The EFL Cookbook

Various
Edited by Ben 'technikolor' Rockwood

The EFL Cookbook

by Various and Ben 'technikolor' Rockwood

Stuff.

Table of Contents

1. Introduction 1
2. Imlib2 2
    Recipe: Image Watermarking 2
    Recipe: Image Scaling 4
    Recipe: Free rotation 5
    Recipe: 90 degree Image rotation 6
    Recipe: Image Flipping 7
3. EVAS 9
    Recipe: Using Ecore_Evas to simplify X11 canvas initialization 9
    Recipe: Key Binds, using EVAS Key Events 10
    Recipe: Evas Smart Object Introduction 12
4. Ecore 21
    Recipe: Ecore Config Introduction 21
    Recipe: Ecore Config Listeners 23
    Recipe: Connecting to a server with Ecore_Con 25
    Recipe: Ecore Ipc Introduction 30
    Recipe: Ecore Timers 36
    Recipe: Adding Ecore Events 37
5. EDB & EET 40
    Recipe: Creating EDB files from the shell 40
    Recipe: EDB introduction 41
    Recipe: EDB key retrieval 43
6. Esmart 44
    Recipe: Esmart Trans Introduction 44
    Recipe: Esmart Container Introduction 48
7. Epeg & Epsilon 61
    Recipe: Simple Thumbnailing with Epeg 61
    Recipe: Simple Thumbnailing with Epsilon 62
8. Etox 65
    Recipe: Etox Overview 65
9. Edje 68
    Recipe: A Template for building Edje applications 68
    Recipe: Creating/Triggering Edje Callbacks 69
    Recipe: Working with Edje files 72
        edje_cc 73
        edje_decc 73
        edje_recc 74
        edje_ls 74
        edje 74
10. Edje EDC & Embryo 76
    Recipe: Edje/Embryo toggle 76
    Recipe: Edje text effect fading 82
11. EWL 84
    Recipe: EWL Introduction 84
12. Evoak 94
    Recipe: Evoak hello client 94
13. Emotion 97
    Recipe: Quick DVD player with Emotion 97
    Recipe: Expanded Media player with Emotion 98

List of Examples

2.1. Imlib2 WaterMark Program 3
2.2. Image Scaling 4
2.3. Free rotation 5
2.4. 90 degree Image rotation 6
2.5. Image Flipping 7
3.1. Ecore_Evas Template 9
3.2. Key grabbing using EVAS Events 10
3.3. EVAS Keybind Compile 11
3.4. foo.h 12
3.5. foo.c 13
3.6. main.c 18
3.7. Compilation 19
4.1. Simple Ecore_Config program 21
4.2. Compilation command 22
4.3. Simple config.db script (build_cfg_db.sh) 22
4.4. Ecore_Config listener 23
4.5. Compilation 25
4.6. preamble 26
4.7. gettin' the party started 26
4.8. hook her up 27
4.9. connecting 27
4.10. go speed racer 28
4.11. added 28
4.12. del'd 29
4.13. data 29
4.14. compilation 30
4.15. Ecore_Ipc client: preamble 31
4.16. Ecore_Ipc client: main setup 31
4.17. Ecore_Ipc client: main creating client 31
4.18. Ecore_Ipc client: main end 32
4.19. Ecore_Ipc client: sig_exit_cb 32
4.20. Ecore_Ipc client: the callbacks 32
4.21. Ecore_Ipc server: preamble 33
4.22. Ecore_Ipc server: main setup 33
4.23. Ecore_Ipc server: main creating server 34
4.24. Ecore_Ipc client: main end 34
4.25. Ecore_Ipc server: sig_exit callback 34
4.26. Ecore_Ipc server: the callbacks 35
4.27. Ecore_Ipc: compilation 35
4.28. Ecore Timers 36
4.29. Compilation 37
4.30. Ecore event example 37
4.31. 38
5.1. EDB file shell script 40
5.2. EDB introduction 41
5.3. Compiling 42
5.4. EDB key retrieval 43
5.5. Compiling 43
6.1. Includes and declarations 44
6.2. main 44
6.3. exit and del callbacks 45
6.4. _freshen_trans 45
6.5. resize_cb 46

6.6. move_cb 46
6.7. Setup ecore/ecore_evas 46
6.8. Creating Esmart_Trans object 47
6.9. Simple makefile 47
6.10. Includes and declarations 48
6.11. main 48
6.12. Initialization 49
6.13. Shutdown 50
6.14. Window callbacks 50
6.15. make_gui 51
6.16. Edje Callbacks 52
6.17. container_build 52
6.18. Adding Elements to the Container 53
6.19. _set_text 54
6.20. _left_click_cb 55
6.21. _right_click_cb 55
6.22. _item_selected 56
6.23. The Edc 56
6.24. The Container Part 58
6.25. The Element Group 59
6.26. Makefile 60
7.1. Epeg Simple Thumbnail 61
7.2. 62
7.3. Epsilon Simple Thumbnail 62
7.4. 63
8.1. Etox Overview Example 65
8.2. 66
9.1. Edje Template 68
9.2. Callback program 69
9.3. EDC file 71
9.4. Compilation 72
9.5. edje_cc Usage 73
9.6. edje_decc Usage 73
9.7. edje_recc Usage 74
9.8. edje_ls Usage 74
9.9. edje Usage 75
10.1. Creating the variable 77
10.2. Initializing variables 77
10.3. The toggler button 77
10.4. Hooking into the mouse events 78
10.5. Build script 79
10.6. Edje toggle without Embryo 80
10.7. Fade effect with text 82
10.8. Compliation 83
11.1. Includes and declarations 84
11.2. main 84
11.3. mk_gui: creating the window 85
11.4. The main container 86
11.5. Create the menu bar 86
11.6. Create the scrollpane 87
11.7. Create the text area 88
11.8. Add menu contents 88
11.9. Attach callbacks 89
11.10. Destroy callback 89
11.11. File menu open callback 89
11.12. File dialog destroy callback 90
11.13. File dialog open button callback 90
11.14. File dialog home button callback 91

11.15. Read text file 91
11.16. Key press callback 92
11.17. Compilation 93
12.1. Includes and pre-declarations 94
12.2. main 94
12.3. Canvas info callback 95
12.4. Disconnect callback 95
12.5. setup routine 95
12.6. Compilation 96
13.1. Compiling 97
13.2. DVD player in 55 lines of code 97
13.3. Emotion Media Player 98

Chapter 1. Introduction

Welcome to the enlightened state of programming. This cookbook is a collection of tips, tricks, tutorials and introductions arranged into recipe style in order to help you quickly become proficient with the Enlightenment Foundation Libraries. The Enlightement Foundations Libraries, refered to simply as the EFL, is a collection of libraries origionally written for use by the Enlightenment DR17 Window Manager. However, as these libraries grew and were tested and deployed more and more, general functionality was added bringing us to enjoy a rich and powerful set of libraries that can be used to help solve all sorts of problems and act as a venerable alternative to the currently popular GTK and QT library sets.

The EFL is a comprehensive group of C libraries that can provide a means for almost any graphical need on many platforms. The following is a concise breakdown of libraries which make up the EFL.

EFL Componant List

Imlib2	Complete image manipulation library, including rendering to X11 drawables.
EVAS	Canvas library with multiple backend engines including hardware OpenGL acceleration.
Ecore	Modular library with event handling and timers. It also includes sockets, IPC, both FB and X11 setup, teardown, event handling, job control, configuration handling and more.
EDB	A database library capabile of storing strings, values, and data for simple, consistant and centralized configuration handling.
EET	A flexable container format for storing binary images and data.
Edje	An image abstraction library and toolkit based on EVAS, keeping all aspects of the user interface completely seperate from application code through the use of signal passing.
Embryo	A scripting language typically used with Edje for advanced control.
Etox	A text formatting and manipulation library, complete with text stylization capabilities.
Esmart	A library consisting of various EVAS smart objects for easy reuse, including the popular transpancy hack.
Epeg	A lightning fast thunbnailing library for JPEGs independant of other EFL components.
Epsilon	A flexible and fast thumbnailing library for PNG, XCF, GIF and JPEG.
Evoak	An EVAS canvas server library and toolkit.
EWL	A complete widget library.
Emotion	An Evas object library for video and DVD playback using libxine.

Chapter 2. Imlib2

Imlib2 is the successor to Imlib. It is not just a newer version - it is a completely new library. Imlib2 can be installed alongside Imlib 1.x without any problems since they are effectively different libraries - but they Have very similar functionality.

Imlib2 can do the following:

- Load image files from disk in one of many formats
- Save images to disk in one of many formats
- Render image data onto other images
- Render images to an X-Windows drawable
- Produce pixmaps and pixmap masks of Images
- Apply filters to images
- Rotate images
- Accept RGBA Data for images
- Scale images
- Alpha blend Images on other images or drawables
- Apply color correction and modification tables and factors to images
- Render images onto images with color correction and modification tables
- Render truetype anti-aliased text
- Render truetype anti-aliased text at any angle
- Render anti-aliased lines
- Render rectangles
- Render linear multi-colored gradients
- Cache data intelligently for maximum performance
- Allocate colors automatically
- Allow full control over caching and color allocation
- Provide highly optimized MMX assembly for core routines
- Provide plug-in filter interface
- Provide on-the-fly runtime plug-in image loading and saving interface
- Fastest image compositing, rendering and manipulation library for X

If what you want isn't in the list above somewhere then likely Imlib2 does not do it. If it does, it likely does it faster than any other library you can find (this includes gdk-pixbuf, gdkrgb, etc.) primarily because of highly optimized code and a smart subsystem that does the dirty work for you and picks up the pieces for you so you can be lazy and let Imlib2 do all the optimizations for you.

Imlib2 provides a powerful engine for image manipulation and rendering. Using loaders it can handle a variety of image formats including BMP, GIF (via unGIF), JPEG, PNG, PNM, TGA, TIFF, XPM and more.

Recipe: Image Watermarking

Ben technikolor Rockwood <benr@cuddletech.com>

With so many individuals putting so many images online its easy to forget where they came from and hard to ensure that copyrighted material isn't inadvertently misused. Simply adding a watermark image, such as your sites logo, to each of your images can solve both these problems. But adding watermarks manual is a long and repetitive task. Imlib2 can easily be used to solve this problem. What we need to do is take an input image, and then specify a watermark image (your logo), position the watermark on the input image and then save it out to a new image which we'll use on the site. The app would look something like this:

Example 2.1. Imlib2 WaterMark Program

```
#define X_DISPLAY_MISSING
#include <Imlib2.h>
#include <stdio.h>

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

    Imlib_Image image_input, image_watermark, image_output;
    int     w_input, h_input;
    int     w_watermark, h_watermark;
    char    watermark[] = "watermark.png";

    if(argc > 1)  {
        printf("Input image is: %s\n", argv[1]);
        printf("Watermark is: %s\n", watermark);
    }
    else {
        printf("Usage: %s input_image output_imagename\n", argv[0]);
        exit(1);
    }

    image_input = imlib_load_image(argv[1]);
    if(image_input) {
        imlib_context_set_image(image_input);
        w_input = imlib_image_get_width();
        h_input = imlib_image_get_height();
        printf("Input size is: %d by %d\n", w_input, h_input);
        image_output = imlib_clone_image();
    }

    image_watermark = imlib_load_image(watermark);
    if(image_watermark) {
        imlib_context_set_image(image_watermark);
        w_watermark = imlib_image_get_width();
        h_watermark = imlib_image_get_height();
        printf("WaterMark size is: %d by %d\n",
                w_watermark, h_watermark);
    }

    if(image_output) {
        int dest_x, dest_y;

        dest_x = w_input - w_watermark;
        dest_y = h_input - h_watermark;
        imlib_context_set_image(image_output);

        imlib_blend_image_onto_image(image_watermark, 0,
                0, 0, w_watermark, h_watermark,
                dest_x, dest_y, w_watermark, h_watermark);
        imlib_save_image(argv[2]);
        printf("Wrote watermarked image to filename: %s\n", argv[2]);
    }


        return(0);
}
```

Looking at the example, we first do some really basic argument checking, accepting an input image as the first argument and an output image name for our watermarked copy. Using `imlib_load_image()` we load the input image and then grab its dimensions using the get functions. With the `imlib_clone_image()` function we can create a copy of the input image, which will be the base of our watermarked output. Next we load the watermark image, and notice that we then use `imlib_context_set_image()` to change the context from the input image (image_input) to the watermark image (image_watermark). Now we grab the images dimensions as well. In the final block we do two simple calculations to determine the positioning of the watermark on the output image, in this case I want the watermark on the bottom right-hand corner. The magic function that really does the work in this program is `imlib_blend_image_onto_image()`. Notice that we change context to the output image before proceeding. The blend function will, as the name suggests, blend two images together which we refer to as the source and destination image. The blend function blends a source image onto the current image context which we designate as the destination. The arguments supplied to `imlib_blend_image_onto_image()` can look tricky, we need to tell it which source to use (the watermark), whether to merge the alpha channel (0 for no), the dimensions of the source image (x, y, w, h) and the dimensions of the destination image (x, y, w, h). You'll notice that in the example we set the x and y positions of the source (watermark) image to 0 and then use the full width. The destination (input image) is set to the bottom right hand corner minus the dimensions of the watermark, and then we specify the width and height of the watermark. Finally, we use the `imlib_save_image()` function to save the output image.

While this example should be significantly improved for real use, it outlines the basics of Imlib2 blending to solves a very common problem efficiently.

Recipe: Image Scaling

dan 'dj2' sinclair <zero@perplexity.org>

As more people gain the ability to put images on the Internet it is often desired to scale those images to a smaller size to reduce bandwidth. This can easily be solved using a simple Imlib2 program.

This recipe takes the input image name, the new width, new height and the output image name and scales the input image by the given values, saving it back to the output image.

Example 2.2. Image Scaling

```
#define X_DISPLAY_MISSING

#include <Imlib2.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char ** argv) {
    Imlib_Image in_img, out_img;
    int w, h;

    if (argc != 5) {
        fprintf(stderr, "Usage: %s [in_img] [w] [h] [out_img]\n", argv[0]);
        return 1;
    }

    in_img = imlib_load_image(argv[1]);
    if (!in_img) {
        fprintf(stderr, "Unable to load %s\n", argv[1]);
        return 1;
    }
    imlib_context_set_image(in_img);
```

```
    w = atoi(argv[2]);
    h = atoi(argv[3]);
    out_img = imlib_create_cropped_scaled_image(0, 0, imlib_image_get_width(),
                                                imlib_image_get_height(), w, h );
    if (!out_img) {
        fprintf(stderr, "Failed to create scaled image\n");
        return 1;
    }

    imlib_context_set_image(out_img);
    imlib_save_image(argv[4]);
    return 0;
}
```

There is minimal argument checking done by this example, just make sure we have the correct number of arguments.

The source image is loaded with a call to `imlib_load_image()` which will load the image data into memory. If the call fails, NULL will be returned. Once we have the image data we need to set the image to be the current context. This lets Imlib2 know which image the operations will be preformed upon. This is done by calling `imlib_context_set_image()`. Once the image is set as the current context we can proceed with the scale. This is done by calling `imlib_create_cropped_scaled_image()` which takes as arguments, the starting x position, starting y position, the source width, source height, and the scaled width and scaled height. The reason we pass in the source information is that this function can also crop your image if desired. To crop, just modify the x, y, source width and source height as desired. This will result in a new image being produced out_img. If the scale fails, NULL will be returned. We then set the out_img to be the current context image and issue the save command, `imlib_save_image()`.

Although this program is simple, it shows the simplicity of image scaling using the Imlib2 API.

Recipe: Free rotation

dan 'dj2' sinclair <zero@perplexity.org>

It is sometimes desirable to rotate an image to some specific angle. Imlib2 makes this process easy. This example attempts to shows how its done. If you wish to rotate the image on angles of 90 degrees, see the 90 degree rotation recipe as this recipe will leave a black border around the image.

Example 2.3. Free rotation

```
#define X_DISPLAY_MISSING

#include <Imlib2.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

int main(int argc, char ** argv) {
    Imlib_Image in_img, out_img;
    float angle = 0.0;

    if (argc != 4) {
        fprintf(stderr, "Usage: %s [in_img] [angle] [out_img]\n", argv[0]);
```

```
        return 1;
    }

    in_img = imlib_load_image(argv[1]);
    if (!in_img) {
        fprintf(stderr, "Unable to load %s\n", argv[1]);
        return 1;
    }
    imlib_context_set_image(in_img);

    angle = (atof(argv[2]) * (M_PI / 180.0));
    out_img = imlib_create_rotated_image(angle);
    if (!out_img) {
        fprintf(stderr, "Failed to create rotated image\n");
        return 1;
    }

    imlib_context_set_image(out_img);
    imlib_save_image(argv[3]);
    return 0;
}
```

After some simple argument checking we get into the Imlib2 work. We begin by loading the specified image into memory with `imlib_load_image()` giving the image name as a parameter. We then take that image and make it the current context with `imblib_context_set_image`. Contexts are used by Imlib2 so it knows what image to work on. Whenever you wish to make imlib calls on an image it must be set as the current context. We then convert the given angle from Degrees to Radians as Imlib2s rotation function works in Radians. The rotation is then done with `imlib_create_rotated_image()`. The rotation function will return the new image. In order to save the new image we need to set it as the current context, again with `imlib_context_set_image()`. Then a simple call to `imlib_save_image()` giving the name of the output file saves the new, rotated image.

The rotation function in Imlib2 will place a black border around the image to fill in any blank space. This border is calculated so that the rotated image can fit in the output. This will case borders around the output image even if you rotate by 180 degrees.

Recipe: 90 degree Image rotation

dan 'dj2' sinclair <zero@perplexity.org>

With a digital camera it is sometimes desirable to rotate you image by: 90, 180 or 270 degrees. This recipe will show how to do this easily with Imlib2. This recipe, also, will not put the black borders around the image as is seen in the free rotate example.

Example 2.4. 90 degree Image rotation

```
#define X_DISPLAY_MISSING

#include <Imlib2.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char ** argv) {
    Imlib_Image in_img;
    int dir = 0;
```

```
    if (argc != 3) {
        fprintf(stderr, "Usage: %s [in_img] [out_img]\n", argv[0]);
        return 1;
    }

    in_img = imlib_load_image(argv[1]);
    if (!in_img) {
        fprintf(stderr, "Unable to load %s\n", argv[1]);
        return 1;
    }
    imlib_context_set_image(in_img);
    imlib_image_orientate(1);
    imlib_save_image(argv[2]);
    return 0;
}
```

After some minimal error checking we load the image to be rotated with a call to `imlib_load_image()`. This function accepts a filename and returns the Imlib_Image object, or NULL on load error. Once the image is loaded we set it as the current context image, the image Imlib2 will do its operations upon, with `imlib_context_set_image()`. The rotation is done through the call to: `imlib_image_orientate()`. The parameter to _orientate changes the amount of rotation. The possible values are: [1, 2, 3] meaning a clockwise rotation of [90, 180, 270] degrees respectively. Once the image is rotated we call `imlib_save_image()` giving the filename of the new image to have Imlib2 save the rotated image.

With this example in your hands you should be able to quickly rotate images on 90 degree intervals using Imlib2.

Recipe: Image Flipping

dan 'dj2' sinclair <zero@perplexity.org>

Imlib2 contains functions to do image flipping. This can be done either horizontally, vertically or diagonally. This recipe will show how to implement this functionality.

Example 2.5. Image Flipping

```
#define X_DISPLAY_MISSING

#include <Imlib2.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char ** argv) {
    Imlib_Image in_img;
    int dir = 0;

    if (argc != 4) {
        fprintf(stderr, "Usage: %s [in_img] [dir] [out_img]\n", argv[0]);
        return 1;
    }

    in_img = imlib_load_image(argv[1]);
    if (!in_img) {
        fprintf(stderr, "Unable to load %s\n", argv[1]);
        return 1;
```

```
    }
    imlib_context_set_image(in_img);

    dir = atoi(argv[2]);
    switch(dir) {
        case HORIZONTAL:
            imlib_image_flip_horizontal();
            break;

        case VERTICAL:
            imlib_image_flip_vertical();
            break;

        case DIAGONAL:
            imlib_image_flip_diagonal();
            break;

        default:
            fprintf(stderr, "Unknown value\n");
            return 1;
    }
    imlib_save_image(argv[3]);
    return 0;
}
```

This example does some minimal argument checking to begin, then loads the input image using the `imlib_load_image()` function, passing the filename to load. `imlib_load_image()` will either return the Imlib_Image object, or NULL if the load fails. Once we have the image object we set it as the current context image with a call to `imlib_context_set_image()`. This tells Imlib2 that this is the image we want to work with and all Imlib2 operations will work with this image. With the image context setup we decide on the type of flip we want to preform. This is done with one of the calls: `imlib_image_flip_horizontal()`, `imlib_image_flip_vertical()`, and `imlib_image_flip_diagonal()`. The diagonal flip essentially grabs the top left corner and makes it the bottom right corner. The top right becoming the bottom left. Once the image is flipped we call `imlib_save_image()` giving it the new filename and we're done.

This should give an example of image flipping with Imlib2. It will need enhancements before being put into a real app but the base is there.

Chapter 3. EVAS

Evas is a hardware-accelerated canvas API for X-Windows that can draw anti-aliased text, smooth super and sub-sampled images, alpha-blend, as well as drop down to using normal X11 primitives such as pixmaps, lines and rectangles for speed if your CPU or graphics hardware are too slow.

Evas abstracts any need to know much about the characteristics of your XServer's display, what depth or what magic visuals, it has. The most you need to tell Evas is how many colors (at a maximum) to use if the display is not a truecolor display. By default it is suggested to use 216 colors (as this equates to a 6x6x6 color cube - exactly the same color cube Netscape, Mozilla, gdkrgb etc. use so colors will be shared). If Evas can't allocate enough colors it keeps reducing the size of the color cube until it reaches plain black and white. This way, it can display on anything from a black and white only terminal to 16 color VGA to 256 color and all the way up through 15, 16, 24 and 32bit color.

