Building Interfaces with Edje

The Edje Developers Guide

Ben Rockwood

Building Interfaces with Edje: The Edje Developers Guide

Ben Rockwood

A complete guide to designing interfaces using Edje and utilizing them effectively in your EVAS applications. Includes complete overview of Edje Data Collections (EDC) and Edje API, including sample code and reference material. Utilizing Edje effectively can simplify and streamline application development and make nearly any application completely skinable by even non-programmers.

Table of Contents

List of Figures

List of Tables

List of Examples

Chapter 1. Edje Overview

Introduction

Edje is a componant of the Enlightenment Foundation Libraries which abstracts interface design and functionality from the application code itself. An application using Edje is comprised of two componants, the code in the EVAS application which interacts with the interface, and a created EET which contains all the elements of the interface itself. These EET's are generated from an Edje EDC file in which diffrent parts of the interface are described and laid out, and how those parts interact with your application. This allows for the interface to be completely changed simply by creating a new EDC and generating from it the EET that your application can use. In more popular terms, Edje makes every application that uses it "skinable". Raster describes Edje as "an attempt to find a middleground between themeing and programming without turning the theme itself into just yet another program".

Edje Data Collections (EDC)

The Edje EDC contains every detail about your interface. An EDC is a simple text file that uses C like syntax. The file is broken into three distinct section: images, data, and collections. The image section contains a list of all the images your interface will use. When you compile/generate your EDC into an EET all the images specified will be loaded into the EET itself allowing you to distribute your interface (or skin if you prefer) as a single file. When specifying your images you can even specify the ammount you want the images to be compressed when adding them into the EET to keep your EETs small and portable. The collections section actually describes the interface, how it's laid out and how it will interact with your application code. Optionally, a third section, data, can contain arbitrary data in key-value pairs to provide data to your application.

The collections in an EDC are comprised of one or more groups. Each group contains parts and programs. A part is a single element of your interface, such as a text element, or a rectangle, or an image. Each part is thuroughly described for one or more states. For instance, an image part might actually have two images in it, each in a diffrent state, one for a normal state and one for a clicked (mouse down) state. A part may have as many states defined as you like.

These parts are then referenced in the programs. Programs are descriptions

about how the interface should respond to both the user and the application code itself. A program can accept interface event (such as mouse button 1 down), and then change the state of a part described earlier (change to state clicked) to create an effect. Programs can also emit signals to your application. In your application code you would define a callback for that event. Each program is concise and distinct. For example, to create an animated button, you would create 3 programs, one to change the image state from normal to clicked on a mouse down event, one to emit a signal to your application on the mouse down event, and yet another to change state back to normal on a mouse up event.

Because of the abstraction Edje provides, your applcation only needs to know the name of the EET to use, what signals it will recieve from the interface so that callbacks can be defined when that event is recieved, and what text parts in the interface can be modified. This allows maximum flexability in interface design, including the ability to offload interface to graphic designers and themers freeing the application coders, allowing users of the application to modify the interface without hacking or forking your project, and a much quicker prototyping and design tool than modifying your C application directly.

Chapter 2. A quick tour of Edje EDC

Whenever you design a graphical application you need to first determine what the application needs to do, and then you consider how it should look. When developing with Edje these two tasks are the seperate activities they should be. When determining how to design your interface, it's common to use GIMP as a layout tool, this is particularly helpful for designing your interface with Edje as you can see exactly how each element is going to be positioned and relate to other parts of your interface, which is espcially helpful when learning Edje. Next, it's time to actually build your real interface. This starts by creating a file typically labeled with a .edc extension. Once you've created your EDC, you will generate the EET that will be used by your application using edje_cc, the Edje Collection Compiler. Edje_cc will pack your EDC plus all of the images your interface will need into one tight compact EDC, and will even compress images if you specified compression in your EDC. This generated EET will be the file that your application code, using the Edje API, will access and interact with. This allows you to share or transfer themes by moving just that one file, the EET, and not your entire image structure, or a tarball/zip like nearly every other themeing platform on the planet. EET's can be previewed using the **edje** command as a way to debug and sample interfaces.

Before starting off with Edje on your own, you should look at the example in the Edje source tree. In the `edje/data` directory you will find a script named `e_logo.sh`. Run that script and an EET will be generated. The script simply runs **edje_cc** with the appropriate options to generate a valid EET. You can then preview the interface with the Edje command, **`edje e_logo.eet test`**. You can look at the EDC in the `edje/data/src/` directory, where you will also find another EDC named `test.edc`. The `test.edc` is a wonderful reference, as its constantly updated to reflect all avalible options for EDCs.

Writting your first EDC

Lets look at a small and simple EDC so we can explore its layout and options.

Example 2.1. A Simple Edje EDC

```
// Sample EDC
images {
        image,  "background.png" LOSSY 95;
}

collections {
   group {
      name, "test";
      min, 32 32;
      max, 1024 768;

      parts {
         part {
                name,   "background";
                type,   IMAGE;
                mouse_events, 0;

                description {
                        state, "default" 0.0;

                        rel1 {
                                relative, 0.0 0.0;
                                offset, 0 0;
                        }
                        rel2 {
                                relative, 1.0 1.0;
                                offset, -1 -1;
                        }
                        image {
                                normal, "background.png";
                        }
                }
        }

        part {
                name,   "button";
                type,   RECT;
                mouse_events, 1;

                description {
                        state, "default" 0.0;
                        min, 100 50;
                        max, 100 50;
                        align, 0.5 0.5;

                        color, 211 168 234 255;

                        rel1 {
```

```
                        relative, 0.0 0.0;
                        offset, 0 0;
                }
                rel2 {
                        relative, 1.0 1.0;
                        offset, -1 -1;
                }
        }
        description {
                state, "clicked" 0.0;
                min, 100 50;
                max, 100 50;
                align, 0.5 0.5;

                color, 170 89 214 255;

                rel1 {
                        relative, 0.0 0.0;
                        offset, 0 0;
                }
                rel2 {
                        relative, 1.0 1.0;
                        offset, -1 -1;
                }
        }
}
part {
        name,   "text";
        type,   TEXT;
        mouse_events, 0;

        description {
                state, "default" 0.0;

                rel1 {

                        relative, 0.0 0.0;
                        offset, 0 0;
                        to, "button";
                }
                rel2 {
                        relative, 1.0 1.0;
                        offset, -1 -1;
                        to, "button";
                }
                text {
                        text, "Press Me";
                        font, "redensek";
                        size, 14;
```

```
                                align, 0.5 0.5;
                        }
                }
        }
   } /* Close Parts */

   programs {

        program {
                name, "button_click";
                signal, "mouse,down,1";
                source, "button";
                action, STATE_SET "clicked" 0.0;
                target, "button";
                after,  "do_me";
        }

        program {
                name, "button_unclick";
                signal, "mouse,up,1";
                source, "button";
                action, STATE_SET "default" 0.0;
                target, "button";
                after,  "stop_doing_me";
        }

        program {
                name, "do_me";
                signal, "*";
                source, "button_click";
                action, SIGNAL_EMIT "PANTS ON" "button";
                in, 1.0 0.0;
                after, "do_me";
        }

        program {
                name, "stop_doing_me";
                signal, "*";
                source, "button_unclick";
                action, ACTION_STOP;
                target, "do_me";
        }

   } /* Close Prog */
 } /* Close Group */
} /* Close Coll */
```

