devices.

Eeze functions
udev UDEV functions
Watch Functions that watch for events
Syspath Functions that accept a device /sys/ path
Find Functions which find types of devices

Eio

Eio is a cross platform library providing asynchronous input/output operation.

Supported Platforms

Platform	Support Level
Linux	Full
BSD	Full
Mac	?
Solaris	Full
Windows XP	Full ?
Windows CE	Full ?

Most operation are done in a separated thread to prevent lock.

See **Eio Reference API**.
Some helper to work on data received in Eio callback are also provided see **Eio Reference helper API**.

Ethumb

Supported Platforms

Platform	Support Level
Linux	Full
BSD	?
Mac	?
Solaris	?
Windows XP	?
Windows CE	?

[auto]

C-S tute:
Ethumb provides both in process and client-server generation methods. The advantage of the client-server method is that current process will not do the heavy operations that may block, stopping animations and other user interactions. Instead the client library will configure a local **Ethumb** instance and mirrors/controls a remote process using DBus. The simple operations like most setters and getters as well as checking for thumbnail existence (**ethumb_client_thumb_exists()**) is done locally, while expensive (**ethumb_client_generate()**) are done on server and then reported back to application when it is finished (both success or failure).