Recipe: Using Ecore_Evas to simplify X11 canvas initialization

Ben technikolor Rockwood

Evas is a powerful and simple library to use, but before it can establish a canvas a X11 drawable needs to be setup. Manually setting up X11 can be a messy and frustrating task which detracts from concentrating on what you really want to do: develope an Evas application. But all this can be avoided by using the Ecore_Evas module of Ecore to do all the hard work for you.

The following example is a basic template that can be used as a starting point for any Evas application you want to build, significantly cutting down development time.

Example 3.1. Ecore_Evas Template

```
#include <Ecore_Evas.h>
#include <Ecore.h>

#define WIDTH 400
#define HEIGHT 400

        Ecore_Evas  *   ee;
        Evas        *   evas;
        Evas_Object *   base_rect;

int main(){

        ecore_init();

   ee = ecore_evas_software_x11_new(NULL, 0,  0, 0, WIDTH, HEIGHT);
        ecore_evas_title_set(ee, "Ecore_Evas Template");
        ecore_evas_borderless_set(ee, 0);
        ecore_evas_show(ee);


   evas = ecore_evas_get(ee);
        evas_font_path_append(evas, "fonts/");


   base_rect = evas_object_rectangle_add(evas);
        evas_object_resize(base_rect, (double)WIDTH, (double)HEIGHT);
```

```
        evas_object_color_set(base_rect, 244, 243, 242, 255);
        evas_object_show(base_rect);

        /* Insert Object Here */

        ecore_main_loop_begin();

        return 0;
}
```

Full details on Ecore_Evas can be found in the Ecore chapter of this book, but this basic template should get you playing with Evas right away. The important calls to note are `ecore_evas_borderless_set()` which defines whether the Evas window is windowed by your window manager or borderless, and `evas_font_path_append()` which defines the font path(s) used by your Evas app.

Recipe: Key Binds, using EVAS Key Events

Ben technikolor Rockwood

Many applications can benifit from providing key binds for commonly used operations. Whether accepting text in ways the EFL doesn't normally expect or just a way to bind the + key to raise the volume of a mixer, keybinds can add just the bit of functionality that makes your app a hit.

The following code is a simple and complete application that is useful in exploring keybinds using EVAS event callbacks. It creates a black 100 by 100 pixel window in which you can hit keys.

Example 3.2. Key grabbing using EVAS Events

```
#include <Ecore_Evas.h>
#include <Ecore.h>

#define WIDTH 100
#define HEIGHT 100

        Ecore_Evas  *   ee;
        Evas        *   evas;
        Evas_Object *   base_rect;

static int main_signal_exit(void *data, int ev_type, void *ev)
{
   ecore_main_loop_quit();
   return 1;
}

void key_down(void *data, Evas *e, Evas_Object *obj, void *event_info) {
        Evas_Event_Key_Down *ev;

        ev = (Evas_Event_Key_Down *)event_info;
        printf("You hit key: %s\n", ev->keyname);
}

int main(){
        ecore_init();
        ecore_event_handler_add(ECORE_EVENT_SIGNAL_EXIT,
                        main_signal_exit, NULL);
```

```
   ee = ecore_evas_software_x11_new(NULL, 0,  0, 0, WIDTH, HEIGHT);
        ecore_evas_title_set(ee, "EVAS KeyBind Example");
        ecore_evas_borderless_set(ee, 0);
        ecore_evas_show(ee);

   evas = ecore_evas_get(ee);

   base_rect = evas_object_rectangle_add(evas);
        evas_object_resize(base_rect, (double)WIDTH, (double)HEIGHT);
        evas_object_color_set(base_rect, 0, 0, 0, 255);
        evas_object_focus_set(base_rect, 1);
        evas_object_show(base_rect);

        evas_object_event_callback_add(base_rect,
                        EVAS_CALLBACK_KEY_DOWN, key_down, NULL);

        ecore_main_loop_begin();

        ecore_evas_shutdown();
        ecore_shutdown();

        return 0;
}
```

You can compile this example in the following manner:

Example 3.3. EVAS Keybind Compile

```
gcc `evas-config --libs --cflags` `ecore-config --libs --cflags` \
> key_test.c -o key_test
```

In this example the canvas is setup in the usual way using Ecore_Evas to do the dirty work. The real magic occurs in the `evas_object_event_callback_add()` callback.

```
        evas_object_event_callback_add(base_rect,
                        EVAS_CALLBACK_KEY_DOWN, key_down, NULL);
```

By adding a callback to the base_rect, which is acting as the canvas background, we can execute a function (`key_down()` in this case) whenever we encounter a key down event, defined in `Evas.h` as EVAS_CALLBACK_KEY_DOWN.

There is one very important thing to do in addition to defining the callback: setting the focus. The `evas_object_focus_set()` function sets the focus on a given Evas object. It is the object that has focus that will actually accept the events, even though you explicitly define the Evas object that the callback is attached to. And only one object can be focused at a time. The most common problem encountered with Evas callbacks is forgetting to set focus.

```
void key_down(void *data, Evas *e, Evas_Object *obj, void *event_info) {
        Evas_Event_Key_Down *ev;

        ev = (Evas_Event_Key_Down *)event_info;
        printf("You hit key: %s\n", ev->keyname);
```

```
}
```

The `key_down()` function is called anytime a key down event occurs after defining it's callback. The function declaration is that of a standard Evas callback (see `Evas.h`). The important piece of information we need to know is what key was pressed, which is contained in the Evas `event_info` structure. After setting up the `Evas_Event_Key_Down` structure for use as seen above we can access the element `keyname` to determine which key was pressed.

In most cases you'll likely use a `switch` or nested `if`'s to define which keys do what, and it's recommended that this functionality be paired with an configuration EDB to provide centralization and easy expansion of your applications keybind settings.

Recipe: Evas Smart Object Introduction

dan 'dj2' sinclair <zero@perplexity.org>

As you work with Evas more, you'll begin to have several `Evas_Objects` that you are working with and applying the same operations to keep them in sync. It would be much more convenient to group all of these separate `Evas_Object`s into a single object that the transforms can be applied too.

Evas smart objects provide the ability to write your own objects and have Evas call your functions to do the moving, resizing, hiding, layer and all of those other things an `Evas_Object` is responsible to handle. Along with the normal `Evas_Object` callbacks, smart objects allow you to define your own functions for the object to handle any special operations you may require.

This introduction is broken into three files: foo.h, foo.c, and main.c. The smart object being created is called `foo` and is defined in the foo.[ch] files. main.c is there to show how the new smart object can be used.

The smart object itself is just two squares, one inside the other, with the inner one offset 10% from the edge of the outer square. As the main program executes an Ecore timer callback will reposition and resize the smart object.

The basic file for this smart object is from an Evas Smart Object template by Atmos located at: `www.atmos.org/code/src/evas_smart_template_atmos.c [http://www.atmos.org/code/src/evas_smart_template_atmos.c]` which in turn was based off of a template by Rephorm.

First we need to define the external interface to our smart object. In this case we only need a call to create the new object.

Example 3.4. foo.h

```
#ifndef _FOO_H_
#define _FOO_H_

#include <Evas.h>

Evas_Object *foo_new(Evas *e);

#endif
```

With that out of the way, we get into the dark underbelly of the beast, the smart object code.

Example 3.5. foo.c

```
#include <Evas.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct _Foo_Object Foo_Object;
struct _Foo_Object {
        Evas_Object *clip;
        Evas_Coord x, y, w, h;

        Evas_Object *outer;
        Evas_Object *inner;
};
```

The `Foo_Object` will store all the information we need for our object. In this case it is the outer box object, the inner box object, a clipping object and the current position and size of the object.

```
static Evas_Smart *_foo_object_smart_get();
static Evas_Object *foo_object_new(Evas *evas);
static void _foo_object_add(Evas_Object *o);
static void _foo_object_del(Evas_Object *o);
static void _foo_object_layer_set(Evas_Object *o, int l);
static void _foo_object_raise(Evas_Object *o);
static void _foo_object_lower(Evas_Object *o);
static void _foo_object_stack_above(Evas_Object *o, Evas_Object *above);
static void _foo_object_stack_below(Evas_Object *o, Evas_Object *below);
static void _foo_object_move(Evas_Object *o, Evas_Coord x, Evas_Coord y);
static void _foo_object_resize(Evas_Object *o, Evas_Coord w, Evas_Coord h);
static void _foo_object_show(Evas_Object *o);
static void _foo_object_hide(Evas_Object *o);
static void _foo_object_color_set(Evas_Object *o, int r, int g, int b, int a);
static void _foo_object_clip_set(Evas_Object *o, Evas_Object *clip);
static void _foo_object_clip_unset(Evas_Object *o);
```

The predeclarations required for the smart object. These will be explained as we come to there actual implementation.

```
Evas_Object *foo_new(Evas *e) {
    Evas_Object *result = NULL;
    Foo_Object *data = NULL;

    if ((result = foo_object_new(e))) {
        if ((data = evas_object_smart_data_get(result)))
            return result;
        else
            evas_object_del(result);
    }

    return NULL;
}
```

`foo_new()` is our one external interface and is responsible for setting up the object itself. The call to `foo_object_new()` will do all of the heavy lifting in the object creation. The

`evas_object_smart_data_get()` is more of an error check then anything else. When the `foo_object_new()` runs it will add the smart object to the evas and this will result in an `add` call on the object. In our case this `add` call will create a `Foo_Object`. So, we're just making sure that the `Foo_Object` has been created.

```
static Evas_Object *foo_object_new(Evas *evas) {
    Evas_Object *foo_object;

    foo_object = evas_object_smart_add(evas, _foo_object_smart_get());
    return foo_object;
}
```

Our `foo_object_new()` function has the simple task of adding our smart object onto the given Evas. This is done through the `evas_object_smart_add()` passing the Evas and the `Evas_Smart *` object. Our `Evas_Smart *` is produced by the `_foo_object_smart_get()` call.

```
static Evas_Smart *_foo_object_smart_get() {
    static Evas_Smart *smart = NULL;
    if (smart)
        return (smart);

    smart = evas_smart_new("foo_object",
                           _foo_object_add,
                           _foo_object_del,
                           _foo_object_layer_set,
                           _foo_object_raise,
                           _foo_object_lower,
                           _foo_object_stack_above,
                           _foo_object_stack_below,
                           _foo_object_move,
                           _foo_object_resize,
                           _foo_object_show,
                           _foo_object_hide,
                           _foo_object_color_set,
                           _foo_object_clip_set,
                           _foo_object_clip_unset,
                           NULL
                          );
    return smart;
}
```

You'll notice that the `Evas_Smart *smart` in this function is declared `static`. This is because no matter how many `Evas_Objects` we create, there will be only one `Evas_Smart` object. As Raster put it, an `Evas_Smart` is like a C++ class definition, not an instance. The `Evas_Object` is the instance of the `Evas_Smart`.

The smart object itself is created through the call to `evas_smart_new()`. To this function we pass in the name of the smart object, all of the callback routines for the smart object, and any user data. In this case we don't have user data so we set it to NULL.

```
static void _foo_object_add(Evas_Object *o) {
    Foo_Object *data = NULL;
    Evas *evas = NULL;

    evas = evas_object_evas_get(o);
```

```
    data = (Foo_Object *)malloc(sizeof(Foo_Object));
    memset(data, 0, sizeof(Foo_Object));

    data->clip = evas_object_rectangle_add(evas);
    data->outer = evas_object_rectangle_add(evas);
    evas_object_color_set(data->outer, 0, 0, 0, 255);
    evas_object_clip_set(data->outer, data->clip);
    evas_object_show(data->outer);

    data->inner = evas_object_rectangle_add(evas);
    evas_object_color_set(data->inner, 255, 255, 255, 126);
    evas_object_clip_set(data->inner, data->clip);
    evas_object_show(data->inner);

    data->x = 0;
    data->y = 0;
    data->w = 0;
    data->h = 0;

    evas_object_smart_data_set(o, data);
}
```

When the call to `evas_object_smart_add()` is called in `foo_object_new()`, this function, `_foo_object_add()` will be called so we can setup any internal data for this smart object.

For this smart object we setup three internal `Evas_Object`s. Those being `data->clip` used for clipping the other two objects, `data->outer` our outer rectangle and `data->inner` our inner rectangle. The inner and outer rectangles have there clipping set to the clip object and are shown immediately. The clip object is not shown, it will be shown when the user calls `evas_object_show()` on this object.

Finally we call `evas_object_smart_data_set()` to set our new `Foo_Object` as data to this smart object. This data will be retrieved in the other functions of this object by calling `evas_object_smart_data_get()`.

```
static void _foo_object_del(Evas_Object *o) {
    Foo_Object *data;

    if ((data = evas_object_smart_data_get(o))) {
        evas_object_del(data->clip);
        evas_object_del(data->outer);
        evas_object_del(data->inner);
        free(data);
    }
}
```

The `_foo_object_del()` callback will be executed if the user calls `evas_object_del()` on our object. For this object its as simple as `evas_object_del`ing our three rectangles and freeing our `Foo_Object` data structure.

```
static void _foo_object_layer_set(Evas_Object *o, int l) {
    Foo_Object *data;

    if ((data = evas_object_smart_data_get(o))) {
        evas_object_layer_set(data->clip, l);
    }
}
```

```
static void _foo_object_raise(Evas_Object *o) {
    Foo_Object *data;

    if ((data = evas_object_smart_data_get(o))) {
        evas_object_raise(data->clip);
    }
}

static void _foo_object_lower(Evas_Object *o) {
    Foo_Object *data;

    if ((data = evas_object_smart_data_get(o))) {
        evas_object_lower(data->clip);
    }
}

static void _foo_object_stack_above(Evas_Object *o, Evas_Object *above) {
    Foo_Object *data;

    if ((data = evas_object_smart_data_get(o))) {
        evas_object_stack_above(data->clip, above);
    }
}

static void _foo_object_stack_below(Evas_Object *o, Evas_Object *below) {
    Foo_Object *data;

    if ((data = evas_object_smart_data_get(o))) {
        evas_object_stack_below(data->clip, below);
    }
}
```

This group of functions: `_foo_object_layer_set()`, `_foo_object_raise()`, `_foo_object_lower()`, `_foo_object_stack_above()`, and `_foo_object_stack_below()` all work in the same fashion, applying the required `evas_object_*` function on the `data->clip` object.

These functions are triggered through the use of: `evas_object_layer_set()`, `evas_object_raise()`, `evas_object_lower()`, `evas_object_stack_above()`, and `evas_object_stack_below()` respectively.

```
static void _foo_object_move(Evas_Object *o, Evas_Coord x, Evas_Coord y) {
    Foo_Object *data;

    if ((data = evas_object_smart_data_get(o))) {
        float ix, iy;
        ix = (data->w - (data->w * 0.8)) / 2;
        iy = (data->h - (data->h * 0.8)) / 2;

        evas_object_move(data->clip, x, y);
        evas_object_move(data->outer, x, y);
        evas_object_move(data->inner, x + ix, y + iy);

        data->x = x;
        data->y = y;
    }
}
```

The `_foo_object_move()` callback will be triggered when `evas_object_move()` is called on our object. Each of the internal objects is moved into its correct positioning with calls to `evas_object_move()`.

```
static void _foo_object_resize(Evas_Object *o, Evas_Coord w, Evas_Coord h) {
    Foo_Object *data;

    if ((data = evas_object_smart_data_get(o))) {
        float ix, iy, iw, ih;
        iw = w * 0.8;
        ih = h * 0.8;

        ix = (w - iw) / 2;
        iy = (h - iw) / 2;

        evas_object_resize(data->clip, w, h);
        evas_object_resize(data->outer, w, h);

        evas_object_move(data->inner, data->x + ix, data->y + iy);
        evas_object_resize(data->inner, iw, ih);

        data->w = w;
        data->h = h;
    }
}
```

The `_foo_object_resize()` calback will be triggered when the user calls `evas_object_resize()` on our object. So, for our object, we need to resize `data->clip` and `data->outer` to the full size available for our object. This is done with the `evas_object_resize()` calls. We then need to move and resize the `data->inner` object so it stays in the correct positioning in the outer rectangle. These are done with the `evas_object_move()` and `evas_object_resize()` respectively. We then store the current width and height back into our object so we can reference them later.

```
static void _foo_object_show(Evas_Object *o) {
    Foo_Object *data;

    if ((data = evas_object_smart_data_get(o)))
            evas_object_show(data->clip);
}
```

The `_foo_object_show()` callback will be triggered when `evas_object_show()` is called on our object. In order to show our object all we need to do is show the clip region as our actual rectangles are clipped to it. This is done through the call to `evas_object_show()`.

```
static void _foo_object_hide(Evas_Object *o) {
    Foo_Object *data;

    if ((data = evas_object_smart_data_get(o)))
        evas_object_hide(data->clip);
}
```

The `_foo_object_hide()` callback will be triggered when a call to `evas_object_hide()` is made on our object. Because we are using in internal clipping object we just need to hide the clip object, `data->clip`, to hide our smart object. This is done through the call to `evas_object_hide()`.

```
static void _foo_object_color_set(Evas_Object *o, int r, int g, int b, int a) {
}
```

The `_foo_object_color_set()` function will be called when `evas_object_color_set()` is called on our `Evas_Object`. But, since I don't want my object to change colours, I ignore this callback.

```
static void _foo_object_clip_set(Evas_Object *o, Evas_Object *clip) {
    Foo_Object *data;

    if ((data = evas_object_smart_data_get(o)))
        evas_object_clip_set(data->clip, clip);
}
```

The `_foo_object_clip_set()` callback will be triggered when the `evas_object_clip_set()` call is made on our object. In this case we propagate this setting to our internal clipping object, data->clip through the call to `evas_object_clip_set()`.

```
static void _foo_object_clip_unset(Evas_Object *o) {
    Foo_Object *data;

    if ((data = evas_object_smart_data_get(o)))
        evas_object_clip_unset(data->clip);
}
```

The `_foo_object_clip_unset()` callback will be triggered when a call to `evas_object_clip_unset()` is made on our object. We just remove the clip set to our internal clipping object through the call to `evas_object_clip_unset()`.

Once the smart object code is complete we can create our main program to utilize the new smart object.

Example 3.6. main.c

```
#include <stdio.h>
#include <Ecore_Evas.h>
#include <Ecore.h>
#include "foo.h"

#define WIDTH 400
#define HEIGHT 400
#define STEP 10

static int dir = -1;
static int cur_width = WIDTH;
static int cur_height = HEIGHT;

static int timer_cb(void *data);

int main() {
    Ecore_Evas  *ee;
    Evas        *evas;
    Evas_Object *o;

    ecore_init();

    ee = ecore_evas_software_x11_new(NULL, 0, 0, 0, WIDTH, HEIGHT);
    ecore_evas_title_set(ee, "Smart Object Example");
```

```
    ecore_evas_borderless_set(ee, 0);
    ecore_evas_show(ee);

    evas = ecore_evas_get(ee);

    o = evas_object_rectangle_add(evas);
    evas_object_resize(o, (double)WIDTH, (double)HEIGHT);
    evas_object_color_set(o, 200, 200, 200, 255);
    evas_object_layer_set(o, -255);
    evas_object_show(o);

    o = foo_new(evas);
    evas_object_move(o, 0, 0);
    evas_object_resize(o, (double)WIDTH, (double)HEIGHT);
    evas_object_layer_set(o, 0);
    evas_object_show(o);

    ecore_timer_add(0.1, timer_cb, o);
    ecore_main_loop_begin();

    return 0;
}

static int timer_cb(void *data) {
    Evas_Object *o = (Evas_Object *)data;
    Evas_Coord x, y;

    cur_width += (dir * STEP);
    cur_height += (dir * STEP);

    x = (WIDTH - cur_width) / 2;
    y = (HEIGHT - cur_height) / 2;

    if ((cur_width < STEP) || (cur_width > (WIDTH - STEP)))
        dir *= -1;

    evas_object_move(o, x, y);
    evas_object_resize(o, cur_width, cur_height);
    return 1;
}
```

Most of this program is the same as that given in the using Ecore_Evas recipe given earlier. The creation of our new smart object is seen in the code snippet:

```
    o = foo_new(evas);
    evas_object_move(o, 0, 0);
    evas_object_resize(o, (double)WIDTH, (double)HEIGHT);
    evas_object_layer_set(o, 0);
    evas_object_show(o);
```

Once your new `foo_new()` returns our object we can manipulate it with the normal Evas calls, so we proceed to set it position, size, layer and then show the object.

Once the new smart object is created and shown we setup an Ecore timer to trigger every 0.1 seconds. When the timer is triggered it will execute the function `timer_cb()`. This callback function will either shrink, or grow the size of our smart object while keeping it centered in the main window.

Compiling this example is as simple as:

Example 3.7. Compilation

```
zero@oberon [evas_smart] -> gcc -o foo foo.c main.c \
     `ecore-config --cflags --libs` `evas-config --cflags --libs`
```

Evas smart objects are simple to create but provide a powerful mechanism to abstract out pieces of your program. To see some more smart objects take a look at any of the Esmart objects, Etox or Emotion.

Chapter 4. Ecore

What is Ecore? Ecore is the core event abstraction layer and X abstraction layer that makes doing selections, Xdnd, general X stuff, and event loops, timeouts and idle handlers fast, optimized, and convenient. It's a separate library so anyone can make use of the work put into Ecore to make this job easy for applications.

Ecore is completely modular. At its base is the event handlers and timers, and initialization and shutdown functions. The abstraction modules for Ecore include:

- Ecore X
- Ecore FB
- Ecore EVAS
- Ecore TXT
- Ecore Job
- Ecore IPC
- Ecore Con
- Ecore Config

Ecore is so modular and powerful that it can be extremely useful even in non-graphics programing by itself. As an example, several web servers have been written that were based solely on Ecore and the Ecore_Con module for abstract socket communication.

Recipe: Ecore Config Introduction

dan 'dj2' sinclair <zero@perplexity.org>

The Ecore_Config module provides the programmer with a very simple way to setup configuration files for their program. This recipe will give an example of how to integrate the beginnings of Ecore_Config into your program and use it to get configuration data.