The first thing to notice is that the syntax is very C like. The second is that EDCs have two major sections: images and collections. And lastly, that every directive is in the form: keyword, arg1 arg2 ...;. Now lets talk thru this EDC section by section.

The images section is the first section to examine. It simply contains a list of images that we with to include in our interface. Each image will have a line describing how to store it, in the form: image, "filename" STORAGE_METHOD;. The storage method is a description of how Edje should store the file in the generated EET. Valid storage methods are COMP, RAW, and LOSSY. RAW stores the image in raw format, note that this doesn't mean unmodified format, a RAW stored PNG will be uncompressed and stored as the raw image, thus you tidy 1K PNG may be stode as an 8K raw image. The COMP method will use lossless compression on your image. And the LOSSY method will use a lossy compression on your image. The LOSSY method is followed by an integer between 0 and 100, defining the quality level.

The next section is collections. This section contains one or more groups. Each group contains a complete interface. In this way, you could create two groups in your EDC each having a complete interfaces, for either diffrent parts of your applications interface or diffrent looks for your one interface, all in one EET for easy distribution. Each group is given a name, in this case "test". Any time you reference your EET you will specify both the EET file name, and the group to use so choose your name descriptively. The next two lines are the min and max size of our interface in pixels. The arguments to min and max are horizontal size then the verital size. So our interface case a minimum size of 32x32 and maximum of 1024x768.

Groups contain parts and programs. Each part is a particular piece of your interface, such as a text label, or a rectangle, or an image. Each part has an individual name for reference by other parts of your EDC, again choose this name as descriptively as you can. Following the name we can define our part as one of the following types, IMAGE, RECT or TEXT. Looking at our EDC example you can see that our first part is an image named "background". The keyword "mouse_events" is a boolian value defining whether or not this part accepts mouse events (such as clicks, mouse overs, etc), 0 for no and 1 for yes. More options than just these 3 can be used, please refer to the reference for other avalible options.

A part will contain one or more description sections. Each description is a diffrent state of our part. The descriptions define how our part should look, what images or text the part uses, where it should be positioned, how it should be tiled, etc. In this case our background only needs one state, which we call default, using the state keyword. The number following the state

name defines an index value which is currently unused, simply use 0.0. Next we see two sections define the positioning of our part, rel1 and rel2. Each rel section will contain the keywords relative and offset, and optionally to. If the to keyword is omited then the position is relative to the full size of the interface as described by min and max in the group section. rel1 is the positioning of the top left corner of the part and rel2 is the positioning of the bottom right corner of the part. The relative keyword is followed by two doubles ranging from 0.0 to 1.0. As with min and max, the first value is the horizontal and the second is the vertical. The offset specified the pixel deviation from the relative point. In the case of our background, the top left (rel1) of our part (image) is relative to the top left corner of the interface, with no offset. The bottom right (rel2) of our part is relative to the bottom right of the interface, offset by 1 pixel left and 1 pixel up. Therefore, this part fills the entire interface. More information about positioning can before later in this guide. The last section in this parts description is the image section. This section describes which images to use. One or more images can be specified, the image that is first seen is denoted by the keyword normal. More images can be added using the keyword "tween" to form animations, but we will discuse that more later.

While parts may seem confusing and complicated at first, hopefully you now can look at the part we just reviewed and simply say that it's an image part named background that doesn't accept mouse events, fills the interface completely using the image background.png.

Our next part is a rectangle named button and it does accept mouse events. This part has two descriptions, one for the default state and one for the clicked state. The default state will define the normal look and positioning of the rectangle we will use as a button, or in otherwords, the unclicked state of our button. The second state is the clicked state. You'll notice that the default and clicked states look almost identical, the only change is the color and name. This means that when we change from state default to state clicked, the only thing that changes is the color of the rectangle. States are changed using programs, which we will discuse later. You'll notice that the descriptions contain min and max keywords, these are used to define the size of the part. If they are omited, like in the background part, the part will fill the maximum ammount of avalible space (ie: the whole interface, as limited by rel1 and rel2). The alignment keyword specifies alignment of our part within its avalible space (container). Values for align are again horizontal alignment followed by verital alignment, using doubles. So in this case the rectangle will be an absolute 100x50 pixels, with a container the size of the whole interface (as defined by rel1/rel2), and is positioned in the middle of that container. No matter how big or small the interface is the 100x50 pixel rectangle will always stay in exactly the middle of the screen and never resize. The color keyword is applicable only to rectangles and text, and describes the color of the rectangle in the form: color, red green blue alpha.

The last part is a text part, named text. It does not accept mouse events. It has only one state, default. Notice that the rel1 and rel2 sections use the to keyword, this modifies the meaning of the relativity. This means that the top left corner (rel1) of the part is relative to the top left corner (0.0 0.0) of the part named button (to, "button";). Likewise, the bottom right corner (rel2) of the part is relative to the bottom right corner (1.0 1.0) of the part named button, but moving the bottom right corner of the part by 1 pixel to the left and upwards from that point. The text section describes the text itself. The text keyword describes the text to display ("Press Me"), the font to use ("redensek"), the size of that font (14), and the alignment of the text within the container as defined by rel1/rel2. The font specified to be used must be added to your EVAS font path in your application, and the font name is the filename minus it's extention (ie: .ttf).

