

Efl Concept Overview

[Actions](#) ▼

🕒 Updated 3,344 Days Ago 🌐 Public



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.



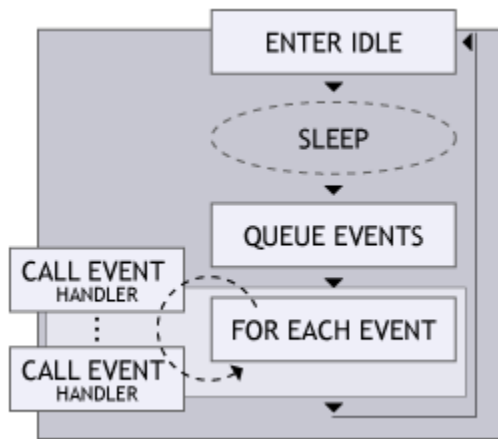
One thing that has been important to EFL is efficiency. That is in both speed and size. The core EFL libraries even with Elementary are about half the size of the equivalent "small stack" of GTK+ that things like GNOME use. It is in the realm of one quarter the size of Qt. Of course these are numbers 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.

The Model

One thing EFL does do is bringing a few new or different paradigms to the table. Some of these mesh well with what is already done, others are vastly different, but bring major benefits, if understood and used correctly. One of these major ones is the rendering and canvas model that EFL has thanks to Evas, Ecore and Edje mostly. In order to understand this, the general event and work-flow model will first have to be introduced. Everything slots in nicely if this model is embraced (and extended appropriately).

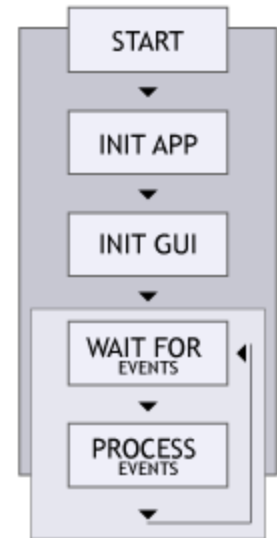
EFL has embraced the same mainloop concept that GTK+ and many other toolkits and software have adopted. It has rolled it wholeheartedly into the system as the core. The idea is that you do some initialization of the application and then enter a mainloop function (in Ecore it's the `ecore_mainloop_begin()` or if you use Elementary it wraps this with a shorter `elm_run()` function). This function will sit in a loop and check for events, handle timers

and other services and call callbacks set up beforehand or during the running of the loop. The loop function will exit if it has been requested to exit (in Ecore `ecore_main_loop_quit()` and in Elementary that wraps Ecore it's `elm_exit()`). For those familiar with traditional game programming, this seems familiar, except you would also have implemented the mainloop too with an infinite while or for loop that fetches input events, updates game world then renders the updated scene, then loops and repeats as quickly as it can.



In the EFL case, the mainloop isn't "dumb" and will not consume CPU resources unless there is work to do. It will sleep and consume no CPU time until some event happens (except in rare circumstances -

when you use Idlers for example that are called in a loop during what normally would be idle time waiting for something to happen). From EFL's point of view all of this is handled in Ecore and it supports many constructs for manipulating this mainloop in a logical and flexible way. It handles animation via animators



where the mainloop handles timing out and scheduling these at regularly spaced intervals in time (on a best-effort basis), as with Timers, Pollers, Idler Enterers, Idle Exiters, Idlers, Jobs, Fd Handlers (File descriptor Handlers) and Event handlers.

In the EFL view, the application, when executing any callbacks other than Idlers, is "active". It will go in and out of this active state via calling Idle Enterer and Exiter callbacks (edge-triggered callbacks). These get called whenever going in and out of the Idle state. Idlers themselves do not as such transition the mainloop from being in an idle state, so any Idler that needs to "wake up" the loop conceptually to become active needs to queue something that would ordinarily wake up the main loop, like a Job, Timer, etc.. This is the only exception due to the conceptual model and the need for efficiency (not entering and exiting idle per idler call).

Otherwise you will have the loop model pretty much start at calling the Idle Enterers, then sleeping until "something happens". That would depend on the system, but wakeup events may be time based (timers and animators) and these are scheduled by Ecore on a "best effort" basis. That means they use the system sleep mechanism (`select()` with a timeout, `epoll()` etc.) to send the CPU to sleep and wait until an event on an input (a file descriptor) wakes up, or until the timeout happens. The system may impose granularity limits on such a sleep, so beware that this is not a guaranteed thing, but in general will be "good enough". Once Ecore wakes up, it will find out why it woke up and handle things appropriately. It will call Timers or Animators that expired, call Fd handlers to read data (or write data) on "ready" Fd's and queue events as a result of this. Ecore also does some things that other mainloops do not, and that is to serialize UNIX system handlers into the mainloop event queue too, so things like SIGCHLD from child processes is handled for you with an event for the child exit placed in your event queue.