Example 4.1. Simple Ecore_Config program

```
#include <Ecore_Config.h>

int main(int argc, char ** argv) {
    int i;
    float j;
    char *str;

    if (ecore_config_init("foo") != ECORE_CONFIG_ERR_SUCC) {
        printf("Cannot init Ecore_Config");
        return 1;
    }

    ecore_config_int_default("/int_example", 1);
    ecore_config_string_default("/this/is/a/string/example", "String");
    ecore_config_float_default("/float/example", 2.22);

    ecore_config_load();

    i = ecore_config_int_get("/int_example");
    str = ecore_config_string_get("/this/is/a/string/example");
    j = ecore_config_float_get("/float/example");
```

```
    printf("str is (%s)\n", str);
    printf("i is (%d)\n", i);
    printf("j is (%f)\n", j);

    free(str);

    ecore_config_shutdown();
    return 0;
}
```

As you can see from this example the basic usage of Ecore_Config is simple. The system is initialized with a call to `ecore_config_init`. The program name setting control where Ecore_Config will look for your configuration database. The directory and file name are: ~/.e/apps/PROGRAM_NAME/config.db.

For each configuration variable you are getting from Ecore_Config, you can assign a default value in the case that the user does not have a config.db file. The defaults are assigned with the ecore_config_*_default where * is one of the Ecore_Config types. The first parameter is the key under which this is to be accessed. These keys must be unique over your program. The value passed is of the type appropriated for this call.

The `ecore_config_load` call will read the values from the config.db file into Ecore_Config. After which we can access the files with the ecore_config_*_get methods (again * is the type of data desired). These routines take the key name for this item and return the value associated with that key. Each function returns a type that corresponds to the function call name.

`ecore_config_shutdown` is then called to shutdown the Ecore_Config system before the program exits.

Example 4.2. Compilation command

```
gcc -o ecore_config_example ecore_config_example.c `ecore-config --cflags --libs`
```

To compile the program you can use the ecore-config script to get all of the required linking and library information for Ecore_Config. If you run this program as is you will receive the values put into ecore_config as the defaults as output. Once you know the program is working, you can create a simple config.db file to read the values.

Example 4.3. Simple config.db script (build_cfg_db.sh)

```
#!/bin/sh

DB=config.db

edb_ed $DB add /int_example int 2
edb_ed $DB add /this/is/a/string/example str "this is a string"
edb_ed $DB add /float/example float 42.10101
```

When build_cfg_db.sh is executed it will create a config.db file in the current directory. This file can

then be copied into ~/.e/apps/PROGRAM_NAME/config.db where PROGRAM_NAME is the value passed into ecore_config_init. Once the file is copied in place, executing the test program again will show the values given in the config file instead of the defaults. You can specify as many, or as few of the configuration keys in the config file and Ecore_Config will either show the user value or the default value.

Recipe: Ecore Config Listeners

dan 'dj2' sinclair <zero@perplexity.org>

When using Ecore Config to handle the configuration of your application it is nice to be able to be notified when that configuration has been changed. This is accomplished through the use of listeners in Ecore_Config.

Example 4.4. Ecore_Config listener

```
#include <Ecore.h>
#include <Ecore_Config.h>

static int listener_cb(const char *key, const Ecore_Config_Type type,
            const int tag, void *data);

enum {
    EX_ITEM,
    EX_STR_ITEM,
    EX_FLOAT_ITEM
};

int main(int argc, char ** argv) {
    int i;
    float j;
    char *str;

    if (!ecore_init()) {
        printf("Cannot init ecore");
        return 1;
    }

    if (ecore_config_init("foo") != ECORE_CONFIG_ERR_SUCC) {
        printf("Cannot init Ecore_Config");
        ecore_shutdown();
        return 1;
    }

    ecore_config_int_default("/int/example", 1);
    ecore_config_string_default("/string/example", "String");
    ecore_config_float_default("/float/example", 2.22);

    ecore_config_listen("int_ex", "/int/example", listener_cb,
                                                  EX_ITEM, NULL);
    ecore_config_listen("str_ex", "/string/example", listener_cb,
                                                  EX_STR_ITEM, NULL);
    ecore_config_listen("float_ex", "/float/example", listener_cb,
                                                  EX_FLOAT_ITEM, NULL);

    ecore_main_loop_begin();
    ecore_config_shutdown();
    ecore_shutdown();
```

```
    return 0;
}

static int listener_cb(const char *key, const Ecore_Config_Type type,
                                                const int tag, void *data) {

    switch(tag) {
        case EX_ITEM:
            {
                int i = ecore_config_int_get(key);
                printf("int_example :: %d\n", %i);
            }
            break;

        case EX_STR_ITEM:
            {
                char *str = ecore_config_string_get(key);
                printf("str :: %s\n", %str);
                free(str);
            }
            break;

        case EX_FLOAT_ITEM:
            {
                float f = ecore_config_float_get(key);
                printf("float :: %f\n", %f);
            }
            break;

        default:
            printf("Unknown tag (%d)\n", tag);
            break;
    }
}
```

Ecore_Config is setup in the usual way, and we create some default keys as happens normally. The interesting parts come into play with the calls to `ecore_config_listen()`. This is the call that tells Ecore_Config we want to be notified of configuration changes. `ecore_config_listen()` takes five parameters:

- name
- key
- listener callback
- id tag
- user data

The `name` field is a name string given by you to identify this listener callback. The `key` is the name of the key you wish to listen on, this will be the same as the key name given in the _default calls above. The `listener callback` is the calback function to be executed on change. The `id tag` is a numeric tag that can be given to each listener and will be passed to the calback function. Finally, `user data` is any data you wish to have passed to the callback when it is executed.

The callback function has a signature similar to:

```
int listener_cb(const char *key, const Ecore_Config_Type type,
                                      const int tag, void *data);
```

The `key` is the key name that was being listen on. The `type` parameter will contain the Ecore_Config type. This can be one of:

PT_NIL	Property with no value
PT_INT	Integer property
PT_FLT	Float property
PT_STR	String property
PT_RGB	Colour property
PT_THM	Theme property

The `tag` parameter is the value that was given in the listener creation call above. Finally, the `data` is any user data attached to the listener when it was created.

If you wish to remove the listener at a later date `ecore_config_deaf()` is provided. This takes three parameters:

- name
- key
- listener callback

Each of these parameters corresponds to the parameter given in the initial `ecore_config_listen()` call.

Example 4.5. Compilation

```
zero@oberon [ecore_config] -> gcc -o ecfg ecfg_listener.c \
     `ecore-config --cflags --libs`
```

If you execute the program you will see the default values printed back to the screen. If you now launch examine as follows:

```
zero@oberon [ecore_config] -> examine foo
```

(foo is the name given to `ecore_config_init()`). You should then be able to modify the settings to the application and, after pressing 'save', see the modified values printed back to the console.

Recipe: Connecting to a server with Ecore_Con

dan 'dj2' sinclair <zero@perplexity.org>

Client/server are becoming quite common these days. To that end, Ecore can make your life simpler. Ecore has an Ecore_Con subsystem which handles all the sticky bits of connecting to servers and having clients connect to your server.

This recipe looks at connecting to another server and receiving some information back. In this case we will be connecting to the enlightenment website and grabbing one of the pages.

Now, before you get comfused. Ecore_Con terminology can be a bit confusing till you wrap your brain around it. When you are connecting to a server you will be working with the server calls in ecore_con. This includes the server callbacks.

As a side note, I do very little checking of return values in this program. You'll probably want to do a

bunch more if your using this yourself.

Example 4.6. preamble

```
#include <Ecore.h>
#include <Ecore_Con.h>

static int server_add_cb(void *data, int type, void *ev);
static int server_del_cb(void *data, int type, void *ev);
static int server_data_cb(void *data, int type, void *ev);

struct Data {
    char *data;
    int data_size;
};
```

If all your using is ecore_con all you need are the `Ecore.h` and the `Ecore_Con.h` headers to get you going. We have a couple of predeclaraions to keep the compiler happy and a simple data struct we'll be passing around.

Example 4.7. gettin' the party started

```
int
main(int argc, char ** argv)
{
    struct Data gd;
    struct Data *sd;
    Ecore_Con_Server *srv;
    Ecore_Event_Handler *add = NULL, *del = NULL, *data = NULL;

    sd = calloc(1, sizeof(struct Data));
    gd.data = "In the land of the night "
              "the ship of the sun "
              "is drawn by the grateful dead."
              "~ Egyptian book of the Dead";


    ecore_init();
    ecore_con_init();
```

We start into the main body of the program with some declarations. When you call the connect method for ecore_con, it will return you a `Ecore_Con_Server *` which we can use to manipulate the connection if we so desire.

I'm also keeping around the `Ecore_Event_Handler`s that I will be creating so I can be a good little program and clean up after myself a bit.

The `gd` variable is my “global” data. It will be pass to every call of each of the handlers. The `sd` variable is my “local” data. It will be passed to the handler when its server is acted upon.

But, before we actually do anything cool, we must make sure to setup ecore and ecore_con with the calls to `ecore_init()` and `ecore_con_init()`. As I mentioned before, you should really be checking return values.

Example 4.8. hook her up

```
add = ecore_event_handler_add(ECORE_CON_EVENT_SERVER_ADD,
                              server_add_cb, &gd);
del = ecore_event_handler_add(ECORE_CON_EVENT_SERVER_DEL,
                              server_del_cb, &gd);
data = ecore_event_handler_add(ECORE_CON_EVENT_SERVER_DATA,
                              server_data_cb, &gd);
```

If we actually want to receive any events for anything we need to hook ourself into the ecore event system. This is done through calls to `ecore_event_handler_add()`. In the case of connecting to another server the events we are interested in are:

ECORE_CON_EVENT_SERVER_ADD	Called when we have connected to the server
ECORE_CON_EVENT_SERVER_DEL	Called when we have disconnected from the server
ECORE_CON_EVENT_SERVER_DATA	Called when we have received data from the server

To each one of these handler calls we pass in the function that will be handling the event and the global data `gd` that we created above. If you don't have any global data you can always pass `NULL` as the pointer.

Example 4.9. connecting

```
srv = ecore_con_server_connect(ECORE_CON_REMOTE_SYSTEM,
                               "www.enlightenment.org", 80, sd);
```

Once everything is setup we can create the actual connection to the server. This is done with a call to `ecore_con_server_connect()`. The connect call will return a `Ecore_Con_Server *` to us that we can keep around if we so desire. The connect call takes four parameters. The type of connection, the host to connect to, the port to connect to, and any data associated with this server.

The connection type is one of:

ECORE_CON_LOCAL_USER	This will connect to the server listening on the Unix socket at `~/.ecore/[name]/[port]`
ECORE_CON_LOCAL_SYSTEM	This will connect to the server listening on the Unix socket at `/tmp/.ecore_service|[name]|[port]`
ECORE_CON_REMOTE_SYSTEM	This will connect to the server listening on the TCP port `[name]:[port]`
ECORE_CON_USE_SSL	If SSL was compiled into the library this will instruct the connect to use SSL encrypton with the connection

The last parameter is data that is specifc to this server connection that we want to pass to each of the

handlers. If you have no per-server data then you can safley pass `NULL` as the parameter.

Example 4.10. go speed racer

```
    ecore_main_loop_begin();

    ecore_event_handler_del(add);
    ecore_event_handler_del(del);
    ecore_event_handler_del(data);

    ecore_con_shutdown();
    ecore_shutdown();
    return 0;
}
```

Now that we're connected we startup the main event loop with `ecore_main_loop_begin()`. Since I'm a nice programmer I'm also cleaning up my handlers when we exit the main loop with, `ecore_event_handler_del()`. After that, its just a matter of shutting down everything we started, `ecore_con_shutdown()` and `ecore_shutdown()` do the trick for us in this case.

Of course, you can always create your connections after the main loop has started, everything would work the same way (tho you only need to register the event handlers once). I'm just doing it before since its easier in the case of this example.

Example 4.11. added

```
static int
server_add_cb(void *data, int type, void *ev)
{
    Ecore_Con_Event_Server_Add *e;
    struct Data *sd;
    struct Data *gd;
    char buf[1024];

    e = ev;
    gd = data;
    sd = ecore_con_server_data_get(e->server);

    printf("Connected to server ...\n");

    snprintf(buf, 1024, "GET http://www.enlightenment.org/"
                        "pages/enlightenment.html HTTP/1.0\r\n");
    ecore_con_server_send(e->server, buf, strlen(buf));

    snprintf(buf, 1024, "\r\n");
    ecore_con_server_send(e->server, buf, strlen(buf));

    return 1;
}
```

So now, the `ECORE_CON_EVENT_SERVER_ADD` handler, in this case `server_add_cb` will be triggered when we have established a connection of the server.

The add event will provide use with a `Ecore_Con_Event_Server_Add` structure containing information about this event. The principle one we will be concerned with is the `server` member which is the server handle the same as was returned from the connect call.

The global data that we set on the handler will be passed in the `void *data` parameter and the per-server data that we set on the connect call can be retrieved with a call to `ecore_con_server_data_get()`.

Now that we're connected to the server we can send a request for a document, in this case I'm sending a HTTP request. Data is sent to the server by calling, `ecore_con_server_send()` and passing in the server to send too, the data, and the length of the data.

Make sure to return 1 from each of your handlers so Ecore will continue to call it. If you return 0 Ecore will remove the handler from its list of available handlers.

Example 4.12. del'd

```
static int
server_del_cb(void *data, int type, void *ev)
{
    Ecore_Con_Event_Server_Del *e;
    struct Data *sd;
    struct Data *gd;

    e = ev;
    gd = data;
    sd = ecore_con_server_data_get(e->server);

    ecore_con_server_del(e->server);

    printf("%s\n\n", gd->data);
    if (sd->data) {
        printf("%s\n", sd->data);
        free(sd->data);
    }

    ecore_main_loop_quit();
    return 1;
}
```

We now come to the `ECORE_CON_EVENT_SERVER_DEL` handler. This works similar to the add handler but the event type passed is an, `Ecore_Con_Event_Server_Del`.

Now, the del callback will be triggered when we disconnect from the remote server. This means that it is our responsiblity to clean up the server memory. This is done with a call to `ecore_con_server_del()`. We, of course, don't have to clean up here, we can do it whenever it is convenient for us.

In the case of this example, since the webserver will close the connection once its returned the webpage, I am doing the data processing in the del handler. I then don't want to continue after gettting the page so I call `ecore_main_loop_quit()` to break the main event loop.

Example 4.13. data

```
static int
server_data_cb(void *data, int type, void *ev)
{
    Ecore_Con_Event_Server_Data *e;
    struct Data *sd;
    struct Data *gd;

    e = ev;
    gd = data;
    sd = ecore_con_server_data_get(e->server);

    sd->data = realloc(sd->data, sd->data_size + e->size + 1);
    memcpy(sd->data + sd->data_size, e->data, e->size);

    sd->data_size += e->size;
    sd->data[sd->data_size] = '\0';

    return 1;
}
```

The `ECORE_CON_EVENT_SERVER_DATA` is again similar to the add and del callbacks. The event structure this time is a `Ecore_Con_Event_Server_Data` and presents us with two extra interesting members, those being: `size` and `data`. These provide us with the data received from the server and the size of that data. This data is *not* NULL terminated so make sure you use the `size` paramater.

In my case I'm just going to store the data in my per-server structure and deal with it when I get the disconnection. You can of course do any procesing you want in here.

Example 4.14. compilation

```
zero@oberon [ecore_con] -> gcc -o srv main.c `ecore-config --cflags --libs`
```

Once its all built, compilation is a simple matter. The program can then be executed with a simple

```
./srv
```

command and it should print out the selected webpage.

Ecore_Con makes it easy to work with any kind of remote servers, from HTTP to IRC to something custom, all the functionaility is wrapped and handled nicely through the Ecore event loop.

Recipe: Ecore Ipc Introduction

dan 'dj2' sinclair <zero@perplexity.org>

The Ecore_Ipc library provides a robust and efficient wrapper around the Ecore_Con module. Ecore_Ipc allows you to set up your server communications and handles all of the tricky stuff under the hood. This recipe will give a simple example of an Ecore client and an Ecore server.

When working with Ecore_Ipc, when writing a client or a server app an Ecore_Ipc_Server object will be created. This is because in either case it is a server being manipulated, either the one being setup, or the one being communicated with. After that, everything is easy.

Example 4.15. Ecore_Ipc client: preamble

```
#include <Ecore.h>
#include <Ecore_Ipc.h>

int sig_exit_cb(void *data, int ev_type, void *ev);
int handler_server_add(void *data, int ev_type, void *ev);
int handler_server_del(void *data, int ev_type, void *ev);
int handler_server_data(void *data, int ev_type, void *ev);
```

The Ecore.h file is included so we can have access to the exit signal type. The functions will be explained later when their callbacks are hooked up.

Example 4.16. Ecore_Ipc client: main setup

```
int main(int argc, char ** argv) {
    Ecore_Ipc_Server *server;

    if (!ecore_init()) {
        printf("unable to init ecore\n");
        return 1;
    }

    if (!ecore_ipc_init()) {
        printf("unable to init ecore_con\n");
        ecore_shutdown();
        return 1;
    }
    ecore_event_handler_add(ECORE_EVENT_SIGNAL_EXIT, sig_exit_cb, NULL);
```

As mentioned earlier, even though we are writing a client app, we still use an Ecore_Ipc_Server object. Using Ecore_Ipc requires the setup of Ecore itself. This is done with a simple call to `ecore_init`. Ecore_Ipc is then setup with a call to `ecore_ipc_init`. If either of these return 0, the appropriate action is taken to undo any initialization take to this point. The ECORE_EVENT_SIGNAL_EXIT callback is hooked up so we can exit gracefully if required.

Example 4.17. Ecore_Ipc client: main creating client

```
    server = ecore_ipc_server_connect(ECORE_IPC_REMOTE_SYSTEM,
                                      "localhost", 9999, NULL);
    ecore_event_handler_add(ECORE_IPC_EVENT_SERVER_ADD,
                                      handler_server_add, NULL);
    ecore_event_handler_add(ECORE_IPC_EVENT_SERVER_DEL,
                                      handler_server_del, NULL);
    ecore_event_handler_add(ECORE_IPC_EVENT_SERVER_DATA,
                                      handler_server_data, NULL);
```

In this example we are creating a remote connection to the server named "localhost" on the port 9999. This is done with the `ecore_ipc_server_connect` method. The first parameter is the type of connection being made, which can be one of: ECORE_IPC_REMOTE_SYSTEM, ECORE_IPC_LOCAL_SYSTEM, or ECORE_IPC_LOCAL_USER. If OpenSSL was available when Ecore_Ipc was compiled, ECORE_IPC_USE_SSL can be or'd with the connection type to create an SSL connection.

The three calls to `ecore_event_handler_add` setup the callbacks for the different types of events we will be receiving from the server. A server was added, a server was deleted, or the server sent us data.

Example 4.18. Ecore_Ipc client: main end

```
    ecore_ipc_server_send(server, 3, 4, 0, 0, 0, "Look ma, no pants", 17);

    ecore_main_loop_begin();

    ecore_ipc_server_del(server);
    ecore_ipc_shutdown();
    ecore_shutdown();
    return 0;
}
```

For the purposes of this example, the client is sending a message on startup to the server, which the server will respond to. The client message is sent with the `ecore_ipc_server_send` command. `ecore_ipc_server_send` takes the server to send to, the message major, message minor, a reference, a reference to, a response, the data and a size. These parameters, except for the server are up the the client and can refer to anything required. This hopefully gives the maximum flexibility in creating client/server IPC apps.

After the server message is sent we enter into the main ecore loop and wait for events. If the main loop is exited we delete the server object, shutdown Ecore_Ipc with a call to `ecore_ipc_shutdown`, and shutdown ecore through `ecore_shutdown`.

Example 4.19. Ecore_Ipc client: sig_exit_cb

```
int sig_exit_cb(void *data, int ev_type, void *ev) {
    ecore_main_loop_quit();
    return 1;
}
```

The `sig_exit_cb` just tells ecore to quit the main loop. This isn't strictly necessary because if the only thing being done is calling `ecore_main_loop_quit()`, Ecore will handing this itself if there is not handler setup. But this shows how a handler can be created if one is needed for your application.

Example 4.20. Ecore_Ipc client: the callbacks

```
int handler_server_add(void *data, int ev_type, void *ev) {
```

```
    Ecore_Ipc_Event_Server_Add *e = (Ecore_Ipc_Event_Server_Add *)ev;
    printf("Got a server add %p\n", e->server);
    return 1;
}

int handler_server_del(void *data, int ev_type, void *ev) {
    Ecore_Ipc_Event_Server_Del *e = (Ecore_Ipc_Event_Server_Del *)ev;
    printf("Got a server del %p\n", e->server);
    return 1;
}

int handler_server_data(void *data, int ev_type, void *ev) {
    Ecore_Ipc_Event_Server_Data *e = (Ecore_Ipc_Event_Server_Data *)ev;
    printf("Got server data %p [%i] [%i] [%i] (%s)\n", e->server, e->major,
                                e->minor, e->size, (char *)e->data);
    return 1;
}
```

These three callbacks, `handler_server_add`, `handler_server_del`, and `handler_server_data` are body of the client handling all events related to the server we are connected to. Each of the callbacks has an associated event structure, Ecore_Ipc_Event_Server_Add, Ecore_Ipc_Event_Server_Del and Ecore_Ipc_Event_Server_Data containing information on the event itself.

When we first connect to the server the `handler_server_add` callback will be executed allowing any setup to be accomplished.

If the server breaks the connection the `handler_server_del` callback will be executed allowing any required cleanup.

When the server sends data to the client the `handler_server_data` callback will the executed. Which in this example just prints some information about the message itself and the message body.

And thats the client. The code itself is pretty simple thanks to the abstractions provided by Ecore.

Example 4.21. Ecore_Ipc server: preamble

```
#include <Ecore.h>
#include <Ecore_Ipc.h>

int sig_exit_cb(void *data, int ev_type, void *ev);
int handler_client_add(void *data, int ev_type, void *ev);
int handler_client_del(void *data, int ev_type, void *ev);
int handler_client_data(void *data, int ev_type, void *ev);
```

As with the client, the Ecore.h header is included to get access the to the exit signal. The Ecore_Ipc.h header is required for apps making use of the Ecore_Ipc library. Each sig handler will be explained with its code.