A word about layering. There is not specific keyword for layering in EDCs on a part-by-part basis. Each new part is layered on top of the previous. Therefore in our sample EDC the text is rendered ontop of the button, which sits atop the background. If we had defined the button before the background it would not have been visable. While this is common sense for the mostpart, it can be a common cause of confusion when modifying a large EDC if you aren't careful. Whenever you modify or add parts to your EDCs check whats above and below it.

The final section in our EDC is the programs section. Programs breath life into the static parts that make up our interface. Programs are largely based on the reception of signals. Signals can be generated by user interfaction, Edje itself, or an external force (usually your window manager). For instance, a user generated signal would be mouse in (when the user moves their pointer over a part), or mouse down (user depreses mouse button over a part). An Edje generated signal would include a "load" signal (Edje loads the EET), or "program,start" (when some other program starts running). An externally generated signal would effect the entire Edje interface, generally from a window manager, such as "move" (your interface window is moved) or "show" (your interface is displayed).

The first program in our example is to change the state of our button part when it is clicked on. The program name is "button_click". The program is run when it recieves the signal "mouse,down,1", meaning when the left mouse button is depressed over your part that accepts mouse events the program activates. Signals are globable, meaning if we wanted the program to run when any mouse button is depressed on our part we could use the signal "mouse,down,*", in fact we could use the signal "*" meaning that ANY signal effecting the part would run the program. The next keyword is source, which defines the part (or program) from which the signal will be recieved, in this case button. Sources are also globable. The next keyword is action, this

what the program actually does. Actions can change part states, stop the action of other programs, and emit signals. In this case the action STATE_SET still change state to "clicked". The following double (0.0) is currently unused and should simply be set to 0.0. The target argument that follows is the part or program on which the action acts apon. The final keyword, after, optionally defines another program to be run after the current program completes. When a signal is recieved all the programs which accept the incoming signal and match the source will be run, and in this way very often the keyword "after" is not required, however it can still be used for some crafty purposes which we'll explore later. After's can also be used as a looping mechignism, by specifying the current program to re-run after it completes, however it should be noted that any signal specified for the program must be met on every run of that program, even if it loops back to itself.

Looking at the first program again, we can now clearly see that the program "button_click" will be run when the left mouse button is depressed on our "button" rectangle. It will change the state of the target "button" to "clicked" (which will change the color as noted earlier), and once it completes will run the "do_me" program. Thus, looking at the next program "button_unclick" we can see that it will change the state of target "button" to the "default" state (back to its origonal color) when the left mouse button is released over the source part "button". Hence we have an animated button! Typically images would be used instead of simple rectangles, which would simply omit the color keyword and add an image section to both states, one image for the default state and one for the clicked state. Lastly we see that the after keyword is used to run the program "stop_doing_me" after "button_unclick" completes running. It should be noted that all program and part names are completely arbitrary, there is no restrictions on program or part state descriptions with the exception of the default state, which should always be named as such. Programs must always contain at least the keywords name, source, signal and action, even if source and signal are globbed to match anything ("*").

The third program as referenced by the "button_click" program. This program will accept any signal (as denoted by a * for the signal). The source is defined as a program in this case, rather than a part, so the program will run when any signal is accepted from the "button_click" program. The action defined is SIGNAL_EMIT, which will send the specified signal, here "PANTS ON", which is typically used by your application code. The third argument of of action for SIGNAL_EMIT is the source from which the signal came. In your application code this signal would be recieved by a callback handler, which would call a specified function based on the recipt of a specified signal from a specific source. We'll learn more about these signals when we discuse the Edje API later. The keyword "in" accepts two arguments, both doubles. "in" specifies a delay on running your program, the first argument is the number of seconds to wait before running the program, and

the second argument specifies a the maximum random delay which is added to the first argument. This is useful when you want the program to wait for a random amount of time that is at least a half a second but no more than 3 seconds, which would be described as "in, 0.5 3.0;". Delays always occur before the action specified by the program is preformed. Our final keyword is after, which will run the program "do_me" after the current program completes, which in this case is a loop. Note that there is no target specified in this program, because the action isn't performed on any other program or part. Because this is a loop we can say that this program will be run after the "button_click" program completes, and will emit the signal "PANTS ON" from the source "button" every one second.

The final program is named "stop_doing_me", which is run after the program "button_unclick" completes and accepts any signal. The action "ACTION_STOP" is used to break a loop or other running program, as specified by the target, in this case "do_me".

You should now take the sample EDC above, and use Edje_CC to build an eet. You can get the background image here: BACKGROUNDIMG. Using the edje_cc build your EET like this: "edje_cc -v -id . sample.edc sample.eet", putting the image in the same directory with the EDC. You should put the font "redensek.ttf" in a directory named "fonts/" where your EET will be used with edje (the viewer) from. You can then preview that EET with Edje, specifying the EET filename and the group name: "edje sample.eet test". Play with the EDC alittle untill you think you are familar with the syntax, layout and basic functionality of Edje EDCs.

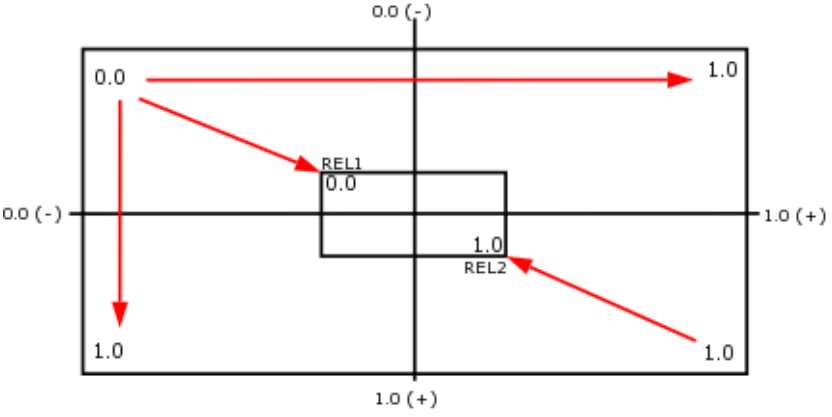
Chapter 3. Parts: Interface componants

Parts come in four varieties: RECT, IMAGE, TEXT and NONE.

