

[www.enlightenment.org /docs/themes/start](http://www.enlightenment.org/docs/themes/start)

Unknown Title

EDC File Basics

Let's start with a "Hello World" example. This is going to be in verbose syntax to make it as easy as possible to break down before we start making it more compact. Start with a source "EDC" file like below:

[example.edc](#)

```
collections {
  group {
    name: "example";
    parts {
      part {
        name: "background";
        type: RECT;
        description {
          state: "default";
          color: 64 64 64 255;
        }
      }
      part {
        name: "label";
        type: TEXT;
        description {
          state: "default";
          color: 255 255 255 255;
          text {
            font: "Sans";
            size: 10;
            text: "Hello World";
          }
        }
      }
    }
  }
}
```

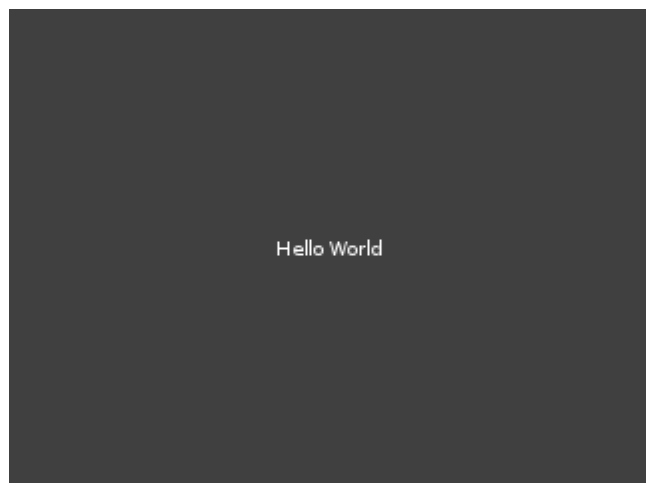
Compiling it is easy. The compiled EDJ file is portable. It can be used on any OS and any architecture. It's basically a structured archive of all of the source material packed into a single file that is compressed and usable "as-is" at runtime without installation or unpacking.

```
edje_cc example.edc
```

Now a quick way to see your result is with the `edje_player` tool that will load a specific group in a file or just use the first it finds. In our case here we have only a single group so it should load just this group within the collections and let us see it:

```
edje_player example.edj
```

You now should see something like this in a window:



The window will be resizable, so resize it to see what happens when your group is resized too. One of the key ideas behind Edge is to be able to make both resizable AND scaleable components that can adapt to sizing needs and UI/DPI scaling needs.

The Anatomy of an Edge File

An Edge file is compiled from a combination of source “EDC” files and other data (images, fonts, sound samples, ...) into a single stand-alone binary “EDJ” file that is used at runtime. These files are, by default, de-compileable again back to source with the `edje_decc` tool. These files do not execute code, and are system-independent. That may use various compression schemes based on options to `edje_cc`. The source EDC files are passed through a C pre-processor so they can use CPP directives such as `#include`, `#define`, `#ifdef`, the macros defined by `#define`, etc. and this can be used to make your EDC files more maintainable by being able to generate content from smaller macros or included files, split your content up into many files that you `#include` together and so on. But let's get into the actual structure.

An EDC file will look a bit like a mix of JSON and a C language structure definition. Everything has some kind of parent/child hierarchy with parent sections containing child sections and so on. It is also possible to skip a section and use periods (.) to declare a parent and child relationship directly on a line, so You could express:

[example.edc](#)

```
text {  
    font: "Sans";
```

```
    size: 10;  
    text: "Hello World";  
}
```

As

[example.edc](#)

```
text.font: "Sans";  
text.size: 10;  
text.text: "Hello World";
```

You can also remove whitespace before keywords and after statement delimiters (;), so the above could also be done expressed as:

[example.edc](#)

```
text { font: "Sans"; size: 10; }  
text.text: "Hello World";
```

You should take advantage of white-space compression to make your EDC files less lengthy and thus easier to read and maintain. So let's take the above hello world example and make it a bit more compact before we continue:

[example.edc](#)

```
collections {  
  group { name: "example";  
    parts {  
      part { name: "background"; type: RECT;  
        description { state: "default";  
          color: 64 64 64 255;  
        }  
      }  
      part { name: "label"; type: TEXT;  
        description { state: "default";  
          color: 255 255 255 255;  
          text { font: "Sans"; size: 10; }  
          text.text: "Hello World";  
        }  
      }  
    }  
  }  
}
```

We will use this kind of white-space compression from now on to keep things more compact. Remember that the same declaration is able to be expressed in many different ways, so don't get confused when you see the same thing expressed with different white space or with different parent/child syntax as they are

equivalent.