Example 4.22. Ecore_Ipc server: main setup

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

```
    Ecore_Ipc_Server *server;

    if (!ecore_init()) {
        printf("Failed to init ecore\n");
        return 1;
    }

    if (!ecore_ipc_init()) {
        printf("failed to init ecore_con\n");
        ecore_shutdown();
        return 1;
    }

    ecore_event_handler_add(ECORE_EVENT_SIGNAL_EXIT, sig_exit_cb, NULL);
```

This is the same as the client setup above.

Example 4.23. Ecore_Ipc server: main creating server

```
    server = ecore_ipc_server_add(ECORE_IPC_REMOTE_SYSTEM, "localhost", 9999, NULL
    ecore_event_handler_add(ECORE_IPC_EVENT_CLIENT_ADD, handler_client_add, NULL);
    ecore_event_handler_add(ECORE_IPC_EVENT_CLIENT_DEL, handler_client_del, NULL);
    ecore_event_handler_add(ECORE_IPC_EVENT_CLIENT_DATA, handler_client_data, NULL
```

Unlike the client, for the server we add a listener to port 9999 on the machine "localhost" through the call `ecore_ipc_server_add`. This will create and return the server object to us. We then hook in the required signal handlers, the difference to the client being we want CLIENT events this time instead of SERVER events.

Example 4.24. Ecore_Ipc client: main end

```
    ecore_main_loop_begin();

    ecore_ipc_server_del(server);
    ecore_ipc_shutdown();
    ecore_shutdown();
    return 0;
}
```

This again is identical to the client shutdown, minus the sending of data to the server.

Example 4.25. Ecore_Ipc server: sig_exit callback

The `sig_exit_cb` is again identical to that seen in the client.

Example 4.26. Ecore_Ipc server: the callbacks

```
int handler_client_add(void *data, int ev_type, void *ev) {
    Ecore_Ipc_Event_Client_Add *e = (Ecore_Ipc_Event_Client_Add *)ev;
    printf("client %p connected to server\n", e->client);
    return 1;
}

int handler_client_del(void *data, int ev_type, void *ev) {
    Ecore_Ipc_Event_Client_Del *e = (Ecore_Ipc_Event_Client_Del *)ev;
    printf("client %p disconnected from server\n", e->client);
    return 1;
}

int handler_client_data(void *data, int ev_type, void *ev) {
    Ecore_Ipc_Event_Client_Data *e = (Ecore_Ipc_Event_Client_Data *)ev;
    printf("client %p sent [%i] [%i] [%i] (%s)\n", e->client, e->major,
                                  e->minor, e->size, (char *)e->data);

    ecore_ipc_client_send(e->client, 3, 4, 0, 0, 0, "Pants On!", 9);
    return 1;
}
```

The event callbacks are similar to those seen in the client app. The main difference is that the events are _Client_ events instead of _Server_ events.

The add callback is when a client connects to our server, with the del callback being its opposite when the client disconnects. The data callback is for when a client sends data to the server.

At the end of the `handler_client_data` callback we do a call to `ecore_ipc_client_send`. This sends data to the client. As with sending data to the server, the parameters are: the client to send to, major number, minor number, reference, reference to, response, data and the data size.

Example 4.27. Ecore_Ipc: compilation

```
CC = gcc

all: server client

server: server.c
    $(CC) -o server server.c `ecore-config --cflags --libs`

client: client.c
    $(CC) -o client client.c `ecore-config --cflags --libs`

clean:
    rm server client
```

As with other ecore apps, it is very easy to compile an Ecore_Ipc app. As long as your Ecore was compiled with Ecore_Ipc, simply invoking the 'ecore-config --cflags --libs' command will add all of the required library paths and linker information.

As seen in this example, Ecore_Ipc is an easy to use library to create client/server apps.

Recipe: Ecore Timers

dan 'dj2' sinclair <zero@perplexity.org>

If you need to have a callback triggered at a specific time, with the possibility of repeating the timer continuously, then the `Ecore_Timer` is what you are looking for.

Example 4.28. Ecore Timers

```
#include <stdio.h>
#include <Ecore.h>

static int timer_one_cb(void *data);
static int timer_two_cb(void *data);

int main(int argc, char ** argv) {
    ecore_init();

    ecore_timer_add(1, timer_one_cb, NULL);
    ecore_timer_add(0.5, timer_two_cb, NULL);

    ecore_main_loop_begin();
    ecore_shutdown();

    return 0;
}

static int timer_one_cb(void *data) {
    printf("1");
    fflush(stdout);
    return 1;
}

static int timer_two_cb(void *data) {
    printf("2");
    fflush(stdout);
    return 1;
}
```

The creation of the timers is as simple as calling `ecore_timer_add()`. This will return an `Ecore_Timer` strut on success or NULL on failure. In this case I'm ignoring the return value. The three parameters are:

- double timeout
- int (*callback)(void *data)
- const void *user_data

The timeout gives the number of seconds in which this timer will expire. In the case of this example we give it 1 second and 0.5 seconds respectively. The callback function is the one that will be executed when the timer expires and the user_data is any data to be passed to the callback function.

The callback functions all have the same signature `int callback(void *data)`. The return value of the timer should be either 0 or 1. If you return 0 the timer will expire and will *not* be run again. If you

return 1, the timer will be rescheduled to re-execute in the amount of time given by the timeout. This allows you to activate or continue the timer as required by your program.

If you have a timer that you wish to remove at some point in the future you can call `ecore_timer_del(Ecore_Timer *)`. If this delete succeeds the pointer will be returned otherwise NULL will be returned. After calling the delete function the `Ecore_Timer` structure will be invalid and you should not use it again in your program.

Compiling the example is as simple as:

Example 4.29. Compilation

```
gcc -Wall -o etimer etimer.c `ecore-config --cflags --libs`
```

If you run the program you should see a series of '1's and '2's on the screen with twice as many '2's as '1's.

The `Ecore_Timer`s are easy to setup and use and provide a powerful timing mechanism to your programs.

Recipe: Adding Ecore Events

dan 'dj2' sinclair <zero@perplexity.org>

If you ever need to make your own events you can easily hook them into Ecores event system. This gives you the ability to add events into the event queue for your event and have Ecore deliever them to another part of the app.

The following program creates a simple event and a timer. When the timer is triggered it will add our new event into the event queue. Our event will then print a message and quit the application.

Example 4.30. Ecore event example

```
#include <stdio.h>
#include <Ecore.h>

static int timer_cb(void *data);
static int event_cb(void *data, int type, void *ev);
static void event_free(void *data, void *ev);

int MY_EVENT_TYPE = 0;

typedef struct Event_Struct Event_Struct;
struct Event_Struct {
    char *name;
};

int
main(int argc, char ** argv)
{
    ecore_init();

    MY_EVENT_TYPE = ecore_event_type_new();
    ecore_event_handler_add(MY_EVENT_TYPE, event_cb, NULL);
    ecore_timer_add(1, timer_cb, NULL);
```

```
    ecore_main_loop_begin();
    ecore_shutdown();
    return 0;
}

static int
timer_cb(void *data)
{
    Event_Struct *e;
    Ecore_Event *event;

    e = malloc(sizeof(Event_Struct));
    e->name = strdup("ned");

    event = ecore_event_add(MY_EVENT_TYPE, e, event_free, NULL);
    return 0;
}

static int
event_cb(void *data, int type, void *ev)
{
    Event_Struct *e;

    e = ev;
    printf("Got event %s\n", e->name);
    ecore_main_loop_quit();
    return 1;
}

static void
event_free(void *data, void *ev)
{
    Event_Struct *e;

    e = ev;
    free(e->name);
    free(e);
}
```

Each event has an associated event id. This id is a simple an int value that is assigned through the call to `ecore_event_type_new()`. Once we have the event id we can use it in `ecore_event_handler_add()` calls. Thats all there is to creating the event.

Ecore gives us the ability to pass an event structure to our event. Just note that you have to be careful, if you don't specify a free function for the struct ecore will use a generic function which just calls `free` on the value. So, don't pass anything you need into there without care. (Or prepare to send an evening tracking down very strange segv's in your program)

In this example we create a simple `Event_Struct` that is passed. The call to actually create the event is `ecore_event_add()`. This takes the event id, any event data, the event data free function, and any data to pass to the free function.

So as you can see we pass our `Event_Struct` as the event data and set the `event_free` function as the cleanup for this structure.

And thats it. You can compile as below and everything should work.

Example 4.31.

```
zero@oberon [ecore_event] -> gcc -o ev main.c `ecore-config --cflags --libs`
```

As this shows, it is really easy to extend Ecore with your own events. The system has been setup to allow it to be extened as required.

Chapter 5. EDB & EET

EDB is a database convenience library wrapped around the Berkeley DB 2.7.7 by Sleepycat Software. It is intended to make accessing database information portable, easy, fast and efficient.

EET is a tiny library designed to write arbitary chunks of data to a file and optionally compress each chunk (very much like a zip file) and allows for fast random-access reading of the file later on. It does not do zip as a zip itself has more complexity than is needed, and it was much simpler to impliment this once here.

EDB provides an excellent method of storing and retrieving application configuration information, although it can be used for more extensively than that. Ebits, the predecessor to Edje, even used EDB as a container for Ebits themes prior to EET. An Edb consists of a series of key/value pairs, which can consist of a variety of data types, including integers, floating point values, strings, and binary data. The simplified API provides simple, complete, and unified functions for managing and accessing your database.

In addition to the library, a variety of tools are available to access and modify your EDBs. The edb_ed tool provides a simple command line interface that can easily be scripted, especially useful for use with the GNU autotools suite. The edb_vt_ed tool provides an easy to use curses interface. Finally, edb_gtk_ed provides an elegant and easy GUI interface, especially useful for end user editing of configuration data contained in EDBs.

Eet is extremely fast, small and simple. Eet files can be very small and highly compressed, making them very optimal for just sending across the internet without having to archive, compress or decompress and install them. They allow for lightning-fast random-acess reads once created, making them perfect for storing data that is written once (or rarely) and read many times, but the program does not want to have to read it all in at once.

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 arbitary files on the system. All data is encoded in a platform independant way and can be written and read by any architecture.

Recipe: Creating EDB files from the shell

dan 'dj2' sinclair <zero@perplexity.org>

It is often desired to create the EDB files from a simple shell script, it can then be made part of the build process.

This can easily be accomplished by using the **edb_ed** program. **edb_ed** is a very simple interface into EDB, allowing you to create/edit/delete keys/value pairs inside of EDB databases.

Example 5.1. EDB file shell script

```
#!/bin/sh

DB=out.db

edb_ed $DB add /global/debug_lvl int 2
edb_ed $DB add /foo/theme str "default"
edb_ed $DB add /bar/number_of_accounts int 1
edb_ed $DB add /nan/another float 2.3
```

If the output file does not exist the first time an `add` command is called, then **edb_ed** will create the file and do any required setup. The `add` is used to add entries into the DB. The first parameter after add is the key that the data will be inserted into the DB with. This key will be used to look up the data by your application in the future. The next parameter is the type of data to be added. This can be one of:

- int
- str
- float
- data

The last parameter is the value that is to be associated with this key.

Using **edb_ed** you can quickly and easily create/edit/view any EDB files required for your application.

Recipe: EDB introduction

dan 'dj2' sinclair `<zero@perplexity.org>`

EDB provides a powerful database backend for use in your application. This recipe is a simple introduction that will open a database, write several keys and then read them back out.

Example 5.2. EDB introduction

```
#include <stdio.h>
#include <Edb.h>

#define INT_KEY     "/int/val"
#define STR_KEY     "/str/val"
#define FLT_KEY     "/float/val"

int main(int argc, char ** argv) {
    E_DB_File *db_file = NULL;
    char *str;
    int i;
    float f;

    if (argc < 2) {
        printf("Need db file\n");
        return 0;
    }

    db_file = e_db_open(argv[1]);
    if (db_file == NULL) {
        printf("Error opening db file (%s)\n", argv[1]);
        return 0;
    }

    printf("Adding values...\n");
    e_db_int_set(db_file, INT_KEY, 42);
    e_db_float_set(db_file, FLT_KEY, 3.14159);
    e_db_str_set(db_file, STR_KEY, "My cats breath smells like...");

    printf("Reading values...\n");
    if (e_db_int_get(db_file, INT_KEY, &i))
        printf("Retrieved (%s) with value (%d)\n", INT_KEY, i);

    if (e_db_float_get(db_file, FLT_KEY, &f))
        printf("Retrieved (%s) with value (%f)\n", FLT_KEY, f);
```

```
    if ((str = e_db_str_get(db_file, STR_KEY)) != NULL) {
        printf("Retrieved (%s) with value (%s)\n", STR_KEY, str);
        free(str);
    }

    e_db_close(db_file);
    e_db_flush();

    return 1;
}
```

In order to use EDB you must include <Edb.h> in your file to have access to the API. The initial portions of the program are pretty standard, I have a tendency to make typing mistakes so I defined the different keys that I will be using. As long as we have a file name we try to open/create the database.

The database will be opened, or if it doesn't exist, created with the call to `e_db_open()` which will return NULL if an error was encountered.

Once the database is open we can write our values. This is done through the three calls: `e_db_int_set()`, `e_db_float_set()` and `e_db_str_set()`. You can also set generic data into a db file with `e_db_data_set()`.

Along with normal data, you can store meta-data about the db into the file itself. This data can not be retrieved with the normal get/set methods. These properties are set with `e_db_property_set()`

Each of the type setting methods takes three parameters:

- E_DB_File *db
- char *key
- value

The `value` parameter is of the corresponding type to the method, int, float, char * or void * for _int_set, _float_set, _str_set and _data_set respectively.

Once the values are in the db they can be retrieved with the getter methods. Each of these methods takes 3 parameters and returns an int. The return value is 1 on successful retrieval and 0 otherwise.

As with the setter methods, the getter methods parameters are the db, key and a pointer to place to put the value.

Once we're finished with the database we can close it with a call to `e_db_close()`. The call to `e_db_close()` does not guarantee that the database has been written to disk, to do this we call `e_db_flush()` which will write all databases are not being used and writes the contents out to disk.

Example 5.3. Compiling

```
zero@oberon [edb] -> gcc -o edb edb_ex.c \
        `edb-config --cflags --libs`
```

If you execute the program you should see the values written out to the screen, and after execution there will be a .db file with the name you specified. You can then take a look at the .db file with **edb_gtk_ed** and see the values entered.

Recipe: EDB key retrieval

dan 'dj2' sinclair <zero@perplexity.org>

The EDB API makes it a simple task to retrieve all of the available keys into the database. These keys can then be used to determine the types of the object in the database, or to just retrieve the object if required.

Example 5.4. EDB key retrieval

```
#include <Edb.h>

int main(int argc, char ** argv) {
    char ** keys;
    int num_keys, i;

    if (argc < 2)
        return 0;

    keys = e_db_dump_key_list(argv[1], &num_keys);
    for(i = 0; i < num_keys; i++) {
        printf("key: %s\n", keys[i]);
        free(keys[i]);
    }
    free(keys);
    return 1;
}
```

Retrieving the keys is done simply through the call to `e_db_dump_key_list()`. This will return a char ** array of key strings. These strings, and the array itself, must be freed by the application. `e_db_dump_key_list()` will also return the number of keys in the array in the `num_keys` parameter.

You'll notice we do not need to open the db in order to call the `e_db_dump_key_list()`. This function works on the file itself instead of a db object.

Example 5.5. Compiling

```
zero@oberon [edb] -> gcc -o edb edb_ex.c \
        `edb-config --cflags --libs`
```

Executing the program should produce a listing of all the keys in the given database. This can be verified by viewing the db through an external tool like, **edb_gtk_ed**.

Chapter 6. Esmart

Esmart provides a variety of EVAS smart objects that provide significant power to your EVAS and EFL based applications.

Recipe: Esmart Trans Introduction

dan 'dj2' sinclair <zero@perplexity.org>

Transparency is increasingly becoming a common trait of applications. To this end, the Esmart_Trans object has been created. This object will do all of the hard work to produce a transparent background for your program.

Esmart trans makes the integration of a transparent background into your application very easy. You need to create the trans object, and then make sure you update it as the window is moved or resized.

Example 6.1. Includes and declarations

```
#include <stdio.h>
#include <Ecore.h>
#include <Ecore_Evas.h>
#include <Esmart/Esmart_Trans_X11.h>

int sig_exit_cb(void *data, int ev_type, void *ev);
void win_del_cb(Ecore_Evas *ee);
void win_resize_cb(Ecore_Evas *ee);
void win_move_cb(Ecore_Evas *ee);

static void _freshen_trans(Ecore_Evas *ee);
void make_gui();
```

Every application that uses an Esmart_Trans object is going to require the Ecore, Ecore_Evas and the Esmart/Esmart_Trans header files. The next four declarations are callbacks from ecore for events on our window, exit, delete, resize, and move respectively. The last two declarations are convenience functions being used in the example and do not need to be in your program.

Example 6.2. main

```
int main(int argc, char ** argv) {
    int ret = 0;

    if (!ecore_init()) {
        printf("Error initializing ecore\n");
        ret = 1;
        goto ECORE_SHUTDOWN;
    }

    if (!ecore_evas_init()) {
        printf("Error initializing ecore_evas\n");
        ret = 1;
        goto ECORE_SHUTDOWN;
    }
```

```
    make_gui();
    ecore_main_loop_begin();

    ecore_evas_shutdown();

ECORE_SHUTDOWN:
    ecore_shutdown();

    return ret;
}
```

The main routine for this example is pretty simple. Ecore and Ecore_Evas are both initialized, with appropriate error checking. We then create the gui and start the main ecore event loop. When ecore exits we shut everything down and return.

Example 6.3. exit and del callbacks

```
int sig_exit_cb(void *data, int ev_type, void *ev) {
    ecore_main_loop_quit();
    return 1;
}

void win_del_cb(Ecore_Evas *ee) {
    ecore_main_loop_quit();
}
```

The exit and del callbacks are the generic ecore callbacks. The exit callback isn't strictly necessary, as Ecore will call `ecore_main_loop_quit()` if no handler is registered, but is included to show how its done.

Example 6.4. _freshen_trans

```
static void _freshen_trans(Ecore_Evas *ee) {
    int x, y, w, h;
    Evas_Object *o;

    if (!ee) return;

    ecore_evas_geometry_get(ee, &x, &y, &w, &h);
    o = evas_object_name_find(ecore_evas_get(ee), "bg");

    if (!o) {
        fprintf(stderr, "Trans object not found, bad, very bad\n");
        ecore_main_loop_quit();
    }
    esmart_trans_x11_freshen(o, x, y, w, h);
}
```

The `_freshen_trans` routine is a helper routine to update the image that the trans is shown. This will be called when we need to update our image to whats currently under the window. The function

grabs the current size of the ecore_evas, and then gets the object with the name "bg" (this name is the same as the name we give our trans when we create it). Then, as long as the trans object exists, we tell esmart to freshen the image being displayed.

Example 6.5. resize_cb

```
void win_resize_cb(Ecore_Evas *ee) {
    int w, h;
    int minw, minh;
    int maxw, maxh;
    Evas_Object *o = NULL;

    if (ee) {
        ecore_evas_geometry_get(ee, NULL, NULL, &w, &h);
        ecore_evas_size_min_get(ee, &minw, &minh);
        ecore_evas_size_max_get(ee, &maxw, &maxh);

        if ((w >= minw) && (h >= minh) && (h <= maxh) && (w <= maxw)) {
            if ((o = evas_object_name_find(ecore_evas_get(ee), "bg")))
                evas_object_resize(o, w, h);
        }
    }
    _freshen_trans(ee);
}
```

When the window is resized we need to update our evas to the correct size and then update the trans object to display that much of the background. We grab the current size of the window `ecore_evas_geometry_get` and the min/max size of the window. As long as our currently desired size is within the min/max bounds set for our window, we grab the "bg" (same as title again) object and resize it. Once the resizing is done, we call the `_freshen_trans` routine to update the image displayed on the bg.

Example 6.6. move_cb

```
void win_move_cb(Ecore_Evas *ee) {
    _freshen_trans(ee);
}
```

When the window is moved we need to freshen the image displayed as the transparency.

Example 6.7. Setup ecore/ecore_evas

```
void make_gui() {
    Evas *evas = NULL;
    Ecore_Evas *ee = NULL;
    Evas_Object *trans = NULL;
    int x, y, w, h;

    ecore_event_handler_add(ECORE_EVENT_SIGNAL_EXIT, sig_exit_cb, NULL);
```

```
    ee = ecore_evas_software_x11_new(NULL, 0, 0, 0, 300, 200);
    ecore_evas_title_set(ee, "trans demo");

    ecore_evas_callback_delete_request_set(ee, win_del_cb);
    ecore_evas_callback_resize_set(ee, win_resize_cb);
    ecore_evas_callback_move_set(ee, win_move_cb);

    evas = ecore_evas_get(ee);
```

The first portion of make_gui is concerned with setting up ecore and ecore_evas. First the exit callback is hooked into ECORE_EVENT_SIGNAL_EXIT, then the Ecore_Evas object is created with the software X11 engine. The window title is set and we hook in the callbacks written above, delete, resize and move. Finally we grab the evas for the created Ecore_Evas.

Example 6.8. Creating Esmart_Trans object

```
    trans = esmart_trans_x11_new(evas);
    evas_object_move(trans, 0, 0);
    evas_object_layer_set(trans, -5);
    evas_object_name_set(trans, "bg");

    ecore_evas_geometry_get(ee, &x, &y, &w, &h);
    evas_object_resize(trans, w, h);

    evas_object_show(trans);
    ecore_evas_show(ee);

    esmart_trans_x11_freshen(trans, x, y, w, h);
}
```

Once everything is setup we can create the trans object. The trans is to be created in the evas returned by `ecore_evas_get`. This initial creation is done by the call to `esmart_trans_x11_new`. Once we have the object, we move it so it starts at position (0, 0) (the upper left corner), set the layer to -5 and name the object "bg" (as used above). Then we grab the current size of the ecore_evas and use that to resize the trans object to the window size. Once everything is resized we show the trans and show the ecore_evas. As a final step, we freshen the image on the transparency to what is currently under the window so it is up to date.