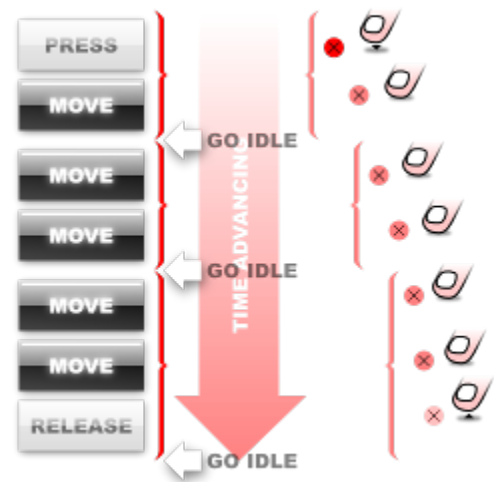
Events

These events in the queue are then processed one at a time. Each event has a unique ID (type) and a certain event type may have multiple event handlers set to handle it. Each handler will be called for that event type (unless one of them steals the event and then returns FALSE to indicate no further handlers are to be called on that event). This way multiple subsystems can listen in on these events and marshal the information off into their respective subsystems to be handled as desired. The event handlers, timers and animators are normally 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. At some point the stream of events in the queue is "handled" and the application is "done" with updating its state. At this point the application enters the idle state again, and here Idle Enterers are called.

Callbacks

One thing EFL (Ecore-Evas in this case) does is handling marshaling of lots of the lower level "raw" events like mouse moves, presses, etc. and "feeds" them to the Evas canvases to which those events belong for you. Evas then figures out what objects they belong to and will call the appropriate callbacks set on those objects for you, so in general at the Evas level you just create objects and set up callbacks for them and wait for these callbacks to be called. These callbacks react to all manner of logic transitions and user input, adjusting your program state more by modifying other objects, creating them or destroying them.

When you enter Idle, Ecore-Evas set up an Idle Enterer for you that calls the Evas render call (`evas_render()`) on each canvas you have, which in turn "magically" appears to update the GUI. This way you only render once per "state change" (per stream of events, as generally a stream of events will change a whole series of things, not just a single object, and often events may be queued and processed together as a single action, saving rendering time, and allowing implicit "playing catch-up" as part of the event model).



Advice

As general advice, it would generally be inadvisable to call `evas_render()` yourself, if using EFL as this will work against the model and optimizations as above. Also using the right primitive for the right task is important. Do not use Timers as Animators. Use Animators as they all will get called in-sync and be able to key off a single frame clock in an application, to maintain synchronized and artifact-free rendering. Pollers are very rough polling mechanisms and highly suggested for anything that needs to regularly "check" something, but does not need an exact time at which to do it, as Pollers get grouped together on power-of-two boundaries to minimize wakeups and improve power consumption. Also 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).

Just 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. If you work with this model, things will work out for the best.

Threads

EFL 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. But 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.

Rendering and Canvas

Now within this model live Evas and Edje. Evas is the canvas engine. It handles the entire "state" of a "window" (canvas) by filling that canvas with objects and manipulating their state. Show them, hide them, move them, resize, set the file sources for images, colors for rectangles, fonts and text content for text and more. When Evas renders it evaluates the state of all objects, if they changed since the last rendering and works out what to render and how to make the updated state of the canvas be reflected in the output. It will try and minimize this update work, since it knows the prior and current state of every object, as well as knowing the entire canvas contents, so it can discard work well in advance as it knows that "later I will cover up this rendering anyway" and thus skip it. It also will defer as much work until render time as it can, as at this point it considers the canvas state as "stable" and needing an update, so you may find Evas loading fonts and images from disk at the time it renders to avoid repeated "create object, set image file, delete the object, create new one" cycles that may happen often during setup or major state changes. By deferring work as late as possible, it can avoid lots of "busy work" in doing useless tasks. This allows for the application using Evas to worry less about such optimizing and more on its logic flow, as these "skip the work" optimizations will often be done for you.

But one major change by having such a canvas and state model for the GUI, is no longer actually rendering anything yourself. The mindset needs to change from a "I draw" to "I create and manipulate" model. This is very different from almost all lower and even mid-level API's that people are used to - everything from Xlib, Cairo, GDI to OpenGL and more. They all work on the "draw and forget" model. Evas is a retained-mode scene graph that you manipulate. Evas will make the cost of most operations (like `evas_object_move()`, `evas_object_resize()`, `evas_object_show()` etc.) be "zero cost" (or well as close to zero as possible). All most of these calls do is to update coordinates within an object and set a changed flag. That's it. They don't draw anything. Even setting the

file on an image object via `evas_object_image_file_set()` will only load the image header so the object can have the original image dimensions and alpha channel flag set. It will not decode the image data, avoiding a lot of file IO and decode overhead, and push this off until later (render time... If the image is visible and the data needed). If you are sure you will need the data though, Evas has the ability to begin an asynchronous preload of the image data in a background thread and inform you when it's ready.

If you mold your programming paradigm around this scene graph and state engine, you will find that you spend less time fighting EFL, and in fact end up with very lean compact and concise code that is almost entirely program and UI logic, which is what your application should be anyway, with all the nasty footwork being done for you. If you want to work with a more traditional model, where you have to take care of rendering yourself and you literally drive the timeline by hand, you will find yourself fighting EFL and spending a lot of code and time working against optimizations and streamlining. Work with the model and your applications will begin to be clean and seamless expressions of high-level logic, with the low-level handled for you by EFL in a clean and efficient way.

Last Author [cedric](#)

Last Edited Mar 15 2013, 8:37 AM

Projects *None*

Subscribers *None*