The EFL Cookbook

Various
Edited by Ben technikolor Rockwood

The EFL Cookbook

by Various and Ben technikolor Rockwood

Stuff.

Table of Contents

List of Examples

Chapter 1. Introduction

This book.......

More intro.

Chapter 2. Imlib2

Imlib2 provides a powerful engine for image manipulation and rendering.

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 what the characteristics of your XServer's display are, what depth or what magic visuals etc, 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. Here are some screen shots of a demo Evas application to show the rendering output in different situations

Recipe: Key Binds, using EVAS Key Events

This is my recipe!

Chapter 4. Ecore

Ecore provides a powerful event handling and modularized abstraction layer which ties and bind your applications various components together in a nearly seemless manner.

Recipe: Ecore Config Introduction

dan 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(PROGRAM_NAME). 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 Ipc Introduction

dan 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 4.4. 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 4.5. 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 4.6. 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 4.7. 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 4.8. 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.

Example 4.9. 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 4.10. 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 sign handler will be explained with its code.

Example 4.11. 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 4.12. 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 4.13. 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 4.14. Ecore_Ipc server: sig_exit callback

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

Example 4.15. 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 4.16. 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.

Chapter 5. EDB & EET

The libs.

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 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.

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(evas). 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 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 libary 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.

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 specifes 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 edjefile 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 specifes 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 seached 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. Etox & Estyle

Etox provides a power library for manipulation and layout of text in your EVAS/EFL application, which Estyle provides a veritile text stylization library and toolset that can be used with or without Etox.

Chapter 8. EWD

EWD is a library providing thread-safe basic data structures like hashes, lists and trees.

Chapter 9. Edje EDC

Edje Data Collections (EDC) source files allow for easy creation of rich and powerful interface designs...

Chapter 10. Edje

Edje is a complex graphical design and layout library.

Chapter 11. EWL

The Enlightenment Widget Library provides a full blown widget kit similar in feel to GTK, QT or MOTIF, but based apon the power of the Enlightenment Foundation Libraries.