Chapter 4. Edje Layout, Fills and Positioning

The Edje positioning model is based heavily on relativity, and is difficult for many new users to adjust to. Simply put, every part defined is relative to something else and is positioned relative to it. If you do not explicitly define what a part is relative to it is relative to the entire interface. The sections rel1 and rel2 are present in every part description and define the positioning. Each part should be thought of as a container, and the contents of the part whether may not neccisarily fill the entire container. Thus we can think of the interface itself as a container, which simply contains other containers. rel1 defines the positioning of the top left corner of the part container, and rel2 defines the positioning of the bottom right corner of the part container. Doubles are used with the keyword relative ranging from 0.0 to 1.0 representing a percentage of left-to-right and top-to-bottom. For instance, on a container the top left corner is 0.0 0.0, where the first double is the horizontal position and the second is the vertical. Whenever positioning is done in Edje you will always specify both horizontal and veritical parameters, and always horizontal first followed by veritcal. Thus the bottom right corner is 1.0 1.0. Everything starts from the top left corner. So the position 0.5 0.5 (relative to the whole interface) would be the middle of the interface, half way across the interface to the right and half way down the interface moving south.

Figure 4.1. Positioning Model



In the figure we can see two containers, the large one representing the whole interface and the smaller in the middle representing a part. On top I have provided an Edje compass which is a helpful tool to visualize positioning. Looking at the compass, notice that the left and northern directions approach 0.0 moving negatively, and that the right and southern directions approach 1.0 moving in the positive direction. Again, this is because everything starts from the top left corner of the parent container, regardless if that parent container is the entire interface or just another part. So the part in the middle of our figure would be positioned starting at roughtly 0.4 0.4, not 0.5 0.5. 0.5 0.5 would place the top left corner of the part exactly in the middle of the interface, not center entire part. To better illistrate this lets look at some examples as you would see them in an EDC.

Figure 4.2. Placing a rectangle

```
part {
        name, "background";
        type, RECT;
        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, "black_rect";
        type, RECT;
        description {
                state, "default" 0.0;
                color, 0 0 0 255;

                rel1 {
                        relative, 0.0 0.0;
                        offset, 10 10;
```

```
                        }

                        rel2 {
                                relative, 1.0 1.0;
                                offset, -10 -10;
                        }
                }

        }
```

In the example above we define two parts, both simple rectangles. The rel1 and rel2 sections are defining the position of these rectangles. The first part named "background" positions the rectangles top left corner (rel1) to 0.0 (left) 0.0 (top) of the interface since no "to" keyword is present. The bottom right corner (rel2) is positioned at 1.0 (far right) 1.0 (far bottom). No offset is specified for either position so the rectangle will stretch from the top left corner of the interface to the bottom right corner of the interface, which is the entire interface. An EVAS has no default background, and by extension, neither does an Edje interface, therefore you will always need to do something similiar to this whether you use an image or a rectangle to fill in the background.

The second part in the example above is very similar to the "background" part, it is not relative to anything, and therefore is relative to the whole interface. However, notice that an offset is specified for both rel1 and rel2. These values are possition modifiers specified in pixels. These offsets are based on the positioning defined by the relative keyword. Again, the two arguments specify horizontal positioning followed by vertical. So by using an offset in rel1 of 10 10, we're moving the top left corner of the part (rel1) by 10 pixels to the right and 10 pixels south of the position specified by relative. In the same way, rel2 is placed at the bottom right corner of the interface, and offset by 10 pixels to the left (-10) and 10 pixels north (-10). Remember that these values are negative because all positioning origionates from the top left corner of the container, so in this case we move backward toward it making the values negative.

So putting these two parts together we are left with an effect of a black interface with a 10 pixel white border. Or if you look at it another way, a white interface with a giant rectangle that is inset by 10 pixels. In Edje there are always several ways to look at placement to simulate effects. One variation that could have been used in the example would have been to place the "black_rect" part relative to the "background" part, rather than positioning both relative to the entire interface. In this example we could have added to the definition "to, "background";" to both rel1 and rel2 of "black_rect"s de-

fault state and achieved the same result with the added bonus that if we for some reason changed the positioning of the background the "black_rect" would be positioned accordingly.

To create a line of buttons in Edje we are presented with some positioning decisions to make. Each button could be positioned absolutely to the corners of the interface, or instead the first button would be positioned absolutely and each following button positioned relative to the first. Which method is right for your interface is important descision as it will effect the ammount of effort needed to modify the interface later if you choose or if you move those parts using a program. If you position each button relative to the first you only need to move the first buttons position to move them all as opposed to moving each button individually. In this way we can group parts together to make them act in a unified positioning manner.

Figure 4.3. Placing a row of buttons

```
part {
        name, "button1";
        type, RECT;
        description {
                state, "default" 0.0;
                color, 0 0 0 255;

                rel1 {
                        relative, 0.0 0.0;
                        offset, 10 10;
                }

                rel2 {
                        relative, 0.0 0.0;
                        offset, 30 20;
                }
        }

}

part {
        name, "button2";
        type, RECT;
        description {
                state, "default" 0.0;
                color, 0 0 0 255;

                rel1 {
```

```
                                relative, 1.0 0.0;
                                offset, 10 0;
                                to, "button1";
                        }

                        rel2 {
                                relative, 1.0 1.0;
                                offset, 30 0;
                                to, "button1";
                        }
                }

        }
```

In this example we have two buttons. Looking at "button1" we see it's top left corner (rel1) is positioned at the top left corner of the interface (relative) and then offset 10 pixels to the right and 10 pixels south (offset). It's bottom right corner (rel2) is relative to the top left corner of the interface as well, but then offset 30 pixels to the right and 20 pixels south. We position both corners from the top left corner instead of from both the top left and bottom right corner of the interface because since we are using static offsets we want to ensure the button stays in the same place even if the interface is resized. Notice we are not specifying the min and max size of either buttons, all sizing is happening based on the position of the corners of that part. The result of the positioning scheme used for "button1" is a container that stretches from the top left corner and is 20 pixels wide and 10 pixels high. If we had specified a min and max size for the part of 10 pixels the rectangle would only fill half of the container, and unless an "align" keyword was used that rectangle would start from the top left corner of the container like everything else, thus the left side of the container would be filled with rect and the other half of the container empty.