Example 6.9. Simple makefile

```
CFLAGS = `ecore-config --cflags` `evas-config --cflags` `esmart-config --cflags`
LIBS = `ecore-config --libs` `evas-config --libs` `esmart-config --libs` \
            -lesmart_trans_x11

all:
    gcc -o trans_example trans_example.c $(CFLAGS) $(LIBS)
```

In order to compile the above program we need to include the library information for ecore, ecore_evas and esmart. This is done through the -config scripts for each library. These -config scripts know where each of the includes and libraries resides and sets up the appropriate linking and include paths for the

compilation.

Recipe: Esmart Container Introduction

dan 'dj2' sinclair <zero@perplexity.org>

There is usually a desire while designing an apps UI to group common elements together and have their layout depend on one another. To this end the Esmart Container library has been created. It has been designed to handle the hard parts of the layout, and in the cases where it does not do what you need, the layout portions of the container are extensible and changeable.

This recipe will give the basics of using an Esmart container. The final product is a program that will let you see some of the different layout combinations of the default container. The layout will be done by Edje with callbacks to the program to change the container layout, and to tell if the user clicked on a container element.

Example 6.10. Includes and declarations

```
#include <Ecore.h>
#include <Ecore_Evas.h>
#include <Edje.h>
#include <Esmart/Esmart_Container.h>
#include <getopt.h>

static void make_gui(const char *theme);
static void container_build(int align, int direction, int fill);
static void _set_text(int align, int direction);
static void _setup_edje_callbacks(Evas_Object *o);
static void _right_click_cb(void* data, Evas_Object* o, const char* emmission,
                                                        const char* source);
static void _left_click_cb(void* data, Evas_Object* o, const char* emmission,
                                                        const char* source);
static void _item_selected(void* data, Evas_Object* o, const char* emmission,
                                                        const char* source);

static Ecore_Evas *ee;
static Evas_Object *edje;
static Evas_Object *container;

char *str_list[] = {"item 1", "item 2",
                    "item 3", "item 4",
                    "item 5"};
```

As with other EFL programs we need to include Ecore, Ecore_Evas, Edje and as this is a container example, the Esmart/Esmart_Container header. Getopt will be used to allow for some command line processing.

Next comes the function prototypes which will be described later when we get to their implementations. Then a few global variables to be used throughout the program. The str_list array is the content to be stored in the container.

Example 6.11. main

```
int main(int argc, char ** argv) {
    int align = 0;
    int direction = 0;
    int fill = 0;
    int ret = 0;
    int c;
    char *theme = NULL;

    while((c = getopt(argc, argv, "a:d:f:t:")) != -1) {
        switch(c) {
            case 'a':
                align = atoi(optarg);
                break;

            case 'd':
                direction = atoi(optarg);
                break;

            case 'f':
                fill = atoi(optarg);
                break;

            case 't':
                theme = strdup(optarg);
                break;

            default:
                printf("Unknown option string\n");
                break;
        }
    }

    if (theme == NULL) {
        printf("Need a theme defined\n");
        exit(-1);
    }
```

The beginning of the main function gets the options out of the command line arguments and sets up the default display. As you can see, we require a theme to display. This could be made more intelligent, searching default install directories and the users application directories, but this example takes the easy way out and forces the theme to be a command line option.

Example 6.12. Initialization

```
    if (!ecore_init()) {
        printf("Can't init ecore, bad\n");
        ret = 1;
        goto EXIT;
    }
    ecore_app_args_set(argc, (const char **)argv);

    if (!ecore_evas_init()) {
        printf("Can't init ecore_evas, bad\n");
        ret = 1;
        goto EXIT_ECORE;
    }

    if (!edje_init()) {
```

```
        printf("Can't init edje, bad\n");
        ret = 1;
        goto EXIT_ECORE_EVAS;
    }
    edje_frametime_set(1.0 / 60.0);

    make_gui(theme);
    container_build(align, direction, fill);

    ecore_main_loop_begin();
```

After receiving the command line arguments, we then proceed to initializing the required libraries, Ecore, Ecore_Evas and Edje. We take the additional step of setting the Edje frame rate.

Once the initialization is complete we create the initial GUI for the app. I have separated the building of the container out into a separate function to make the container code easier to locate in the example.

Once everything is created we call `ecore_main_loop_begin` and wait for events to occur.

Example 6.13. Shutdown

```
    edje_shutdown();

EXIT_ECORE_EVAS:
    ecore_evas_shutdown();

EXIT_ECORE:
    ecore_shutdown();

EXIT:
    return ret;
}
```

The usual end routine, be good programmers and shutdown everything we started.

Example 6.14. Window callbacks

```
static int sig_exit_cb(void *data, int ev_type, void *ev) {
    ecore_main_loop_quit();
    return 1;
}

static void win_del_cb(Ecore_Evas *ee) {
    ecore_main_loop_quit();
}

static void win_resize_cb(Ecore_Evas *ee) {
    int w, h;
    int minw, minh;
    int maxw, maxh;
    Evas_Object *o = NULL;

    if (ee) {
```

```
        ecore_evas_geometry_get(ee, NULL, NULL, &w, &h);
        ecore_evas_size_min_get(ee, &minw, &minh);
        ecore_evas_size_max_get(ee, &maxw, &maxh);

        if ((w >= minw) && (h >= minh) && (h <= maxh) && (w <= maxw)) {
            if ((o = evas_object_name_find(ecore_evas_get(ee), "edje")))
                evas_object_resize(o, w, h);
        }
    }
}
```

Next we setup some generic callbacks to be used by the UI. This will be the exit, destroy and resize callbacks. Again, the usual EFL style functions. Although the exit callback is not strictly necessary as Ecore itself will call `ecore_main_loop_quit()` if no handler is registered for this callback.

Example 6.15. make_gui

```
static void make_gui(const char *theme) {
    Evas *evas = NULL;
    Evas_Object *o = NULL;
    Evas_Coord minw, minh;

    ee = NULL;
    edje = NULL;
    container = NULL;

    ecore_event_handler_add(ECORE_EVENT_SIGNAL_EXIT, sig_exit_cb, NULL);

    ee = ecore_evas_software_x11_new(NULL, 0, 0, 0, 300, 400);
    ecore_evas_title_set(ee, "Container Example");

    ecore_evas_callback_delete_request_set(ee, win_del_cb);
    ecore_evas_callback_resize_set(ee, win_resize_cb);
    evas = ecore_evas_get(ee);

    // create the edje
    edje = edje_object_add(evas);
    evas_object_move(edje, 0, 0);

    if (edje_object_file_set(edje, theme, "container_ex")) {
        evas_object_name_set(edje, "edje");

        edje_object_size_min_get(edje, &minw, &minh);
        ecore_evas_size_min_set(ee, (int)minw, (int)minh);
        evas_object_resize(edje, (int)minw, (int)minh);
        ecore_evas_resize(ee, (int)minw, (int)minh);

        edje_object_size_max_get(edje, &minw, &minh);
        ecore_evas_size_max_set(ee, (int)minw, (int)minh);
        evas_object_show(edje);

    } else {
        printf("Unable to open (%s) for edje theme\n", theme);
        exit(-1);
    }
    _setup_edje_callbacks(edje);
    ecore_evas_show(ee);
}
```

The GUI consists of the Ecore_Evas containing the canvas itself, and the Edje that we will be using to control our layout. The `make_gui` function sets up the callbacks defined above and creates the Ecore_Evas.

Once we have the Evas and the callbacks are defined, we create the Edje object that will define our layout. The `edje_object_add` call is used to create the object on the Evas, and once thats done, we take the theme passed in by the user and set our Edje to use said theme, the "container_ex" parameter is the name of the group inside the EET that we are to use.

Once the theme file is set to the Edje, we use the values in the theme file to setup the size ranges for the app, and show the Edje. We then setup the callbacks on the Edje and show the Ecore_Evas.

Example 6.16. Edje Callbacks

```
static void _setup_edje_callbacks(Evas_Object *o) {
    edje_object_signal_callback_add(o, "left_click",
                        "left_click", _left_click_cb, NULL);
    edje_object_signal_callback_add(o, "right_click",
                        "right_click", _right_click_cb, NULL);
}
```

The program will have two main callbacks attached to the Edje, one for the left click signal and one for the right click signal. These will be used to switch the direction/alignment of the container. The second and third parameters of the callbacks need to match the data emitted with the signal from Edje, this will be seen later when we look at the EDC file. The third parameter is the function to call, and the last, any data we wish to be passed into the callback.

Example 6.17. container_build

```
static void container_build(int align, int direction, int fill_policy) {
    int len = 0;
    int i = 0;
    const char *edjefile = NULL;

    container = esmart_container_new(ecore_evas_get(ee));
    evas_object_name_set(container, "the_container");
    esmart_container_direction_set(container, direction);
    esmart_container_alignment_set(container, align);
    esmart_container_padding_set(container, 1, 1, 1, 1);
    esmart_container_spacing_set(container, 1);
    esmart_container_fill_policy_set(container, fill_policy);

    evas_object_layer_set(container, 0);
    edje_object_part_swallow(edje, "container", container);
```

The `container_build` function will create the container and set our data elements in said container. The creation is easy enough with a call to `esmart_container_new` giving back the Evas_Object that is the container. Once the container is created we can set a name on the container to make reference

easier.

Next, we set the direction, which is either (CONTAINER_DIRECTION_VERTICAL or CONTAINER_DIRECTION_HORIZONTAL) [or in this case, an int being passed from the command line as the two directions map to 1 and 0 respectively]. The direction tells the container which way the elements will be drawn.

After the direction we set the alignment of the container. The alignment tells the container where to draw the elements. The possible values are: CONTAINER_ALIGN_CENTER, CONTAINER_ALIGN_LEFT, CONTAINER_ALIGN_RIGHT, CONTAINER_ALIGN_TOP and CONTAINER_ALIGN_BOTTOM. With the default layout, left and right only apply to a vertical container, and top and bottom only apply to a horizontal container, although center applies to both.

If we wanted to use a different layout scheme then the default, we could place a call to `esmart_container_layout_plugin_set(container, "name")` where the name is the name of the plugin to use. The default setting is the container named "default".

Once the directions and alignment are set, the spacing and padding of the container are specified. The padding specifies the space around the outside of the container taking four numeric parameters: left, right, top and bottom. The spacing parameter specifies the space between elements in the container.

We then continue and set the fill policy of the container. This specifies how the elements are positioned to fill the space in the container. The possible values are: CONTAINER_FILL_POLICY_NONE, CONTAINER_FILL_POLICY_KEEP_ASPECT, CONTAINER_FILL_POLICY_FILL_X, CONTAINER_FILL_POLICY_FILL_Y, CONTAINER_FILL_POLICY_FILL and CONTAINER_FILL_POLICY_HOMOGENOUS.

Once the container is fully specified we set the containers layer, and then swallow the container into the edje and the part named "container".

Example 6.18. Adding Elements to the Container

```
len = (sizeof(str_list) / sizeof(str_list[0]));
for(i = 0; i < len; i++) {
    Evas_Coord w, h;
    Evas_Object *t = edje_object_add(ecore_evas_get(ee));

    edje_object_file_get(edje, &edjefile, NULL);
    if (edje_object_file_set(t, edjefile, "element")) {
        edje_object_size_min_get(t, &w, &h);
        evas_object_resize(t, (int)w, (int)h);

        if (edje_object_part_exists(t, "element.value")) {
            edje_object_part_text_set(t, "element.value", str_list[i]);
            evas_object_show(t);
            int *i_ptr = (int *)malloc(sizeof(int));
            *i_ptr = (i + 1);

            edje_object_signal_callback_add(t, "item_selected",
                                "item_selected", _item_selected, i_ptr);

            esmart_container_element_append(container, t);
        } else {
            printf("Missing element.value part\n");
            evas_object_del(t);
        }
    } else {
        printf("Missing element part\n");
        evas_object_del(t);
```

```
        }
    }
    evas_object_show(container);
    _set_text(align, direction);
}
```

Now that we have a container, we can add some elements to be displayed. Each of the entries in the str_list array defined at the beginning of the program will be added into the container as a text part.

For each element we create a new Edje object on the Evas. We then need to know the name of the theme file used to create our main Edje, so we call `edje_object_file_get` which will set edje file to said value.

We then try to set the group named "element" onto the newly created element. If this fails we print an error and delete the object.

As long as we have found the group "element" we can attempt to grab the part for our element, "element.value". If this part exists, we set the text value of the part to our current string and show the part.

A callback is created through `edje_object_signal_callback_add` and attached to the new element. This will be called if the "item_selected" signal is sent from the Edje. The i_ptr value shows how data can be attached to the element, when the user clicks on an element its number will be printed to the console.

Once the element is created we add it to the container (in this case, appending the element).

To finish, the container is show and we do some extra work to display information about the container in the header through the call `_show_text`.

Example 6.19. _set_text

```
static void _set_text(int align, int direction) {
    Evas_Object *t = edje_object_add(ecore_evas_get(ee));
    const char *edjefile;

    if (direction == CONTAINER_DIRECTION_VERTICAL)
        edje_object_part_text_set(edje, "header_text_direction", "vertical");
    else
        edje_object_part_text_set(edje, "header_text_direction", "horizontal");

    if (align == CONTAINER_ALIGN_CENTER)
        edje_object_part_text_set(edje, "header_text_align", "center");

    else if (align == CONTAINER_ALIGN_TOP)
        edje_object_part_text_set(edje, "header_text_align", "top");

    else if (align == CONTAINER_ALIGN_BOTTOM)
        edje_object_part_text_set(edje, "header_text_align", "bottom");

    else if (align == CONTAINER_ALIGN_RIGHT)
        edje_object_part_text_set(edje, "header_text_align", "right");

    else if (align == CONTAINER_ALIGN_LEFT)
        edje_object_part_text_set(edje, "header_text_align", "left");
}
```

The `_set_text` routine takes the current direction and alignment of the container and sets some text in the header of the program. This is just a simple communication with the user of the current container settings.

Example 6.20. _left_click_cb

```
static void _left_click_cb(void* data, Evas_Object* o, const char* emmission,
                                                        const char* source) {
    Container_Direction dir = esmart_container_direction_get(container);
    Container_Direction new_dir = (dir + 1) % 2;
    Container_Alignment align = esmart_container_alignment_get(container);

    esmart_container_direction_set(container, new_dir);

    if (align != CONTAINER_ALIGN_CENTER) {
        if (new_dir == CONTAINER_DIRECTION_HORIZONTAL)
            align = CONTAINER_ALIGN_TOP;
        else
            align = CONTAINER_ALIGN_LEFT;
    }
    esmart_container_alignment_set(container, align);
    _set_text(align, new_dir);
}
```

When the user clicks the left mouse button on the screen this callback will be executed. We take the current container direction information and switch to the other direction. (e.g. horizontal becomes vertical and visa versa.) We also reset the alignment if we are not currently aligned center to make sure everything is valid for the new direction. The text in the header is updated to be current.

Example 6.21. _right_click_cb

```
static void _right_click_cb(void* data, Evas_Object* o, const char* emmission,
                                                        const char* source) {
    Container_Direction dir = esmart_container_direction_get(container);
    Container_Alignment align = esmart_container_alignment_get(container);

    if (dir == CONTAINER_DIRECTION_HORIZONTAL) {
        if (align == CONTAINER_ALIGN_TOP)
            align = CONTAINER_ALIGN_CENTER;

        else if (align == CONTAINER_ALIGN_CENTER)
            align = CONTAINER_ALIGN_BOTTOM;

        else
            align = CONTAINER_ALIGN_TOP;

    } else {
        if (align == CONTAINER_ALIGN_LEFT)
            align = CONTAINER_ALIGN_CENTER;

        else if (align == CONTAINER_ALIGN_CENTER)
            align = CONTAINER_ALIGN_RIGHT;

        else
            align = CONTAINER_ALIGN_LEFT;
```

```
    }
    esmart_container_alignment_set(container, align);
    _set_text(align, esmart_container_direction_get(container));
}
```

The right click callback will cycle through the available alignments for a given direction as the user clicks the right mouse button.

Example 6.22. _item_selected

```
static void _item_selected(void* data, Evas_Object* o, const char* emmission,
                                                        const char* source) {
    printf("You clicked on the item with number %d\n", *((int *)data));
}
```

Finally the `_item_selected` callback will be executed when the user middle clicks on an item in the container. The data will contain the number set for that element in the create routine above.

Thats the end of the code for the app, next comes the required EDC for everything to display and function correctly.

Example 6.23. The Edc

```
fonts {
    font: "Vera.ttf" "Vera";
}

collections {
    group {
        name, "container_ex";
        min, 300, 300;
        max, 800, 800;

        parts {
            part {
                name, "bg";
                type, RECT;
                mouse_events, 1;

                description {
                    state, "default" 0.0;
                    color, 0 0 0 255;

                    rel1 {
                        relative, 0.0 0.1;
                        offset, 0 0;
                    }
                    rel2 {
                        relative, 1.0 1.0;
                        offset, 0 0;
                    }
                }
            }
```

```
part {
    name, "header";
    type, RECT;
    mouse_events, 0;

    description {
        state, "default" 0.0;
        color, 255 255 255 255;

        rel1 {
            relative, 0.0 0.0;
            offset, 0 0;
        }

        rel2 {
            relative, 1.0 0.1;
            offset, 0 0;
        }
    }
}

part {
    name, "header_text_direction";
    type, TEXT;
    mouse_events, 0;

    description {
        state, "default" 0.0;
        color, 0 0 0 255;

        rel1 {
            relative, 0.0 0.0;
            offset, 0 10;
            to, "header";
        }
        rel2 {
            relative, 1.0 1.0;
            offset, 0 0;
            to, "header";
        }
        text {
            text, "direction";
            font, "Vera";
            size, 10;
        }
    }
}

part {
    name, "header_text_align";
    type, TEXT;
    mouse_events, 0;

    description {
        state, "default" 0.0;
        color, 0 0 0 255;

        rel1 {
            relative, 0.0 0.0;
            offset, 0 0;
            to, "header_text_direction";
        }
        rel2 {
```

```
                    relative, 1.0 1.0;
                    offset, 110 0;
                    to, "header_text_direction";
                }
                text {
                    text, "align";
                    font, "Vera";
                    size, 10;
                }
            }
        }
```

This EDC file expects to have the Vera font embedded inside it, as defined by the font section at the beginning. This means when you compile the edc you either need the Vera.ttf file in the current directory or give edje_cc the -fd flag and specify the directory to the font.

After the fonts are defined the main collections are defined. The first collection is the main portion of the app itself, the "container_ex" group. This group specifies the main window of the app. As such it contains the parts for the background, the header, and the header text. These parts are all fairly standard with some (minimal) alignment done between them.

Example 6.24. The Container Part

```
        part {
            name, "container";
            type, RECT;
            mouse_events, 1;

            description {
                state, "default" 0.0;
                visible, 1;

                rel1 {
                    relative, 0.0 0.0;
                    offset, 0 0;
                    to, bg;
                }
                rel2 {
                    relative, 1.0 1.0;
                    offset, 0 0;
                    to, bg;
                }
                color, 0 0 0 0;
            }
        }
    }
    programs {
        program {
            name, "left_click";
            signal, "mouse,clicked,1";
            source, "container";
            action, SIGNAL_EMIT "left_click" "left_click";
        }

        program {
            name, "right_click";
            signal, "mouse,clicked,3";
            source, "container";
```

```
                action, SIGNAL_EMIT "right_click" "right_click";
            }
        }
    }
```

The container part is then defined. The part itself is pretty simple, just positioned relative to the background and set to receive mouse events. After the parts are defined we specify the programs for this group, of which there are two. The first program "left_click" specifies what is to happen on a click event of the first mouse button.

The action is to emit a signal, the two parameters after SIGNAL_EMIT match up to the values put in the callback in the application code.

There is a similar callback for the third mouse button as the first, just emitting a slightly different signal.

Example 6.25. The Element Group

```
    group {
        name, "element";
        min, 80 18;
        max, 800 18;

        parts {
            part {
                name, "element.value";
                type, TEXT;
                mouse_events, 1;
                effect, NONE;

                description {
                    state, "default" 0.0;
                    visible, 1;

                    rel1 {
                        relative, 0.0 0.0;
                        offset, 0 0;
                    }
                    rel2 {
                        relative, 1.0 1.0;
                        offset, 0 0;
                    }
                    color, 255 255 255 255;

                    text {
                        text, "";
                        font, "Vera";
                        size, 10;
                    }
                }
            }
        }
        programs {
            program {
                name, "center_click";
                signal, "mouse,clicked,2";
                source, "element.value";
                action, SIGNAL_EMIT "item_selected" "item_selected";
            }
```

```
        }
    }
}
```

The element group specifies how each element of the container is to be displayed. You will notice that the names given here match up to the names searched for in the application code itself while creating the elements.

There is one program in this group which will emit a signal of "item_selected" when the middle mouse button is pressed while hovering over one of the elements in the list.

Thats the end of the EDC code. To compile the app code, a makefile similar to that below could be used.

Example 6.26. Makefile

```
CFLAGS = `ecore-config --cflags` `evas-config --cflags` `esmart-config --cflags`
LIBS = `ecore-config --libs` `evas-config --libs` `esmart-config --libs` \
            -lesmart_container

container_ex: container/container_ex.c
        gcc -o container/container_ex container/container_ex.c $(CFLAGS) $(LIBS)
```

And to create the EET file, a simple 'edje_cc default.edc' should suffice as long as the Vera.ttf file is in the current directory.

Thats it, assuming everything goes as planned, you should have a simple app in which clicking the right/ left mouse buttons moves the container to different portions of the window. While clicking the middle mouse button on elements prints out the number of the element printed.

Chapter 7. Epeg & Epsilon

In this modern era of digital photography presentation becomes a problem due to the shear volume of images being created. Unlike the old days when film was used sparingly we are now generating hundreds or thousands of images in a week. The solution to this presentation problem is the thumbnail, a scaled down images that can be indexed in a table or application and quickly scanned visually to find the images you desire. But image scaling is a very intensive operation, even though it might only take your mighty AMD Athlon 1 second to scale down a 1600x1200 photo to your desired resolution, if you have 2,000 photos that'll take 30 minutes, and this assumes your not doing the operation manually in an editor such as Photoshop or GIMP. The problem clearly calls for a tool that can scale images with blazing speed and effeciency, with as much control avalible as possible. The solution is answered by two libraries in the EFL: Epeg and Epsilon.