Edje files generally contain several major sections. `collections` is the section where a series of group sections are declared. Each group is accessible by its name via Edje or Elementary [API](#). A single EDJ file may contain 0, 10, 100, or 1000 groups ... or more. It depends on how much you pack in. The names of such groups is a free-form string, but generally there is a convention to use something like a file path so `group/children/thing` would be common. So a very basic EDC source file may begin as:

[example.edc](#)

```
collections {
  group { name: "example";
  }
}
```

We have a group in our collections, and this group has nothing in it. What exactly is a group then? It's a group of child objects, programs that respond to signals/events and other layout logic that define a single graphical element. Code can ask the Edje library to use this defined group from that file such as:

[example.c](#)

```
obj = elm_layout_add(win);
elm_layout_file_set(obj, "example.edj", "example");
elm_win_resize_object_add(win, obj);
evas_object_show(obj);
```

Elementary and Enlightenment have code like this everywhere (or the lower level versions of it). Almost every Elementary widget actually creates such layout/Edje objects internally and sources them from theme files such as `default.edj` that ships with EFL. Every single widget will likely go back to this file and figure out which bit of data maps to this name, load it and instantiate the objects based on the design data in the EDJ file. This is mostly transparent to a user or programmer, other than when they wish to change the look of their UI ... they then can make their own themes and override the standard search for a theme with their own.

This is how any application may provide a custom look/feel of its own (it's own Themes), as well as added layout elements that are not normal widgets, built out of such data files and much much more. Enlightenment even will only accept EDJ files as wallpapers because then it then doesn't need to know how to tile, scale or otherwise lay out the wallpaper content. Edje takes care of this. The Wallpaper import dialog is simply generating a sample EDC file that includes the selected image file with the layout options selected. An added bonus is that a single EDJ file can include multiple resolutions of an image to optimally load the best one, scale and adapt on the fly and even animate.

So learning how Edje works is the key to the kingdom of Enlightenment. Master Edje and your UI can be changed from clean modern flat styles through to skeumorphic madness and anything else your imagination can come up with. So let's continue with the basic anatomy.

Every group will likely have parts, possibly programs and may even have it's own `images` section as

well as a few others. The `parts` section defines a series of parts in stacking order from bottom to top that comprise the group object. So a small step forward with our example may become something like this:

[example.edc](#)

```
collections {
  group { name: "example";
    parts {
      part { name: "background"; type: RECT;
      }
      part { name: "label"; type: TEXT;
      }
    }
  }
}
```

We have 2 parts, one named “background” that is of type `RECT` (a basic rectangle) and one “label” that is of type `TEXT` (a simple single line text object with no markup etc.). You generally will want names for any part you wish to later access or address by changing its state or otherwise accessing it from outside Edje (i.e. from the calling application). Come up with names that are memorable. In EFL we generally namespace our names. If the part name is meant to be accessed from outside the group (e.g. from an app), then we will namespace something like “e.text” or “elm.text” etc. but using a dot (.) with a prefix of the namespace then following namespaces. A sign that a part is “official” and is expected to exist or be interacted with is that it has a dot in its name with a namespace.

But these parts are pretty useless as Edje has no idea of their description - what color are they? What Font used? What sized font? We need to provide at LEAST a default description.

[example.edc](#)

```
collections {
  group { name: "example";
    parts {
      part { name: "background"; type: RECT;
        description { state: "default";
          color: 64 64 64 255;
        }
      }
      part { name: "label"; type: TEXT;
        description { state: "default";
          color: 255 255 255 255;
          text { font: "Sans"; size: 10; }
          text.text: "Hello World";
        }
      }
    }
  }
}
```

```
}  
}  
}
```

And now we have a basic example. The rectangle is a dim grey (color is in R G B A with values from 0 to 255 for the elements) normally, but you can also use html-style color notations like “#404040ff” instead of 64 64 64 255 like `color: “#404040ff”;`. We have a single line text label that uses the “Sans” font at size 10, is white in color and displays the text “Hello World” by default.