Button2 in our example uses the "to" keyword in both rel1 and rel2. The top left corner of "button2" is positioned at the right top corner of "butt on1", because rel1 is relative to "button1" and the relative keyword specifies that the top left corner of this part is relative to 1.0 0.0 of it. An offset is also used, which moves the "button2" container to the right by 10 pixels horizontally, which is going to act as the button spacing. The bottom right corner (rel2) is again positioned relative to "button1" but this time relative to "button1"s bottom right corner. An offset of 30 pixels is used which provides 10 pixels for our button spacing, and then another 20 pixels for the containers width. In the end this gives us two buttons space 10 pixels apar t that are 20 by 10 pixels in size. If we wanted to add another button we'd make it relative to "button2" so that each button is relative to the next which is positioned ultimately by the

first button.

Suppose we wanted to create a button bar, but rather than using static buttons like in the last example we simply wanted them to fill the width of t he interface completely reguardless of how large or small it was resized. In this case we could use something like the following example.

Figure 4.4. Resizable row of buttons

```
part {
        name, "button1";
        type, RECT;
        description {
                state, "default" 0.0;
                color, 255 0 0 255;

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

                rel2 {
                        relative, 0.5 0.0;
                        offset, 0 30;
                }
        }

 }

 part {
        name, "button2";
        type, RECT;
        description {
                state, "default" 0.0;
                color, 0 255 0 255;

                rel1 {
                        relative, 0.5 0.0;
                        offset, 0 0;
                }

                rel2 {
                        relative, 1.0 0.0;
                        offset, 0 30;
                }
        }
```

```
}
```

In this example we are creating two buttons and positioning them relative to the interface. The first button defines its container from the top left corner of the interface (rel1) over to the middle of the interface, and 30 pixels south of the middle top (rel2). The second buttons container starts half way across the top of the interface (rel1) and extends the rest of the way across the interface and south by 30 pixels (rel2). So we have two buttons that fill the entire width of the interface and is 30 pixels in height. No matter how thin or wide we resize the interface the button bar will look exactly as it should.

If we wanted to modify the example to place the button bar on the bottom of the interface we would change the veritcal parameter of the relative keywords from 0.0 to 1.0 placing everything relative to the bottom of the interface, leaving the horizontal parameters alone and then changing our offsets from a posative 30 pixels to a negative 30 pixels in order to push back upward rather than down.

Chapter 5. Programs: Bring your interface to life

Programs define how your interface reacts to events. These events can come from Edje, user interaction, or an external force like your window manager. All these events come to Edje as signals. Signals from user interaction would include mouse clicks, key presses and mouse movement. Signals from an external source would include window moves, window resizes, window raises and lowers. And signals from Edje internally would include programs starting or stopping, and the loading an EET. Each signal must come from some place, whether it is a part or another program or in the case of a window manager move anywhere. The place from which a signal come is known as the source. If a user left clicks on an image part named "button", a "mouse,clicked,1" signal is generated from the source part "button".

Chapter 6. The Edje API: Putting your interface to work

The API for Edje is simple and easy to use. It should be noted from the outset that the API is designed to be restrictive. If you can not accomplish something in code that you want to do it will almost always mean that you should have done it in the EDC. This design ensures that even a lazy programmer doesn't deminish the power of an independant interface as designed by Edje.

The Edje API includes an initialization function, time related functions, external object functions, and object functions for manipulating and interfacing with your EET as created using EDC. Most commonly used will be the object functions, which provide the ability to add Edje interfaces as an EVAS object and modify animation cycles, resize, get or set values from text parts, define color classes, define and delete callbacks based on signals from your EET, and even emit signals that get sent back to your EET to trigger events. Lets look at a simple example of using Edje in your application, using Ecore and EVAS:

```
/* gcc `edje-config --cflags --libs` `ecore-config --cflags --li
/* A sample EDC can be had here: http://www.cuddletech.com/edje/
#include <Ecore_Evas.h>
#include <Ecore.h>
#include <Edje.h>

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

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

        double   edje_w, edje_h;


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

        ecore_init();
        ecore_event_handler_add(ECORE_EVENT_SIGNAL_EXIT, app_sig

        ecore_evas_init();
   ee = ecore_evas_software_x11_new(NULL, 0,  0, 0, 0, 0);
```

```
        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, "/usr/local/share/edje/data/

        edje_init();
   edje = edje_object_add(evas);
        edje_object_file_set(edje, "crossfade.eet", "test");
        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);

        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;
}
```

In this example we see Edje being initialized with edje_init(), then the edje being added to the evas similiar to any other object being added to an evas. The edje_object_file_set() function defines the EET to be used and the group name we wish to use as defined in our EDC. Using evas_object_move() we ensure that the Edje is placed in the top left corner of the evas. Next, using edje_object_size_min_get() we return the values of the minimum width and height of the group so we can use evas_object_resize() to resize the evas to snuggly fit our edje interface. And finally using evas_object_show() is the usual way we render the edje to the evas. Following this we also resize the entire evas and re-show it to make things tidy. Notice that the evas was created having no width or height only because we later resized it using values from the EET group.

Signals and Callbacks

Signals provide the communication between your interface and your application code. When an Edje program emits a signal your application needs to catch that signal and then do something with it, this functionality is provided by Edje signal callbacks. The Edje API provides you with two calls, one to add and one to delete signal callbacks: edje_object_signal_callback_add() and edje_object_signal_callback_del(). Both functions take 4 arguments: the Evas_Object (your edje), the signal thats emited, the source of that emission, and the function that should be called when the signal is recieved. These functions will not allow you to pass NULL arguments, and therefore if you want to create callbacks for several sources that emit the same signal you will need to add a seperate callback for each of the sources. The add function allows one more argument, a void pointer to any data you want passed to the callback function.

```
        edje_object_signal_callback_add(st_session->edje, "PLAY_

        ...

void prev_file(player_session *data, Evas *e, Evas_Object *obj,

        printf("DEBUG: Previous File Called\n");        /* Repor

        ecore_idler_del(data->play_idler);      /* Stop the curr

        data->play_list = evas_list_prev(data->play_list);

        if(file_is_ogg(evas_list_data(data->play_list))){
                setup_ao();
                get_vorbis(evas_list_data(data->play_list), data
                ao_open();
                data->play_idler = ecore_idler_add(play_loop, da
        } else {
                printf("File %s is not an OggVorbis file\n", eva
        }
}
```

This is a pretty shitty example.....