Epeg was written by Raster to handle the exactly problem mentioned above with his image galleries found at rasterman.com. It is, hands down, the fastest thumbnailer on the planet.With a simple to use API, it can be integrated into any application you could want. The only downside is that it only handles JPEGs (hence the name), but this is hardly a problem considering that every camera on the market uses JPEG as it's default output format.

Epsilon was written by Atmos, inspired by Epeg's blazing speed but in responce to a need for multi-format thumbnailing capability. Epsilon can handle JPEG, PNG, XCF, and GIF. Obviously because it's not a JPEG specific library it doesn't handle JPEG as fast as Epeg, but it can actually use Epeg itself to gain the speed advantages it provides. Epsilon, unlike Epeg, conforms to the freedesktop.org Thumbnail Managing Standard [http://triq.net/~jens/thumbnail-spec/index.html]. As such, it outputs all thumbnails into the directory structure specified by the standard (~/.thumbnails/) rather than to a coder defined location.

Both libraries perform such specific tasks that the APIs are very simple to use. Epeg has only 17 functions and Epsilon has only 9 making these libraries very simple to learn and utilize in your applications right away.

Recipe: Simple Thumbnailing with Epeg

Ben 'technikolor' Rockwood <benr@cuddletech.com>

The simplest thumbnailing application we could write would simply take two arguments, the input filename (image) and the output filename (thumbnail). The following code example uses Epeg to do just that.

Example 7.1. Epeg Simple Thumbnail

```
#include <Epeg.h>

int main(int argc, char *argv[]){

        Epeg_Image * image;
        int w, h;

        if(argc < 2) {
                printf("Usage: %s input.jpg output.jpg\n", argv[0]);
                return(1);
        }

        image = epeg_file_open(argv[1]);
```

```
        epeg_size_get(image, &w, &h);
        printf("%s -  Width: %d, Height: %d\n", argv[1], w, h);
        printf("  Comment: %s", epeg_comment_get(image) );

        epeg_decode_size_set(image, 128, 96);
        epeg_file_output_set(image, argv[2]);
        epeg_encode(image);
        epeg_close(image);

        printf("... Done.\n");
        return(0);
}
```

This example is fairly simplistic, not checking to ensure the input is truely a JPEG, but adiquately points out some features of the library. It can be compiled in the following way:

Example 7.2.

```
gcc `epeg-config --libs --cflags` epeg-test.c -o epeg-test
```

The `epeg_file_open` function opens a JPEG to be manipulated, returning a Epeg_Image pointer. This pointer can then be feed to other Epeg functions for manipulation.

Two diffrent functions are used here to get some information about the input image: `epeg_size_get` and `epeg_comment_get`. Notice that neither return values from these functions are ever used in other Epeg functions, they are simply for informational display. A good use for these return values might be to intellegently define the output thumbnail size, or to modify and pass on a comment to the output thumbnail.

The next set of functions actually do the work. `epeg_decode_size_set` defines the output size of the thumbnail. `epeg_file_output_set` defines the output file name of the thumbnail. And `epeg_encode` actually does the heavy lifting. Notice that while we aren't checking for success here, that `epeg_encode` actually returns an int allowing us to check for success.

Once the thumbnail is created, just call `epeg_close` to seal the deal.

While this example is perhaps overly-simplistic you can see how the basics work. Epeg also has functions for trimming, thumbnail commenting, enabling and disabling comments on thumbnails, colorspace conversion, and quality settings changes which can be used to get just the result you want.

Recipe: Simple Thumbnailing with Epsilon

Ben 'technikolor' Rockwood <benr@cuddletech.com>

Epsilon creates thumbnails that conform to the freedesktop.org Thumbnail Managing Standard [http://triq.net/~jens/thumbnail-spec/index.html]. Thumbnails can be created for a variety of formats, including native PNG support, Epeg support, or any format supported by Imlib2. Lets look at a simple Epsilon application similar to the earlier Epeg example.

Example 7.3. Epsilon Simple Thumbnail

```
#include <stdio.h>
#include <Epsilon.h>

int main(int argc, char *argv[]){

        Epsilon * image = NULL;
        Epsilon_Info *info;

        if(argc < 1) {
                printf("Usage: %s input_image\n", argv[0]);
                return(1);
        }

        epsilon_init();

        image = epsilon_new(argv[1]);

        info = epsilon_info_get(image);
        printf("%s -  Width: %d, Height: %d\n", argv[1], info->w, info->h);


        if (epsilon_generate(image) == EPSILON_OK) {
                printf("Thumbnail created!\n");
        } else {
                printf("Generation failed\n");
        }
        epsilon_free(image);

        return(0);
}
```

It can be compiled in the following way:

Example 7.4.

```
gcc `epsilon-config --libs --cflags` epsilon-simple.c -o epsilon-simple
```

You'll notice almost immediately that no output filename is accepted, nor is any output function used. The freedesktop.org Thumbnail Managing Standard specifies that all thumbnails are to be created in the ~/.thumbnail directory tree. This centralized thumbnail storage allows for sharing of thumbnails between multiple applications that adhere to the standard. After compiling and running the example code against an image look for the thumbnail in ~/.thumbnails/large. Thumbnails are also named according to the standard, replacing the origonal name with an MD5 checksum so that even if the image is renamed the thumbnail doesn't need to be regenerated.

In our example we start by verifying that we get an input image to thumbnail and then initialize Epsilon using the `epsilon_init` function. `epsilon_new` accepts a single argument, the image to be thumbnailed, and returns an epsilon pointer which is used by the other functions.

Epsilon has the ability to pull some basic information from your images. In the above example we use `epsilon_info_get` to return a Epsilon_Info structure containing the input images modification time (mtime), location (URI), width, height, and MIME type. Here we simply report the image width and height using the info structures `w` and `h` elements.

`epsilon_generate` is the heavy lifter. This function will actually generate the thumbnail and place it in the proper location. Its return value indicates success, which the Epsilon header provides CPP macro definations for: `EPSILON_FAIL` and `EPSILON_OK`.

Clean up is provides by `epsilon_free`.

Epsilon, as seen here, is very simple to use and integrate into any application relying on thumbnails. Not only is a simple API provided but integration with the reigning standard used for thumbnailing at no extra cost. For additional information about Epsilon, see the Epsilon Doxygen on Enlightenment.org.

Chapter 8. Etox

Etox is an advanced type setting and text layout library based on Evas, that allows for functionality above and beyond that provided by Evas itself. It is able to simplify displaying, moving, resizing, layering, and clipping of text, as well as text alignment, wrapping, and modification. Etox can even intellegently wrap around obsticles and apply pre-defined *styles*. Nearly any aspect of textual layout can be handled easily and efficiently by leveraging Etox.

Recipe: Etox Overview

Ben 'technikolor' Rockwood <benr@cuddletech.com>

In order to get using Etox quickly, a simple overview example is worthwhile. In the following example code we'll create an X11 Evas using Ecore_Evas and then put some Etox text on it.

Example 8.1. Etox Overview Example

```
#include <Ecore_Evas.h>
#include <Ecore.h>
#include <Etox.h>

#define WIDTH 400
#define HEIGHT 200

        Ecore_Evas  *   ee;
        Evas        *   evas;
        Evas_Object *   base_rect;
        Evas_Object *   etox;
        Etox_Context *  context;

int main(){

        ecore_init();

   ee = ecore_evas_software_x11_new(NULL, 0,  0, 0, WIDTH, HEIGHT);
        ecore_evas_title_set(ee, "ETOX Test");
        ecore_evas_borderless_set(ee, 0);
        ecore_evas_show(ee);

   evas = ecore_evas_get(ee);
        evas_font_path_append(evas, ".");

   base_rect = evas_object_rectangle_add(evas);
        evas_object_resize(base_rect, (double)WIDTH, (double)HEIGHT);
        evas_object_color_set(base_rect, 255, 255, 255, 255);
        evas_object_show(base_rect);

   etox = etox_new(evas);
        evas_object_resize(etox, WIDTH, HEIGHT);


   context = etox_get_context(etox);
        etox_context_set_color(context, 0, 0, 0, 255);
        etox_context_set_font(context, "Vera", 32);
        etox_context_set_align(context, ETOX_ALIGN_LEFT);

        etox_set_soft_wrap(etox, 1);
        etox_set_text(etox, "Welcome to the world of Etox!");
```

```
        evas_object_show(etox);

        ecore_main_loop_begin();

        return 0;
}
```

This example can be compiled in the following way:

Example 8.2.

```
gcc `etox-config --libs --cflags` `ecore-config --libs --cflags` etox-test.c -o et
```

Most of thie example is standard Ecore_Evas functions, so we'll concentrate just on the Etox related parts of it. Notice that we use the `evas_font_path_append()` Evas function to define our font path, this is something that Etox will not do for you.

Your Etox text will always start by adding a new Etox using the `etox_new()` function which returns a new Evas_Object. Because your Etox is itself an Evas object it can be manipulated as such. Etox's layout functions such as clipping and wrapping are dependant upon the size of the Etox itself, therefore `evas_object_resize()` need to be called to define the proper size of the Etox. Please note that the area of the object will *not* default to the size of the Evas itself.

Etox uses the concept of contexts. A `context` is a set of parameters such as color, font, alignment, styles, and markers that are applied to a certain set of text. Each Etox object has at least one context associated with it which is created when calling `etox_new()`. For this reason the `etox_context_new()` function *only* needs to be called when creating additional contexts.

Once you've used `etox_new()` to add your Etox object you need to use the `etox_get_context()` to return a `Etox_Context` pointer which can then be passed to other context functions to modify the attributes of your text. In the example we change the color, font and alignment of our context.

Two of the most interesting and simplistic features of Etox is it's ability to intellegently wrap text and to interpret a standard C newline charrector (\n) as a wrap. These are features that Evas itself does not contain, it is the responsability of the coder to ensure text doesn't trail off the canvas.

Intellegent wrapping comes in two flavors which are not mutually exclusive. The first is soft wrapping, which will wrap text when a charrector is going to exceed the width of the canvas. The second is word wrapping, which will wrap text when a word is going to exceed the width of the canvas. Typically the later is desirable so that you get "This is (wrap) my string" instead of "This is m(wrap)y string." Note, however, that work wrapping will not work unless soft wrapping has already been turned on, thus work wrapping requires calling *both* `etox_set_soft_wrap()` and `etox_set_word_wrap()` functions.

A final note about wrapping is that by default wrapping will insert a `wrap marker` into your output string, a "+" (plus) sign by default. This marker indicates that a wrap has occured and is printed as the first charrector on the next line of the wrap. Your string will therefore likely look more like this: "This is my (wrap) +string." If you would prefer that Etox silently wrap without a marker, simply set the marker to being nothing using the `etox_context_set_wrap_marker()` function.

Etox text strings themselves are set using the `etox_set_text()`. It's important to note that the string applies to the Etox itself and not to the context. There is no direct association between the string and the

context, which facilitates easy modification of the display of the text without having to change the context, or vice versa.

While this is a very simple example of Etox usage, far more can be accomplished and as you can see the API is simple and clean, filling in many of the text handling needs that Evas doesn't provide.

Chapter 9. Edje

Edje is a complex graphical design and layout library. It's purpose is to abstract every element of your Evas applications interface from the code itself.

An Edje application falls into two parts: the C code that makes up your application and an Edje Data Collection (EDC) that describes every element of your interface. The two are connected by signals that are emited from your EDC and recieved by callbacks in your application code. Using this signal model the application code is completely uninterested in exactly what your interface looks like, just so long as it gets a signal. And because signals are proccessed by callbacks, there is no requirement that your interface send every signal avalible, making large scale applications and "demo" size applications possible with a single binary. Whether your interface uses buttons or a drag-bar in order to send data, it doesn't matter to you application.

Recipe: A Template for building Edje applications

Ben 'technikolor' Rockwood <benr@cuddletech.com>

The following example is a template that can be used for quick and easy setup of a Edje application. It looks similar to the template found in the Evas chapter, since this too uses Ecore_Evas.

Example 9.1. Edje Template

```
#include <Ecore_Evas.h>
#include <Ecore.h>
#include <Edje.h>

#define WIDTH 100
#define HEIGHT 100

int app_signal_exit(void *data, int type, void *event);

        /* GLOBALS */
        Ecore_Evas  *   ee;
        Evas        *   evas;
        Evas_Object *   edje;

        Evas_Coord   edje_w, edje_h;


int main(int argv, char *argc[]){

        ecore_init();
        ecore_event_handler_add(ECORE_EVENT_SIGNAL_EXIT, app_signal_exit, NULL);

        ecore_evas_init();

   ee = ecore_evas_software_x11_new(NULL, 0,  0, 0, WIDTH, HEIGHT);
        ecore_evas_title_set(ee, "TITLE");
        ecore_evas_borderless_set(ee, 0);
        ecore_evas_shaped_set(ee, 0);
        ecore_evas_show(ee);


   evas = ecore_evas_get(ee);
```

```
        evas_font_path_append(evas, "edje/fonts/");

        edje_init();
   edje = edje_object_add(evas);
        edje_object_file_set(edje, "edje/XXX.eet", "XXX");
        evas_object_move(edje, 0, 0);
        edje_object_size_min_get(edje, &edje_w, &edje_h);
        evas_object_resize(edje, edje_w, edje_h);
        evas_object_show(edje);

        ecore_evas_resize(ee, (int)edje_w, (int)edje_h);
        ecore_evas_show(ee);

        /* Insert Objects and callbacks here */

        ecore_main_loop_begin();

        return 0;
}


int app_signal_exit(void *data, int type, void *event){

        printf("DEBUG: Exit called, shutting down\n");
        ecore_main_loop_quit();
        return 1;
}
```

Compile this template in the following way:

```
 gcc `edje-config --cflags --libs` `ecore-config --cflags --libs` edje_app.c -o ed
```

The important calls here to look at are contained in the Edje block, following `edje_init()`.

`edje_object_file_set()` defines which Edje EET is used as well as the name of the collection to use.

The rest of the Edje/Evas functions in the Edje block are needed to resize the X11 window to accomidate your Edje. We start by moving the Evas window and then getting the minimum size of the Edje itself using `edje_object_size_min_get()`. Then using `evas_object_resize()` we can resize the Edje, which is really an Evas object, to the size of the Evas itself. Following this we can show the Edje and then resize the Evas itself (and thanks to Ecore the window too) using `ecore_evas_resize()`.

Beyond this callbacks can be added to attach to your interface.

Recipe: Creating/Triggering Edje Callbacks

dan 'dj2' sinclair <zero@perplexity.org>

It is sometimes necessary to signal to your main program that some event has happened in your user interface. But you don't want parts of the implementation to bleed into the UI design. This can be done easily in Edje by triggering a signal from your EDC program and attaching a callback to that signal in the C program.

Example 9.2. Callback program

```
#include <stdio.h>
#include <Ecore.h>
#include <Ecore_Evas.h>
#include <Edje.h>

int exit_cb(void *data, int type, void *ev);
void edje_cb(void *data, Evas_Object *obj,
                     const char *emission, const char *source);

int
main(int argc, char ** argv)

{
    int ret = 0;
    Ecore_Evas *ee = NULL;
    Evas *evas = NULL;
    Evas_Object *edje = NULL;
    Evas_Coord w, h;

    if (!ecore_init()) {
        printf("error setting up ecore\n");
        goto EXIT;
    }
    ecore_app_args_set(argc, (const char **)argv);

    if (!ecore_evas_init()) {
        printf("error setting up ecore_evas\n");
        goto ECORE_SHUTDOWN;
    }

    if (!edje_init()) {
        printf("error setting up edje\n");
        goto ECORE_SHUTDOWN;
    }
    ecore_event_handler_add(ECORE_EVENT_SIGNAL_EXIT, exit_cb, NULL);

    ee = ecore_evas_software_x11_new(NULL, 0, 0, 0, 200, 300);
    ecore_evas_title_set(ee, "Edje CB example");
    ecore_evas_show(ee);

    evas = ecore_evas_get(ee);
    edje = edje_object_add(evas);
    edje_object_file_set(edje, "default.eet", "main");
    evas_object_move(edje, 0, 0);
    edje_object_size_min_get(edje, &w, &h);
    evas_object_resize(edje, w, h);
    ecore_evas_resize(ee, w, h);
    evas_object_show(edje);

    edje_object_signal_callback_add(edje, "foo", "bar", edje_cb, NULL);

    ecore_main_loop_begin();
    ret = 1;

    edje_shutdown();
ECORE_SHUTDOWN:
    ecore_shutdown();
EXIT:
    return ret;
}

int
exit_cb(void *data, int type, void *ev)
```

```
{
    ecore_main_loop_quit();
    return 1;
}

void
edje_cb(void *data, Evas_Object *obj,
                    const char *emission, const char *source)
{
    printf("got emission: %s from source: %s\n", emission, source);
}
```

Most of this is standard setup stuff for Ecore, Ecore_Evas and Edje. The callback is attached with `edje_object_signal_callback_add(Evas_Object *o, char *emission, char *source, (void *)func(void *data, Evas_Object *obj, const char *emission, const char *source), void *user_data)`. The object `o` that the callback is attached to is the Edje object that was created with our EDC file.

The `emission` and `source` values need to be strings that match up to the emit call in the EDC program which will be seen later. The other option is to place a '*' in the emission or source. This will cause the value set to '*' to match on anything. If you wish to catch all the signals that edje emits, you can set both the emission and the source to '*'.

The `func` is the function to call and finally the `user_data` is any extra data you wished passed to the callback.

The callback function can be seen in `edje_cb`. This will receive the user data, the edje object the callback was made on and the emission and source strings.

To activate the callback our EDC file needs a program which will emit the required emission and source.

Example 9.3. EDC file

```
collections {
    group {
        name: "main";
        min: 200 100;

        parts {
            part {
                name: "bg";
                type: RECT;

                description {
                    rel1 {
                        relative: 0.0 0.0;
                        offset: 0 0;
                    }
                    rel2 {
                        relative: 1.0 1.0;
                        offset: -1 -1;
                    }
                    color: 255 255 255 255;
                }
            }
            part {
                name: "button";
```

```
                type: RECT;

                description {
                    rel1 {
                        relative: .4 .4;
                        offset: 0 0;
                    }
                    rel2 {
                        relative: .6 .6;
                        offset: 0 0;
                    }
                    color: 0 0 0 255;
                }
            }
        }
        programs {
            program {
                name: "down";
                signal: "mouse,down,*";
                source: "button";
                action: SIGNAL_EMIT "foo" "bar";
            }
        }
    }
}
```

The piece of interest is the `action: SIGNAL_EMIT "foo" "bar"` this will cause edje to emit an emission of `foo` with a source of `bar`.

Example 9.4. Compilation

```
zero@oberon [edje_cb] -> edje_cc default.edc
zero@oberon [edje_cb] -> gcc -o cb main.c `ecore-config --cflags --libs` \
        `edje-config --cflags --libs`
```

Edje makes it really simple for an interface to be totally abstract from the implementation. The only knowledge the interface has is to send the correct emissions and sources as events happen.

Recipe: Working with Edje files

dan 'dj2' sinclair <zero@perplexity.org>

When your working with .edc files and .eet files you often need to transform one into the other. In order to do this Edje provides a set of tools to facilitate these transformations.

The available programs include:

edje_cc	Compile an EDC file, images and fonts into an EET file
edje_decc	De-compile an EET file back into its EDC file, images and fonts
edje_recc	Re-compile an EET file
edje_ls	List the groups in an EET file

edje	Display the groups in an EET file

Each of these programs are discussed in more detail below.

edje_cc

`edje_cc` is one of the main Edje programs that you will be using. It is responsible for the compilation of your EDC files, including images and fonts, into there corresponding EET files.

Example 9.5. edje_cc Usage

```
edje_cc [OPTIONS] input_file.edc [output_file.eet]
```

Options

-id *image/directory*	Add a directory to look in for relative path images
-fd *font/directory*	Add a directory to look in for relative path fonts
-v	Verbose output
-no-lossy	Do NOT allow images to be lossy
-no-comp	Do NOT allow images to be stored with lossless compression
-no-raw	Do NOT allow images to be stored with zero compression (raw)
-min-quality *VAL*	Do NOT allow lossy images with quality < VAL (0-100)
-max-quality *VAL*	Do NOT allow lossy images with quality > VAL (0-100)
-scale-lossy *VAL*	Scale lossy image pixels by this percentage factor (0 - 100)
-scale-comp *VAL*	Scale lossless compressed image pixels by this percentage factor (0 - 100)
-scale-raw *VAL*	Scale uncompressed (raw) image pixels by this percentage factor (0 - 100)
-D*define_val=to*	CPP style define to define input macro definitions to the .edc source

edje_decc

`edje_decc` allows you to decompile EET files back into there EDC, image and font parts. This makes is easy to distributed your source as you only ever need to provide the EET file and the end user will have access to the source and the final product.

Example 9.6. edje_decc Usage

```
edje_decc input_file.eet
```

edje_recc

`edje_recc` allows you to recompile an EET file without needed to first decompile. This allows you to modify the parameters passed into the `edje_cc` to better fit your viewing and EET size requirements.

Example 9.7. edje_recc Usage

```
edje_recc [OPTIONS] input_file.eet
```

Options

-v	Verbose output
-no-lossy	Do NOT allow images to be lossy
-no-comp	Do NOT allow images to be stored with lossless compression
-no-raw	Do NOT allow images to be stored with zero compression (raw)
-min-quality *VAL*	Do NOT allow lossy images with quality < VAL (0-100)
-max-quality *VAL*	Do NOT allow lossy images with quality > VAL (0-100)

edje_ls

`edje_ls` provides a listing of all the groups inside a given EET file. This is a quick way to see what a given EET is providing.

Example 9.8. edje_ls Usage

```
edje_ls [OPTIONS] input_file.eet ...
```

Options

-o *outputfile.txt*	Output the listing of the collections to a file

edje

`edje` is also one of the main programs you will be using. `edje` allows you to view each of the groups

of your program. It allows you to see how your parts are going to look and how they react to certain signals.

Example 9.9. edje Usage

```
edje file_to_show.eet [OPTIONS] [collection_to_show] ...
```

Options

-gl	Render with OpenGL
-g *WxH*	Set window geometry to WxH
-fill	Make the parts fill the entire window

These five simple tools should provide all of the access to build and maintain your Edje EETs. They also make it easy to retrieve the source that comprises a given EET making it easy to learn how different effect operate.

Chapter 10. Edje EDC & Embryo

Edje Data Collections (EDC) source files allow for easy creation of rich and powerful graphical interfaces. Your Edje application is divided into two distinct parts, the application code (using calls from `Edje.h`) and the interface description in the EDC. The only connectivity needed between your interface and your application code are signals emited by your interface and are recieved by Edje callbacks in your application code.

An EDC is broken into several major sections describing the images and fonts that are used in the interface, descriptions of how the various `parts` of the interface are laid out, and descriptions of actions or `programs` that occur when your interface is interacted with. This functionality can be further supplimented by employing Embryo's scripting language to add C like programability into the Edje EDC itself.

The end result of an EDC, including all of it's fonts and images, is a single EET. Because the entire interface is made avalible in a single file "theme" distribution is greatly simplified.

While Edje EDC's might be considered "themes" they are much more. A traditional "theme" is a file or group of files that augment some existing interface by changing the color of elements or replacing the images that make up the interface itself. But these methods insuffient for really changing the design of an applications interface, limiting themers from modification and often requiring an application redesign at some point in order to expand the capabilities of the interface for greater funcationlity. A GTK application will always look the same, despite what theme it is using. A simple example might be that a GTK or QT application will always have a rectangular shape and if it isn't borderless you can't make it borderless via a theme. However, an Edje application could be change from rectangular to an oval shaped border with a simple modification of the EDC, or could remove and rearrange all of the elements of the interface without ever touching application code. In this way Edje allows a far greater ammount of control and flexablity than provided by any other solution in the Open Source comminity and allows the Open programing model to be truely leverage to allow even non-programmers (as many themers are) to contribute and modify things as they see fit.

Recipe: Edje/Embryo toggle

Corey 'atmos' Donohoe `<atmos@atmos.org>`

Long ago Raster [http://www.rasterman.com] made Edje, and it was good. The cavemen who discovered Edje on the cave walls (#edevelop) were amazed, but early on there were many drawbacks. Given enough creativity you could get away with things, toggles for example, but it was alchemy to do properly. For historical purposes, an Edje Toggle without Embryo is provided, as convoluted as it is. See the Edje without Embryo example below.

You'll notice that you have to speak in signals to your application to determine the state of your toggle. So without further ado, here's an Edje toggle utilizing embryo, in a *much* more elegant fashion.

Embryo Scripting inside of Edje, henceforth EE scripting, gives you variables. You can have integers, floating point numbers, and strings. This means that you can basically have some programming logic in your edjes. Nothing complex, like composite structs, but simple variables contained in a group might resemble the members of structs.

The first part of EE is choosing your variables. In this simple example we only have one variable, and you get it involved in an edje group by declaring a *script { ... }* block. *button_toggle_state* is implicitly an integer, and it will be used as a boolean variable to let us know whether or not the toggle button is on or off. The cool thing about this variable is we can use it as a way of communicating between our application and our edje. Furthermore you can rest easy knowing (assuming you did it right) that some edje trickery isn't going to throw your app into limbo.

Example 10.1. Creating the variable

```
collections {
    group {
        name: "Toggler";
        script {
            public button_toggle_state;
        }
        parts {
            part {
                ...
            }
        }
        programs {
            program {
                ...
            }
        }
    }
}
```

The second part of EE Scripting is initializing your variables. For the most part you can assume these variables will be initialized to zero, but it's good practice to explicitly set them yourself. Edje emits a "load" signal when the group is loaded into memory, this is your opportunity to set your embryo variables.

Example 10.2. Initializing variables

```
program {
    name: "group_loaded";
    signal: "load";
    source: "";
    script {
        set_int(button_toggle_state, 0);
    }
}
```

The third part is actually giving your edje a look. For this example rectangles are used, but images and even text should also work fine. There's a background object just for consistency's sake, and there's a rectangle called "toggler". toggler has two states, default (implicitly off) and on. When toggler is clicked it should, you guessed it, toggle to the other state. off -> on, on -> off. toggler is going to have its default (off) state color red, and it's going to have its on state blue so they can easily be differentiated. The background will be white cause it's not red or blue. :D

Example 10.3. The toggler button

```
collections {
    group {
```

```
        name: "Toggler";
        script {
            public button_toggle_state;
        }
        parts {
            part {
                name: "background";
                type: RECT;
                mouse_events: 0;
                description {
                    state: "default" 0.0;
                    color: 255 255 255 255;
                    rel1 { relative: 0.0 0.0; offset: 0 0; }
                    rel2 { relative: 1.0 1.0; offset: 0 0; }
                }
            }
            part {
                name: "toggle";
                type: RECT;
                mouse_events: 1;
                description {
                    state: "default" 0.0;
                    color: 255 0 0 255;
                    rel1 { relative: 0.0 0.0; offset: 10 10; }
                    rel2 { relative: 1.0 1.0; offset: -10 -10; }
                }
                description {
                    state: "on" 0.0;
                    color: 0 0 255 255;
                    rel1 { relative: 0.0 0.0; offset: 10 10; }
                    rel2 { relative: 1.0 1.0; offset: -10 -10; }
                }
            }
        }
        programs {
            program {
                name: "group_loaded";
                signal: "load";
                source: "";
                script {
                    set_int(button_toggle_state, 0);
                }
            }
        }
    }
}
```

The fourth part is hooking in the mouse events to trigger the toggling as edje programs. Not only changing the Embryo variable, but also change the appearance of our edje. This example uses normal left mouse button clicking to change the toggles state, in edje terms "mouse,clicked,1". This example doesn't use the built in Embryo *set_state* function call, but emits signals which are trapped by other programs. The reasoning behind this approach is to allow for smooth visual transitions between the two states. The Embryo set_state function call is an immediate state change, and it just doesn't look a nice as the SINUSOIDAL transition used in the following snippets.

Example 10.4. Hooking into the mouse events

```
collections {
    group {
        name: "Toggler";
        script {
            public button_toggle_state;
        }
        parts {
            part {
                ...
            }
        }
        programs {
            program {
                name: "toggle_icon_mouse_clicked";
                signal: "mouse,clicked,1";
                source: "toggle";
                script {
                    if(get_int(button_toggle_state) == 0) {
                        set_int(button_toggle_state, 1);
                        emit("toggle,on", "");
                    }
                    else {
                        set_int(button_toggle_state, 0);
                        emit("toggle,off", "");
                    }
                }
            }
            program {
                name: "toggle_on";
                signal: "toggle,on";
                source: "";
                action: STATE_SET "on" 0.0;
                target: "toggle";
                transition: SINUSOIDAL 0.5;
            }
            program {
                name: "toggle_off";
                signal: "toggle,off";
                source: "";
                action: STATE_SET "default" 0.0;
                target: "toggle";
                transition: SINUSOIDAL 0.5;
             }
         }
     }
 }
```

The fifth part is pondering the scenario presented. This is only the tip of the iceberg with respect to EE scripting. You can add many more variables to keep track of internal states that are completely unrelated to your apps. There are nuances between this and practical usage of Embryo variables, however understanding these building blocks will make working with or writing EE scripted apps much more simple.

- What's bad about the technique presented here?
- What if the app wants the toggle "on" by default?

You can use a script similar to the following the build this example.

Example 10.5. Build script

```
#!/bin/sh -e
THEME="default"
APPNAME=""
edje_cc -v $THEME.edc $THEME.eet
if [ $? = "0" ]; then
    if [ "$APPNAME" = "" ]; then
        echo "Build was successful"
    else
        PREFIX=`dirname \`which $APPNAME\` | sed 's/bin//'`
        sudo cp $THEME.eet $PREFIX"share/$APPNAME/themes/"
        echo -n "Installed theme to "
        echo $PREFIX"share/$APPNAME/themes/"
    fi
else
    echo "Building failed"
fi
```

Example 10.6. Edje toggle without Embryo

```
images { }

collections {
    group {
        name, "Rephorm";
        min, 50 50;
        max, 75 75;
        parts {
            part {
                name, "Clip";
                type, RECT;
                mouse_events, 0;
                description {
                    state, "default" 0.0;
                    visible, 1;
                    rel1 { relative, 0.0 0.0; offset, 5 5; }
                    rel2 { relative, 1.0 1.0; offset, -5 -5; }
                    color, 255 255 255 255;
                }
                description {
                    state, "hidden" 0.0;
                    visible, 1;
                    rel1 { relative, 0.0 0.0; offset, 5 5; }
                    rel2 { relative, 1.0 1.0; offset, -5 -5; }
                    color, 255 255 255 128;
                }
            }
            part {
                name, "On";
                type, RECT;
                mouse_events, 1;
                clip_to, "Clip";
                description {
                    state, "default" 0.0;
                    visible, 0;
                    rel1 { relative, 0.0 0.0; offset, 5 5; }
                    rel2 { relative, 1.0 1.0; offset, -5 -5; }
                    color, 255 0 0 0;
```

```
            }
            description {
                state, "visible" 0.0;
                visible, 1;
                rel1 { relative, 0.0 0.0; offset, 5 5; }
                rel2 { relative, 1.0 1.0; offset, -5 -5; }
                color, 255 0 0 255;
            }
        }
        part {
            name, "Off";
            type, RECT;
            mouse_events, 1;
            clip_to, "Clip";
            description {
                state, "default" 0.0;
                visible, 1;
                rel1 { relative, 0.0 0.0; offset, 5 5; }
                rel2 { relative, 1.0 1.0; offset, -5 -5; }
                color, 0 0 255 255;
            }
            description {
                state, "visible" 0.0;
                visible, 0;
                rel1 { relative, 0.0 0.0; offset, 5 5; }
                rel2 { relative, 1.0 1.0; offset, -5 -5; }
                color, 0 0 255 0;
            }
        }
        part {
            name, "Grabber";
            type, RECT;
            mouse_events, 1;
            repeat_events, 1;
            clip_to, "Clip";
            description {
                state, "default" 0.0;
                visible, 1;
                rel1 { relative, 0.0 0.0; offset, 5 5; }
                rel2 { relative, 1.0 1.0; offset, -5 -5; }
                color, 255 255 255 0;
            }
        }
    }
    programs {
        program {
            name, "ToggleOn";
            signal, "mouse,clicked,1";
            source, "Off";
            action, STATE_SET "visible" 0.0;
            target, "Off";
            target, "On";
            transition, SINUSOIDAL 0.5;
        }
        program {
            name, "ToggleOff";
            signal, "mouse,clicked,1";
            source, "On";
            action, STATE_SET "default" 0.0;
            target, "Off";
            target, "On";
            transition, SINUSOIDAL 0.5;
        }
        program {
```

```
                name, "GrabberIn";
                signal, "mouse,in";
                source, "Grabber";
                action, STATE_SET "default" 0.0;
                target, "Clip";
                transition, SINUSOIDAL 0.5;
            }
            program {
                    name, "GrabberOut";
                signal, "mouse,out";
                source, "Grabber";
                action, STATE_SET "hidden" 0.0;
                target, "Clip";
                transition, SINUSOIDAL 0.5;
            }
        }
    }
}
```

Recipe: Edje text effect fading

dan 'dj2' sinclair <zero@perplexity.org>

Text effects can make your program look cool. But what if you want to fade those text effects with your text? Well, Edje makes it possible and relatively simple.

All you need to do is to fade out the `color3` attribute as your fading the `color` attribute of your text. The `color3` will change the colour values of the effect.

This is illustrated in the following example.

Example 10.7. Fade effect with text

```
collections {
    group {
        name, "Main";
        min, 30 30;

        parts {
            part {
                name, "foo";
                type, TEXT;
                effect, SOFT_SHADOW;
                mouse_events, 1;

                description {
                    state, "default" 0.0;
                    rel1 {
                        relative, 0 0;
                        offset, 0 0;
                    }
                    rel2 {
                        relative, 1.0 1.0;
                        offset, -1 -1;
                    }

                    text {
                        text, "foo text";
```

```
                        font, "Vera";
                        size, 22;
                    }
                    color, 255 255 255 255;
                    color3, 0 0 0 255;
                }
                description {
                    state, "out" 0.0;
                    rel1 {
                        relative, 0 0;
                        offset, 0 0;
                    }
                    rel2 {
                        relative, 1.0 1.0;
                        offset, -1 -1;
                    }

                    text {
                        text, "foo text";
                        font, "Vera";
                        size, 22;
                    }
                    color, 0 0 0 0;
                    color3, 255 255 255 0;
                }
            }
        }
        programs {
            program {
                name, "mouse.in";
                signal, "mouse,in";
                source, "foo";
                action, STATE_SET "out" 0.0;
                transition, SINUSOIDAL 2.0;
                target, "foo";
            }
            program {
                name, "mouse.out";
                signal, "mouse,out";
                source, "foo";
                action, STATE_SET "default" 0.0;
                transition, SINUSOIDAL 2.0;
                target, "foo";
            }
        }
    }
}
```

This example can be compiled into a `.eet` with the following command.

Example 10.8. Compliation

```
zero@oberon[edje_text] -> edje_cc text.edc
```

By altering the `color3` value along with the `color` value you will be able to alter the appearance of your effects along with your text.

Chapter 11. EWL

The Enlightened Widget Library (EWL) is a widget toolkit that builds upon the foundations constructed by the other libs in the EFL. Ewl uses Evas as its rendering backend and its appearence is abstracted out for Edje to handle.

EWL is similar in feel to many of the other toolkits out there including GTK, QT or MOTIF. The APIs differ but the concepts are the same.

Recipe: EWL Introduction

dan 'dj2' sinclair <zero@perplexity.org>

Thought the use of the Enlightened Widget Library (EWL), a lot of power can be delivered into the programmers hands with little effort.

This introduction to EWL will show how to create a simple text viewing application with a menu bar and file dialog. The text area will have scrollbars and will also allow scrolling using either the keyboard arrow keys, or a mouse wheel.

Example 11.1. Includes and declarations

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <Ewl.h>

#define PROG    "EWL Text Viewer"

/* globals */
static Ewl_Widget *main_win = NULL;
static Ewl_Widget *fd_win = NULL;

/* pre-declarations */
static void destroy_cb(Ewl_Widget *, void *, void *);
static void destroy_filedialog_cb(Ewl_Widget *, void *, void *);
static void open_file_cb(Ewl_Widget *, void *, void *);
static void home_cb(Ewl_Widget *win, void *ev, void *data);
static void file_menu_open_cb(Ewl_Widget *, void *, void *);
static void key_up_cb(Ewl_Widget *, void *, void *);

static char *read_file(char *);
static void mk_gui(void);
```

The only required include to write an Ewl application is the <Ewl.h> declaration. We make the main window and the file dialog window global to facilitate easier access in the callback functions. They do not need to be global, but for the purposes of this example, its simpler if they are.

Example 11.2. main

```
/* lets go */
int main(int argc, char ** argv) {
    ewl_init(&argc, argv);
    mk_gui();
    ewl_main();
    return 0;
}
```

The main function for our text viewer is very simplistic. We start by initializing ewl through the `ewl_init()` call. Ewl takes the argc and argv entries to do some command line parsing of its own. This includes such things as setting the Ewl theme to use (--ewl-theme) or setting the rendering engine (--ewl-software-x11, --ewl-gl-x11, etc.).

`ewl_init()` takes care of all the dirty work of initializing the other required libs, abstracting all that away from the programmer into a simple interface.

The call to mk_gui will set up the main window and any content required.

The call to `ewl_main()` sets up the main processing loop, and upon exit handles all of the applications required shutdown, hence there is no shutdown call from our main routine.

Example 11.3. mk_gui: creating the window

```
/* build the main gui */
static void mk_gui(void) {
    Ewl_Widget *box = NULL, *menu_bar = NULL;
    Ewl_Widget *text_area = NULL, *scroll = NULL;

    /* create the main window */
    main_win = ewl_window_new();
    ewl_window_title_set(EWL_WINDOW(main_win), PROG);
    ewl_window_name_set(EWL_WINDOW(main_win), PROG);
    ewl_window_class_set(EWL_WINDOW(main_win), PROG);

    ewl_object_size_request(EWL_OBJECT(main_win), 200, 300);
    ewl_object_fill_policy_set(EWL_OBJECT(main_win), EWL_FLAG_FILL_FILL);

    ewl_callback_append(main_win, EWL_CALLBACK_DELETE_WINDOW, destroy_cb, NULL);
    ewl_widget_show(main_win);
```

The first thing we need to do to get our app off of the ground is to create the main application window. This is done through the call to `ewl_window_new()`. Once we have the window we can continue to set the title (as will appear in the WM bar on top of the app), name and class of the window.

Once the default information is set for the window we request a default size for the window to be 200x300 through the call to `ewl_object_size_request()`. Along with the default size we could have set a minimum and maximum size for the window through the calls to `ewl_object_minimum_size_set()` and `ewl_object_maximum_size_set()` size. But as this is not required for this application they are left out.

The final setup of the window is done by setting the fill policy with `ewl_object_fill_policy_set()`. This sets how Ewl will pack widgets into the window, with a possible values of:

EWL_FLAG_FILL_NONE	Do not fill or shrink in either direction
EWL_FLAG_FILL_HSHRINK	Shrink horizontally
EWL_FLAG_FILL_VSHRINK	Shrink vertically
EWL_FLAG_FILL_SHRINK	Shrink both horizontally and vertically
EWL_FLAG_FILL_HFILL	Fill horizontally
EWL_FLAG_FILL_VFILL	Fill vertically
EWL_FLAG_FILL_FILL	Fill both horizontally and vertically
EWL_FLAG_FILL_ALL	Shrink and Fill at the same time

After all the window properties are defined a callback to catch the destruction of the main window is attached with `ewl_callback_append()`. The function `destroy_cb()` will be called if someone requests the window to be destroyed in some fashion.

We show the main with with a call to `ewl_widget_show()`. If `ewl_widget_show()` is not called nothing would appear on the screen. All widgets are hidden until they are explicitly shown. The opposite to this is `ewl_widget_hide()` which will remove a widget from the screen.

Example 11.4. The main container

```
/* create the main container */
box = ewl_vbox_new();
ewl_container_child_append(EWL_CONTAINER(main_win), box);
ewl_object_fill_policy_set(EWL_OBJECT(box), EWL_FLAG_FILL_FILL);
ewl_widget_show(box);
```

We could pack all of our widgets into the main window itself, but that could cause problems later if we wanted to change things easily, so instead we create a box inside the main window to hold all of our widgets.

This is done by creating a vertical box with `ewl_vbox_new()`. The box is then taken and appended to the windows list of children with `ewl_container_child_append()`. After attaching to the window we set the fill policy to fill both horizontal and vertical with `ewl_object_fill_policy_set()`, and show the widget with `ewl_widget_show()`.

The order you put your widgets into the containers will affect the way that the application is displayed. The first widget packed will be the first widget displayed. Since we specified a vertical box we will start by packing the widgets from the top to the bottom of our display.

Example 11.5. Create the menu bar

```
/* create the menu bar */
menu_bar = ewl_hbox_new();
ewl_container_child_append(EWL_CONTAINER(box), menu_bar);
ewl_object_fill_policy_set(EWL_OBJECT(menu_bar), EWL_FLAG_FILL_HSHRINK);
```

```
ewl_object_alignment_set(EWL_OBJECT(menu_bar), EWL_FLAG_ALIGN_LEFT);
ewl_box_spacing_set(EWL_BOX(menu_bar), 4);
ewl_object_padding_set(EWL_OBJECT(menu_bar), 5, 5, 5, 5);
ewl_widget_show(menu_bar);
```

The first widget we put into place is the menu bar. We will place the actual contents into the menu bar later after some of the other widgets are created but we need to put the bar itself into place first.

The calls are the same as many you have seen before, appending ourselves to our parent, setting our fill policy, showing the widget. The ones not seen before include `ewl_object_alignment_set()`, this will set how the widget is aligned within its container. In this case we are using EWL_FLAG_ALIGN_LEFT, but could have used on of the other available alignments including:

- EWL_FLAG_ALIGN_CENTER
- EWL_FLAG_ALIGN_LEFT
- EWL_FLAG_ALIGN_RIGHT
- EWL_FLAG_ALIGN_TOP
- EWL_FLAG_ALIGN_BOTTOM

So the menu will be aligned with the left side of the main box.

We then specify the spacing of items inside the menu box. This will give a little more space between our menu items and is done with `ewl_box_spacing_set()`. After changing the space we change the padding around the box as a whole with the call to `ewl_object_padding_set()`, this will increase the amount of space left around the object as a whole.

Example 11.6. Create the scrollpane

```
/* create the scrollpane */
scroll = ewl_scrollpane_new();
ewl_container_child_append(EWL_CONTAINER(box), scroll);
ewl_object_fill_policy_set(EWL_OBJECT(scroll), EWL_FLAG_FILL_FILL);
ewl_scrollpane_hscrollbar_flag_set(EWL_SCROLLPANE(scroll),
                                   EWL_SCROLLPANE_FLAG_AUTO_VISIBLE);
ewl_scrollpane_vscrollbar_flag_set(EWL_SCROLLPANE(scroll),
                                   EWL_SCROLLPANE_FLAG_AUTO_VISIBLE);
ewl_widget_show(scroll);
```

The scrollpane is going to be the parent of our text object. The scrollpane provides us with all the magic to handle the scrollbars and the scrolling itself.

The scrollpane is created with a call to `ewl_scrollpane_new()`, and we then proceed to attach the scrollpane to the main box, and set its fill policy.

The ewl_scrollpane_[hv]scrollbar_flag_set() calls tell Ewl how the scrollbars should behave. The possible values are:

- EWL_SCROLLPANE_FLAG_NONE
- EWL_SCROLLPANE_FLAG_AUTO_VISIBLE
- EWL_SCROLLPANE_FLAG_ALWAYS_HIDDEN

Once the scrollbars are setup we tell Ewl to show the widget.

Example 11.7. Create the text area