Chapter 7. Edje Programmers Guidelines

The purpose of Edje is to abstract as completely as possible the design of the interface from the application code. Always keep this in mind when using Edje. While at times Edje's API seems inadiquate or frustrating in its lack of customization, it is this way for a reason: to keep you as a coder from taking power away from the themer. If at any point the API seems like a restriction realize that what you want to do is possible, but should be done in the EDC not in the application code.

The following recommendations are just that, recommendations. However observing them will make your application easier to work on and contribute to for all involved.

Edje Spec Files

Once you have created you application and a default EDC you will want to make it as easy as possible for other developers and themers to create new interfaces for you application. While developers can read your code and figure out what does what, many of the themers can not, therefore a standardized method of describing what the application requires and provides should be utilized. This is done by creating a flat text file named README.edje. Below is an example of the format that should be used in your README.edje:

Figure 7.1. Example of README.edje

```
-----------------------------------------------
EDJE Specfication
-----------------------------------------------

Application: ePlayer
Author: Ben Rockwood [ benr@cuddletech.com ]
Last Update: 9/29/03
Desciption: An OggVorbis audio player


Signals accepted by application:
------------------------------
```

```
Signal: PLAY_PREVIOUS
Source: previous_button
Desc: Signal starts playing the previous track in the playlist
                and updates track information text parts.

Signal: PLAY_NEXT
Source: next_button
Desc: Signal starts playing the next track in the playlist
                and updates track information text parts.

Signal: PLAY
Source: play_button
Desc: Signal to start playing the current track from current pos
                negates pause signal

Signal: PAUSE
Source: pause_button
Desc: Signal to pause the current playing track

Signal: SEEK_FORWARD
Source: forward_button
Desc: Signal to seek forward in track by 5 seconds

Signal: SEEK_BACK
Source: back_button
Desc: Signal to seek backwards in track by 5 seconds

Signals sent by application:
----------------------------

Signal: BLINK_ALL
Source: button
Desc: Signal signifying an error state

[ See "Signals accepted" above, application also sends these sig


Text parts set by application:
------------------------------

Part: title
Desc: Window title in border (ie: eVorbisPlayer v0.0)

Part: artist_name
Desc: Artist name from Ogg comments field

Part: album_name
Desc: Album name from Ogg comments field
```

```
Part: song_name
Desc: Song name from Ogg comments field

Part: time_text
Desc: Current position in track, displayed as negative
                        value to end of track (ie: -01:02)

Part: vol_display_text
Desc: Current PCM volume level, 2 digit int.  Values
                        range from 00 to 99

Swallowed areas used by application:
------------------------------------

        None.

Data fields used by application:
--------------------------------

        None.

------------------------------------------------
Features and Notes
------------------------------------------------

- Play list support currently only allows for a command line
        supplied list.
- At a bare minimum it is requested that all alternate
        pants provide Artist and Track name, Play, Next and
        Previous track.  Everything else is up to you.
- A K-Jofol look-alike interface would be kool!

------------------------------------------------
Changelog:
------------------------------------------------
9/20/03: App now will set text part "vol_display_text" to a 2 di
                representing PCM mixer volume level.
9/7/03: App now accepts signal PAUSE
9/5/03: Added new functions for the signals SEEK_FORWARD and SEE
9/1/03: Initial creation.
```

This scheme allows the themer to have an understanding of what functionality is avalible and what parts will be accessed by the application code and therefore must exist, even if hidden. A changelog should be present to list the changes in the application that could effect themes past and present. A "Fea-

tures and Notes" section is optional but allows a place to outline various functionality of the application as well as notes from the coder to the themer as to now the app is intended to be used, their wishes, etc. This file should be present in the top level directory of your codebase.

Source Layout

Themes are typically stored in the data/ directory in the toplevel directory of your codebase. The structure of that directory is entirely of your own choosing. While it is acceptable to include only a pre-compiled EET with your code, it is highly recommended that you also include the EDC and image sources as well. This allows users with porting issues to rebuild the EETs themselves rather than submit needless bug reports, and for other themers to learn from your EDC or improve upon it. This allows themers to be more effectively leveraged to provide an even better app.

Theme Distribution

An important consideration for themeing was how to deal with distribution of themes. Due to the ugly nature of using renamed tarballs for data distribution Edb and then EET came to live. EET provides a simplistic manner of distributing complete themes and interfaces in a convinient single file package. Because EET will compress your images there is no need to re-compress an EET, though it is the distributers discretion to do so. When hosting themes pre-compiled EETs should be provided. The images and source EDC does not, but it is recommended that you make it avalible on request or in the same place as the EET using a simple and clear file convension such as: theme.eet and theme-src.tgz. Providing the source gives back to the community not only a nice theme but also a valuable learning tool for others. Themes should *not* be distributed in tarballs as it defeats the purpose of single file, one step distribution.

Chapter 8. The Edje Preview Program

The **edje** program serves provides an easy to use tool for previewing, testing and debugging EDCs. The tool accepts two arguments, the first is the name of the EET to use, the second is an optional argument specifying the Edje group to use specified within the EDC. If no group name is supplied you will be presented with a list of all the groups within the specified EET, you may click on one of the groups to preview it.

Within the **edje** source there are several printf() statements, which by default are commented out. Uncommenting these statements will cause the tool to print to STDOUT all events, signals and program starts and stops which is useful in debugging. If you choose to use **edje** for debugging purposes it is highly recommended you enable this output. You can find the source for the **edje** program in `edje/src/bin/edje_main.c`.

It should be noted that the **edje** tool does not automatically resize to fit your interface as defined by min and max size for the group. The default window size is 240x320. In order to see your interface at the proper default size you should resize the preview window (the window inside your window managers window) larger and then resize it down till it stops on the minimum size of the interface. [NOTE: Hopefully this will be fixed in time.]