```
/* create the text area */
text_area = ewl_text_new("");
ewl_container_child_append(EWL_CONTAINER(scroll), text_area);
ewl_object_padding_set(EWL_OBJECT(text_area), 1, 1, 1, 1);
ewl_widget_show(text_area);
```

The text area will be responsible for holding the text we display in our viewer. The widget is created with a simple call to `ewl_text_new()`. This will cause the text area to be created, but with the actual text blank. As with the menu bar we increase the padding around the text area to provide a bit of space from the edge of the text to any other elements.

Example 11.8. Add menu contents

```
/* create the menu */
{
    Ewl_Widget *file_menu = NULL, *item = NULL;

    /* create the file menu */
    file_menu = ewl_imenu_new(NULL, "file");
    ewl_container_child_append(EWL_CONTAINER(menu_bar), file_menu);
    ewl_widget_show(file_menu);

    /* add the open entry to the file menu */
    item = ewl_menu_item_new(NULL, "open");
    ewl_container_child_append(EWL_CONTAINER(file_menu), item);
    ewl_callback_append(item, EWL_CALLBACK_SELECT, file_menu_open_cb,
                                                            text_area);
    ewl_widget_show(item);

    /* add the quit entry to the file menu */
    item = ewl_menu_item_new(NULL, "quit");
    ewl_container_child_append(EWL_CONTAINER(file_menu), item);
    ewl_callback_append(item, EWL_CALLBACK_SELECT, destroy_cb, NULL);
    ewl_widget_show(item);
}
```

Now that the text area is created we can proceed to create the menu entries. I've done this inside its own block to limit the number of declarations at the top of the function, this isn't required for any reason.

The menu is created with a call to `ewl_imenu_new()`. This takes two parameters, the first is the image to display with this menu, in this case NULL, being no image. The second parameter is the name of the menu as will appear in the menu bar.

Once the menu is created we can then proceed to add entries to the menu through a call to `ewl_menu_item_new()`. This again takes two parameters, the icon to display beside this entry in the menu, and the name as it will appear in the menu.

As the items are added to the menu we make a call to `ewl_callback_append()` to attach to the EWL_CALLBACK_SELECT call. The given function will be executed when the use clicks on the menu entry. In the "open" case we have passed the text_area to the open callback to allow us to easily

modify its contents.

Other menus could have been added in the same fashion as this, but for this application only one menu is required.

Example 11.9. Attach callbacks

```
    ewl_callback_append(main_win, EWL_CALLBACK_KEY_UP, key_up_cb, scroll);
}
```

Once everything is setup on the main window we attach the callbacks we wish to receive. In this case we are attaching ourselves to the EWL_CALLBACK_KEY_UP callback. We don't need to do anything to have mouse wheel scrolling in the scrollpane as it is configured into the scrollpane itself.

Example 11.10. Destroy callback

```
/* destroy the app */
static void destroy_cb(Ewl_Widget *win, void *ev, void *data) {
    ewl_widget_destroy(win);
    ewl_main_quit();
}
```

When the main window is closed we destroy the widget that is the main window through a call to `ewl_widget_destroy()`. After the window is destroyed we tell Ewl that we wish to exit by calling `ewl_main_quit()`. This will cause Ewl to halt the main processing loop and the previous call to `ewl_main()` will return.

Example 11.11. File menu open callback

```
/* the file menu open button callback */
static void file_menu_open_cb(Ewl_Widget *win, void *ev, void *data) {
    Ewl_Widget *fd = NULL;
    Ewl_Widget *box = NULL;
    Ewl_Widget *home = NULL;

    /* create the file dialog window */
    fd_win = ewl_window_new();
    ewl_window_title_set(EWL_WINDOW(fd_win), PROG " -- file dialog");
    ewl_window_name_set(EWL_WINDOW(fd_win), PROG " -- file dialog");
    ewl_window_class_set(EWL_WINDOW(fd_win), PROG " -- file dialog");
    ewl_object_size_request(EWL_OBJECT(fd_win), 500, 400);
    ewl_object_fill_policy_set(EWL_OBJECT(fd_win),
                EWL_FLAG_FILL_FILL | EWL_FLAG_FILL_SHRINK);
    ewl_callback_append(fd_win, EWL_CALLBACK_DELETE_WINDOW,
                                destroy_filedialog_cb, NULL);
    ewl_widget_show(fd_win);

    /* fd win container */
    box = ewl_vbox_new();
    ewl_container_child_append(EWL_CONTAINER(fd_win), box);
```

```
    ewl_object_fill_policy_set(EWL_OBJECT(box),
                EWL_FLAG_FILL_FILL | EWL_FLAG_FILL_SHRINK);
    ewl_widget_show(box);

    /* the file dialog */
    fd = ewl_filedialog_new(EWL_FILEDIALOG_TYPE_OPEN);
    ewl_callback_append(fd, EWL_CALLBACK_VALUE_CHANGED, open_file_cb, data);
    ewl_container_child_append(EWL_CONTAINER(box), fd);

    /* add a home button */
    home = ewl_button_new("Home");
    ewl_callback_append(home, EWL_CALLBACK_CLICKED, home_cb, fd);
    ewl_object_fill_policy_set(EWL_OBJECT(home), EWL_FLAG_FILL_HFILL);
    ewl_container_child_append(EWL_CONTAINER(fd), home);
    ewl_widget_show(home);

    ewl_widget_show(fd);
}
```

If a use clicks on the open entry in the file menu, the `file_menu_open_cb()` will be executed. When that happens we need to create the file dialog for the use to select the file to view.

In the same fashion as the main window, we create a window to hold the file dialog and set its title, name and class. We request a default size, set its fill policy and attach a callback to handle the destruction of the window itself. We then attach a simple box into the window to hold the file dialog.

Once the window is setup, we make the call to create the file dialog. This is done with a call to `ewl_filedialog_new()`, specifying the type of file dialog we wish to create. In this case we want a dialog to allow us to open a file, so we specify EWL_FILEDIALOG_TYPE_OPEN. We could have specified EWL_FILEDIALOG_TYPE_SAVE if we wished to use the dialog to save a file instead of open.

We then proceed to create an extra button to allow the user to navigate to their home directory with a single click. This is done by calling `ewl_button_new()` and packing the subsequent button into the file dialog itself.

Example 11.12. File dialog destroy callback

```
/* close the file dialog */
static void destroy_filedialog_cb(Ewl_Widget *win, void *ev, void *data) {
    ewl_widget_hide(win);
    ewl_widget_destroy(win);
}
```

When we need to get rid of the file dialog we remove the widget from the screen with a call to `ewl_widget_hide()`, and once it is no longer displayed we destroy the widget with a call to `ewl_widget_destroy()`.

Example 11.13. File dialog open button callback

```
/* the file dialog open button callback */
```

```
static void open_file_cb(Ewl_Widget *w, void *ev, void *data) {
    char *text = NULL;
    int *response = (int *)ev;

    switch (*response) {
        case EWL_RESPONSE_OPEN:
            text = read_file(ewl_filedialog_file_get(EWL_FILEDIALOG(w)));
            break;

        case EWL_RESPONSE_CANCEL:
            break;
    }

    if (text) {
        ewl_text_text_set(EWL_TEXT(data), text);
        free(text);
    }
    text = NULL;

    ewl_widget_hide(fd_win);
}
```

This callback will be executed when the user clicks the open button in the file dialog, or if the user double clicks on a file in a directory. The event passed (the ev parameter) will be the full path to the file that the use has selected.

In our case, we take that file and pass it to the function to read in the file and return the text of the file. Then using that text, as long as it is defined, we call `ewl_text_text_set()` which will set the text of the given text object.

As the user has now finished their selection they file dialog is hidden from view.

Example 11.14. File dialog home button callback

```
/* the fd home button is clicked */
static void home_cb(Ewl_Widget *win, void *ev, void *data) {
    char *home = NULL;
    Ewl_Filedialog *fd = (Ewl_Filedialog *)data;

    home = getenv("HOME");
    if (home)
        ewl_filedialog_path_set(fd, home);
}
```

If the user clicks on the "Home" button in the file dialog we want to display the contents of their home directory to them. We set the file dialog as the user data to the callback, so we cast that back to the Ewl_Filedialog and grabbing the home directory from the environment. The call to `ewl_filedialog_path_set()` changes the current directory the file dialog is displaying to be the users home directory.

Example 11.15. Read text file

```
/* read a file */
static char *read_file(char *file) {
    char *text = NULL;
    FILE *f = NULL;
    int read = 0, st_ret = 0;
    struct stat s;

    f = fopen(file, "r");
    st_ret = stat(file, &s);

    if (st_ret != 0) {
        if (st_ret == ENOENT)
            printf("not a file %s\n", file);
        return NULL;
    }

    text = (char *)malloc(s.st_size * sizeof(char));
    read = fread(text, sizeof(char), s.st_size, f);

    fclose(f);
    return text;
}
```

This is just a simple routine to take the given file, open it and read its contents into memory. Probably not the best idea for a real app, but is sufficient for this example program.

Example 11.16. Key press callback

```
/* a key was pressed */
static void key_up_cb(Ewl_Widget *win, void *ev, void *data) {
    Ewl_Event_Key_Down *e = (Ewl_Event_Key_Down *)ev;
    Ewl_ScrollPane *scroll = (Ewl_ScrollPane *)data;

    if (!strcmp(e->keyname, "q")) {
        destroy_cb(win, ev, data);

    } else if (!strcmp(e->keyname, "Left")) {
        double val = ewl_scrollpane_hscrollbar_value_get(EWL_SCROLLPANE(scroll));
        double step = ewl_scrollpane_hscrollbar_step_get(EWL_SCROLLPANE(scroll));

        if (val != 0)
            ewl_scrollpane_hscrollbar_value_set(EWL_SCROLLPANE(scroll),
                                                val - step);

    } else if (!strcmp(e->keyname, "Right")) {
        double val = ewl_scrollpane_hscrollbar_value_get(EWL_SCROLLPANE(scroll));
        double step = ewl_scrollpane_hscrollbar_step_get(EWL_SCROLLPANE(scroll));

        if (val != 1)
            ewl_scrollpane_vscrollbar_value_set(EWL_SCROLLPANE(scroll),
                                                val + step);

    } else if (!strcmp(e->keyname, "Up")) {
        double val = ewl_scrollpane_vscrollbar_value_get(EWL_SCROLLPANE(scroll));
        double step = ewl_scrollpane_vscrollbar_step_get(EWL_SCROLLPANE(scroll));

        if (val != 0)
            ewl_scrollpane_vscrollbar_value_set(EWL_SCROLLPANE(scroll),
```

```
                                                                    val - step);

    } else if (!strcmp(e->keyname, "Down")) {
        double val = ewl_scrollpane_vscrollbar_value_get(EWL_SCROLLPANE(scroll));
        double step = ewl_scrollpane_vscrollbar_step_get(EWL_SCROLLPANE(scroll));

        if (val != 1)
            ewl_scrollpane_vscrollbar_value_set(EWL_SCROLLPANE(scroll),
                                                                    val + step);
    }
}
```

The `key_up_cb()` will be called whenever the user releases a key on the keyboard. The callback will receive an Ewl_Event_Key_Down structure containing the information on the key press itself. In our case we just need the keyname entry which is the name of the key that was pressed.

If the user presses the "q" key we just call the destroy callback and be done with it.

The "Left", "Right", "Up" and "Down" relate the the arrow keys on the users keyboard. If any of these keys are pressed we force the scrollpane to scroll in a specified direction.

In order to manipulate the scrollpane we need to know where it currently is in the file and the amount of distance each increment/decrement should travel. Luckily Ewl makes this easy. The call to ewl_scrollpane_[hv]scrollbar_value_get() will return the current value of the scroll bar. This is a double value in the range of [0, 1] inclusive. A value of 0 meaning the scrollbar is at the top and a value of 1 being at the bottom. The left and right work the same way, but 0 is absolute left and 1 is absolute right.

The second piece of information is obtained through the call to ewl_scrollpane_[hv]scrollbar_step_get(). The step is the distance the scrollpane will travel with one iteration. So using these two values we can then move the scrollbar in the correct direction with the call to ewl_scrollpane_[hv]scrollbar_value_set().

Example 11.17. Compilation

```
zero@oberon [ewl_intro] -< gcc -Wall -o ewl_text main.c \
  `ewl-config --cflags --libs`
```

Compiling an ewl app is as simple as calling ewl-config and getting the --cflags and --libs.

Thats it. With that you should have a fully functioning Ewl application including menus, a file dialog and a text area with horizontal and vertical scrollbars. This example is just scratching the surface of the power contained inside of the Ewl toolkit with many other types of widgets available for use.

Chapter 12. Evoak

Evoak is a canvas server. This is similar to an X server that serves out a display and graphics operations. Evoak serves out a single canvas to be shared by multiple applications (clients) allowing each client to manipulate its set of objects on the canvas.

Recipe: Evoak hello client

dan 'dj2' sinclair <zero@perplexity.org>

This recipe is a very simple introduction to the world of Evoak programming. Building on grand traditions before, it displays the Canadian version of 'Hello World' on an Evoak canvas.

Example 12.1. Includes and pre-declarations

```
#include <Evoak.h>
#include <Ecore.h>

static unsigned int setup_called = 0;

static int canvas_info_cb(void *, int, void *);
static int disconnect_cb(void *, int, void *);
static void setup(Evoak *);
```

We need to include the Evoak header obviously, and the Ecore header is required to have access to the callback functions.

Example 12.2. main

```
int main(int argc, char ** argv) {
    Evoak *ev = NULL;

    if (!evoak_init()) {
        fprintf(stderr, "evoak_init failed");
        return 1;
    }

    ecore_event_handler_add(EVOAK_EVENT_CANVAS_INFO, canvas_info_cb, NULL);
    ecore_event_handler_add(EVOAK_EVENT_DISCONNECT, disconnect_cb, NULL);

    ev = evoak_connect(NULL, "evoak_intro", "custom");

    if (ev) {
        ecore_main_loop_begin();
        evoak_disconnect(ev);
    }

    evoak_shutdown();
    return 0;
}
```

Evoak needs to be setup initially with a call to `evoak_init`. This will setup all the internal libraries and requirements for Evoak.

As long as Evoak initializes correctly, we hook up two callbacks, the first is for canvas information and the second is for if we get disconnected from the Evoak server. These will be discussed later when the actual callbacks are displayed.

Once the callbacks are in place we need to connect to the Evoak canvas server. This is done through the call to `evoak_connect`. The parameters to `evoak_connect` are: the server to connect to, the client name and the client class. If the first argument is NULL, as in this example, the default Evoak server is connected too. The second argument to `ecore_connect` is the clients name, this value should be something unique as it will be used to distinguish the client from other clients. The final argument, the class, is the type of client, some of the possible values being: "background", "panel", "application" or "custom".

If the call to `evoak_connect` fails a NULL value will be returned. So, as long as we receive a Evoak object back, we start the main ecore loop. When ecore finishes we call `evoak_disconnect` to disconnect from the Evoak server.

We finish off by calling `evoak_shutdown` to clean up after ourselves.

Example 12.3. Canvas info callback

```
static int canvas_info_cb(void *data, int type, void *ev) {
    Evoak_Event_Canvas_Info *e = (Evoak_Event_Canvas_Info *)ev;

    if (!setup_called) {
        setup_called = 1;
        setup(e->evoak);
    }
    return 1;
}
```

A canvas info callback will be made when our client receives information about the Evoak server canvas. Wit this canvas information we can then proceed to setup our clients contents. This is contained inside of a setup_called flag as we only want to initialize once.

Example 12.4. Disconnect callback

```
static int disconnect_cb(void *data, int type, void *ev) {
    printf("disconnected\n");
    ecore_main_loop_quit();
    return 1;
}
```

The disconnect callback will the called when the client has been disconnected from the Evoak server. In this case, the simple solution of exiting is used.

Example 12.5. setup routine

```
static void setup(Evoak *ev) {
    Evoak_Object *o = NULL;

    evoak_freeze(ev);

    o = evoak_object_text_add(ev);
    evoak_object_text_font_set(o, "Vera", 12);
    evoak_object_color_set(o, 255, 0, 0, 255);
    evoak_object_text_text_set(o, "Hello Evoak, eh.");
    evoak_object_show(o);

    evoak_thaw(ev);
}
```

The setup routine will be called once to setup the display of our client. For this example, the client only draws the text 'Hello Evoak, eh'.

The first thing we do is call `evoak_freeze`, this should keep us from getting any unwanted callbacks while we are setting up our interface. At the end of the function we call the reciprocal `evoak_thaw` to undo the previous freeze.

We then proceed to create a text object with `evoak_object_text_add` and taking that text object, set the font, colour and text contents with the calls to, `evoak_object_text_font_set`, `evoak_object_color_set`, and `evoak_object_text_text_set` respectively.

Example 12.6. Compilation

```
zero@oberon [evoak_intro] -> gcc -o hello_evoak main.c \
  `evoak-config --cflags --libs`
```

As with many of the other EFL based libraries, compiling an Evoak application is as simple as calling the evoak-config program and getting the --cflags and --libs contents.

Thats it, this was a really simple introduction into Evoak and the surface remains unscratched as to the potential available for client applications.

Chapter 13. Emotion

Emotion is a video & media object library designed to interface with Evas and Ecore to provide autonomous "video" and "audio" objects that can be moved, resized and positioned like any normal object, but instead they can play video and audio and can be controlled from a high-level control API allowing the programmer to quickly piece together a multi-media system with minimal work. Emotion provides a modular decoder layer system where a decoder module can be plugged in separately to provide decoding resources for Emotion. Emotion currently has 1 decoder module that uses XINE as the decoder, allowing it to play DVD's, MPEG's, AVI's, MOV's, WMV's and much more. Its test program is already a useful DVD player (without a lot of the fancy control interface) and can play multiple video streams with semi-translucency and more.

Recipe: Quick DVD player with Emotion

Carsten 'rasterman' Haitzler <raster@rasterman.com>

To show how easy Emotion makes it to put a video file, DVD, VCD or other media content within a canvas take a look at the following program. It is a complete DVD player, but very simple. It has limited mouse controls, no handling of aspect ratio changes, etc. This is all of 55 lines of C code.

All code in this and the next recipie can be compiled using:

Example 13.1. Compiling

```
$ gcc player.c -o player `emotion-config --cflags --libs`
```

Example 13.2. DVD player in 55 lines of code

```
#include <Evas.h>
#include <Ecore.h>
#include <Ecore_Evas.h>
#include <Emotion.h>

Evas_Object *video;

/* if the window manager requests a delete - quit cleanly */
static void
canvas_delete_request(Ecore_Evas *ee)
{
    ecore_main_loop_quit();
}

/* if the canvas is resized - resize the video too */
static void
canvas_resize(Ecore_Evas *ee)
{
    Evas_Coord w, h;

    evas_output_viewport_get(ecore_evas_get(ee), NULL, NULL, &w, &h);
    evas_object_move(video, 0, 0);
```

```
    evas_object_resize(video, w, h);
}

/* the main function of the program */
int main(int argc, char **argv)
{
    Ecore_Evas *ee;

    /* create a canvas, display it, set a title, callbacks to call on resize */
    /* or if the window manager asks it to be deleted */
    ecore_evas_init();
    ee = ecore_evas_software_x11_new(NULL, 0, 0, 0, 800, 600);
    ecore_evas_callback_delete_request_set(ee, canvas_delete_request);
    ecore_evas_callback_resize_set(ee, canvas_resize);
    ecore_evas_title_set(ee, "My DVD Player");
    ecore_evas_name_class_set(ee, "my_dvd_player", "My_DVD_Player");
    ecore_evas_show(ee);

    /* create a video object */
    video = emotion_object_add(ecore_evas_get(ee));
    emotion_object_file_set(video, "dvd:/");
    emotion_object_play_set(video, 1);
    evas_object_show(video);

    /* force an initial resize */
    canvas_resize(ee);

    /* run the main loop of the program - playing, drawing, handling events */
    ecore_main_loop_begin();

    /* if we exit the main loop we will shut down */
    ecore_evas_shutdown();
}
```

Now we have a very simple introduction to Emotion. This snippet of code can easily be expanded to work with any media format that emotion supports, as well as dealing with aspect ratios, keyboard navigation, and more.

Recipe: Expanded Media player with Emotion

Carsten 'rasterman' Haitzler <raster@rasterman.com>

Expanding on our previous recipie, we can make emotion handle being resized properly (which maintaining aspect ration),

Example 13.3. Emotion Media Player

```
#include <Evas.h>
#include <Ecore.h>
#include <Ecore_Evas.h>
#include <Emotion.h>

Evas_Object *video;

/* if the window manager requests a delete - quit cleanly */
static void
canvas_delete_request(Ecore_Evas *ee)
```

```
{
    ecore_main_loop_quit();
}

/* if the canvas is resized - resize the video too */
static void
canvas_resize(Ecore_Evas *ee)
{
    Evas_Coord w, h;

    evas_output_viewport_get(ecore_evas_get(ee), NULL, NULL, &w, &h);
    evas_object_move(video, 0, 0);
    evas_object_resize(video, w, h);
}

/* the main function of the program */
int main(int argc, char **argv)
{
    Ecore_Evas *ee;

    /* create a canvas, display it, set a title, callbacks to call on resize */
    /* or if the window manager asks it to be deleted */
    ecore_evas_init();
    ee = ecore_evas_software_x11_new(NULL, 0, 0, 0, 800, 600);
    ecore_evas_callback_delete_request_set(ee, canvas_delete_request);
    ecore_evas_callback_resize_set(ee, canvas_resize);
    ecore_evas_title_set(ee, "My DVD Player");
    ecore_evas_name_class_set(ee, "my_dvd_player", "My_DVD_Player");
    ecore_evas_show(ee);

    /* create a video object */
    video = emotion_object_add(ecore_evas_get(ee));
    emotion_object_file_set(video, "dvd:/");
    emotion_object_play_set(video, 1);
    evas_object_show(video);

    /* force an initial resize */
    canvas_resize(ee);

    /* run the main loop of the program - playing, drawing, handling events */
    ecore_main_loop_begin();

    /* if we exit the main loop we will shut down */
    ecore_evas_shutdown();
}
```