Because certain elements of Edje need to be specified in code, **edje** provides you with some code side elements you can draw on for design and debugging purposes. It defines the EVAS font path to two locations: `(data_dir)/data/test/fonts` (typical `/usr/local/share/`) and `./fonts/`. Therefore when using **edje** for debugging you should include all your custom fonts in a `fonts/` directory relative to where you preview it from. Several key events are accepted for testing purposes. edje_object_play_set(edje, 1) is executed when the RETURN key is pressed, and edje_object_play_set(edje, 0) is executed when ESC is pressed. edje_object_animation_set(edje, 1) is executed when key A is pressed, and edje_object_animation_set(edje, 0) when key S is pressed. Using the directional keys UP, LEFT and RIGHT will change the text string of a text part named "text" to "String 1" for LEFT, "Buttox" for UP, and "You presed (KEY).." for RIGHT. Further, four color classes are defined, each with the color class name "bg", using the F1, F2, F3 and F4 keys.

Table 8.1. Edje Tool Event Facilities

Key Press	Edje Function Called
RETURN	edje_object_play_set(edje, 1);
ESC	edje_object_play_set(edje, 0);
a	edje_object_animation_set(edje, 1);
s	edje_object_animation_set(edje, 0);
LEFT_ARROW	edje_object_part_text_set(edje, "text", "String 1");
UP_ARROW	edje_object_part_text_set(edje, "text", "Buttox");
RIGHT_ARROW	edje_object_part_text_set(edje, "text", "You pressed \"U\". Nice one stenchie!");
F1	edje_object_color_class_set(edje, "bg", 255, 255, 255, 255, 0, 0, 0, 0, 0, 0, 0, 0);
F2	edje_object_color_class_set(edje, "bg", 255, 200, 120, 255, 0, 0, 0, 0, 0, 0, 0, 0);
F3	edje_object_color_class_set(edje, "bg", 120, 200, 255, 200, 0, 0, 0, 0, 0, 0, 0, 0);
F4	edje_object_color_class_set(edje, "bg", 255, 200, 50, 100, 0, 0, 0, 0, 0, 0, 0, 0);

Effective use of the **edje** tool can significantly ease debugging of your Edje interface.

Appendix A. EDC Reference

```
//EDC Form
images {
        // Images
}

collections {
   group {
        // Group1 Params

        parts {
                part {
                        // Part Params

                        description {
                                // State Params
                        }

                }
        }

        programs {
                program {
                        // Program Params
                }
        }
     }

   group {
        // Group2 Params
        parts {
        ........
        }
        programs {
        ......
        }
   }
}
```

Table A.1. EDC Keyword Reference

Section	Keyword	Parameters	Description
images	image,	"image" STOR-	Where storage

Section	Keyword	Parameters	Description
		AGE_METHOD;	method is of the following: COMP for lossless compressed, RAW for lossless uncompressed, or LOSSY for lossy compressed follorwed by the quality level (0-100) ex: image, "button.png" LOSSY 85; ex: image, "backdrop.jpg" RAW;
group	name,	"group_name";	Name used to access individual interface in an EET
group	min,	0 0;	Integer values specifying minimum horizontal (arg1) and vertical (arg2) size of interface in pixels.
group	max,	0 0;	Integer values specifying maximum horizontal (arg1) and vertical (arg2) size of interface in pixels.
part	name,	"part_name";	Symbolic part name, used for later reference in EDC
part	type,	TYPE;	Where type is: IMAGE, RECT, TEXT or NONE. If no type is specified IMAGE is the default type.
part	effect,	FX_TYPE;	Effect type applied to text as

Section	Keyword	Parameters	Description
			rendered by the part. Acceptable FX_TYPEs are: NONE, PLAIN, OUTLINE, SOFT_OUTLINE, SHADOW, SOFT_SHADOW, OUTLINE_SHADOW, OUTLINE_SOFT_SHADOW. Default type is NONE. ex: effect, SOFT_OUTLINE;
part	mouse_events,	0;	Boolian value specifying whether the part accepts mouse events or not. No signals are generated from parts that do not accept events.
part	repeat_events,	0;	Boolian value specifying whether a part repeats an event to the part below it. When repeat is set to 0 (off, the default) and two parts that acc ept events are on top of each other the top most object will recieve the event and not any parts below it, turning repeat to 1 (on) will continue to send the event down to the next

Section	Keyword	Parameters	Description
			part below it.
part	clip_to,	"part";	Clip to the size of the specified part. Any amount of the current part that exends beyond the size of the clipped to part will be clipped off. C lipped text parts always truncate the text string to "...".
part	color_class,	"class";	Name of color class to apply to the current part. Color classes are defined in application code.
part	text_events,	"class";	Name of text class to apply to the current part. Text classes are defined in application code.
description	state,	"name" INDEX;	Descriptive name for the individual state, the default state must always be named "default". The INDEX value is a double between 0.0 and 1.0 whic h indicates levels of completion, that defaults to 0.0. Multiple states can have the same name yet with a different index value. ex: state,

Section	Keyword	Parameters	Description
			"default" 0.0;
description	visible,	0;	Boolian value specifying whether part is visable or not. Non-visable parts do not generate events.
description	align,	HOR_VAL VER_VAL;	Specify alignment of the part within it's container as specified by rel1/rel2. Values are specified as doubles from 0.0 (align left/top) to 1.0 (align right/bottom). ex: align, 0.5 0.5; (Aligns part in center of container) ex: align, 0.0 1.0; (Aligns part to bottom left of container)
description	min,	HOR_SIZE VER_SIZE;	Integer values specifying minimum horizontal (arg1) and vertical (arg2) size of part in pixels. ex: min, 100 100;
description	max,	HOR_SIZE VER_SIZE;	Integer values specifying maximum horizontal (arg1) and vertical (arg2) size of part in pixels. ex: max, 100 100;
description	step,	HOR_VAL VERT_VAL;	Integer stepping values in integer pixels for hori-

Section	Keyword	Parameters	Description
			zontal (arg1) and vertical (arg2) scaling. When stepping is enabled the width or/and height of the image will always be divisable by it's stepping value when scaled. Default stepping values are 0 0 (ie: stepping disabled). ex: step, 20 1 (Image width must always be multple of 20, ie: 0, 20, 40, 60, etc. Height can be any value)
description	aspect,	MIN MAX;	Double min (arg1) and max (arg2) aspect ratio values. This controls the aspect ratio (ratio of width to height) of a scaled part, typically image s. The default ratio is 0.0. If both values are the same the ratio is fixed. ex: aspect, 1.0 5.0; (Minimum aspect of 1:1, maximum of 5:1 - Width:Height);
description	border,	LEFT RIGHT TOP BOTTOM;	Border scaling values for an image part as specified in integer

Section	Keyword	Parameters	Description
			pixel widths, for each four sides of an image. This will stop Edje from scaling t he outside edge of an image when scaling an image part. ex: border, 10 10 10 10; (Scale the edge of the image part 10 pixels on all sides)
description	color,	RED GREEN BLUE ALPHA;	Integer values ranging from 0 to 255 specifying the color of a rectangle or text part. ex: color, 0 0 0 255; (Part is colored black)
description	color2,	RED GREEN BLUE ALPHA;	Integer values ranging from 0 to 255 specifying the color of a text parts shadow. ex: color2, 0 0 255 255; (Shadow is blue)
description	color3,	RED GREEN BLUE ALPHA;	Integer values ranging from 0 to 255 specifying the color of a text parts outline. ex: color3, 255 0 0 255; (Outline is colored red)
rel1/rel2	relative,	HOR_VAL VERT_VAL;	Doubles representing the horizontal (arg1) and vertical (arg2) positioning of top

Section	Keyword	Parameters	Description
			left corner (for rel1) or bottom right corner (for rel2) as rel ative to the part specified by the "to" keyword. If no "to" keyword is present, the values are relative to the corners of the interface. ex: relative, 0.0 1.0; (For rel1 with no "to": top left corner of part is positioned at the left (0.0), bottom (1.0) corner of the interface.)
rel1/rel2	offset,	HORZ_OFF VERT_OFF;	Integers specifying deviation in pixels from the position as defined by the relative keyword, both horizontally (arg1) and vertically (arg2) ex: offset, 5 10; (Position 5 px to the right and 10 px down from the position as stated by the relative keyword)
rel1/rel2	to,	"part_name";	Specify another part as the reference to be used for the positioning of the current part. ex: to, "some_part";

Section	Keyword	Parameters	Description
rel1/rel2	to_x,	"part_name";	Specify another part as the reference to be used for the positioning of the current part. Same as "to", but relativity applies only on the X axis .
rel1/rel2	to_y,	"part_name";	Specify another part as the reference to be used for the positioning of the current part. Same as "to", but relativity applies only on the Y axi s.
image	normal,	"image_name";	Name of image to be used. In an animation, this is the first and last image displayed.
image	tween,	"image_name";	Name of an image to be used in an animation loop. Images are dispay in the order they are listed. There is no limit to the number of tweens tha t can be specified.
fill	smooth,	0;	Boolian value determinaning whether scaled images will be smoothed, 0 for no, 1 for yes.
fill { origin	relative,	HOR_VAL VERT_VAL;	Doubles representing the horizontal (arg1) and

Section	Keyword	Parameters	Description
			vertical (arg2) position from which a fill (tile) should start within it's container as defined by rel1/rel2. Tiling then occurs in all directions from that point of origin. This is similar in use to relativity by rel1 except that it is relative to the parts con tainer rather than the whole interface. ex: relative, 0.5 0.5; (part starts tiling from the middle of it's container)
fill { origin	offset	HOR_VAL VERT_VAL;	Integers specifying a pixel offset horizontally (arg1) and vertically (arg2) from the relative position specified by origin{relative,}. This is similar in use to offset used in rel1.
fill { size	relative,	HOR_VAL VERT_VAL;	Doubles representing the horizontal (arg1) and vertical (arg2) position of the bottom right corner of a fill (tile). This is similar in use to r elativity

Section	Keyword	Parameters	Description
			by rel2 except that it is relative to the parts container rather than the whole interface. ex: relative, 1.0 1.0; (Tile fills entire space)
fill { size	offset,	HOR_VAL VERT_VAL;	Integers specifying a pixel offset horizontally (arg1) and vertically (arg2) from the relative position specified by size{relative,}. This is si milar in use to offset used in rel2.
text	text,	"some string";	Text string to be rendered.
text	font,	"font_name";	Font used for text, where "font_name" is the name of the font file minus its extension. Path to font is determined by your applications evas font path. ex: font, "Impact"; (Font used is Impact.ttf found in the evas font path)
text	size	12;	Font size in points.
text	fit,	HOR_VAL VERT_VAL;	Boolian values specifying whether to scale text to fill its container horizontally

Section	Keyword	Parameters	Description
			(arg1) and/or vertically (arg2). Default is 0 0;
text	min,	HOR_VAL VERT_VAL;	Boolian values specifying whether the current text string should define the minimum size of the part, such that all future changes to the text st ring can be no smaller both horizontally (arg1) and vertically (arg2).
text	align,	0.5 0.5;	Alignment of text within its containers as defined by rel1/rel2, horizontally (arg1) and vertically (arg2).
program	name,	"prog_name";	Symbolic name of program as a unique identifier.
program	signal,	SIGNAL;	Specifies signal(s) that a should cause program to run. The signal recieved must match the specified source to run. Signals may be globbed, but only one signal keyword per program may be used. ex: signal, "mouse,clicked,*"; (clicking any mouse button that

Section	Keyword	Parameters	Description
			matches source starts program)
program	source,	"signal-source";	Source of accepted signal. Sources may be globbed, but only one source keyword per program may be used. ex: source, "button-*"; (Signals from any part or program named "button-*" are accepted)
program	action,	ACTION (param1) (param2);	Action to be performed by the program. Valid actions are: STATE_SET, ACTION_STOP and SIGNAL_EMIT. Only one action can be specified per program.
program	transition,	TYPE LENGTH;	Defines how transistions occur using STATE_SET action. Where type is the style of the transistion and length is a double specifying the number o f seconds in which to preform the transistion. Valid types are: LINEAR, SINUSOIDAL, ACCELERATE, and DECELERATE

Section	Keyword	Parameters	Description
program	target,	"action-target";	Program or part on which the specified action acts. Multiple target keywords may be specified, one per target. SIGNAL_EMITs do not have targets .
program	after,	"next-program";	Specifies a program run after the current program completes. The source and signal parameters of a program run as an "after" are ignored.