GStreamer Plugin Writer's Guide (0.10.23)

Richard John Boulton

Erik Walthinsen

Steve Baker

Leif Johnson

Ronald S. Bultje

Stefan Kost

Tim-Philipp MÃ¼ller

GStreamer Plugin Writer's Guide (0.10.23)

by Richard John Boulton, Erik Walthinsen, Steve Baker, Leif Johnson, Ronald S. Bultje, Stefan Kost, and Tim-Philipp MÃ¼ller

Table of Contents

Chapter 1. Preface

What is GStreamer?

GStreamer is a framework for creating streaming media applications. The fundamental design comes from the video pipeline at Oregon Graduate Institute, as well as some ideas from DirectShow.

GStreamer's development framework makes it possible to write any type of streaming multimedia application. The GStreamer framework is designed to make it easy to write applications that handle audio or video or both. It isn't restricted to audio and video, and can process any kind of data flow. The pipeline design is made to have little overhead above what the applied filters induce. This makes GStreamer a good framework for designing even high-end audio applications which put high demands on latency.

One of the the most obvious uses of GStreamer is using it to build a media player. GStreamer already includes components for building a media player that can support a very wide variety of formats, including MP3, Ogg/Vorbis, MPEG-1/2, AVI, Quicktime, mod, and more. GStreamer, however, is much more than just another media player. Its main advantages are that the pluggable components can be mixed and matched into arbitrary pipelines so that it's possible to write a full-fledged video or audio editing application.

The framework is based on plugins that will provide the various codec and other functionality. The plugins can be linked and arranged in a pipeline. This pipeline defines the flow of the data. Pipelines can also be edited with a GUI editor and saved as XML so that pipeline libraries can be made with a minimum of effort.

The GStreamer core function is to provide a framework for plugins, data flow and media type handling/negotiation. It also provides an API to write applications using the various plugins.

Who Should Read This Guide?

This guide explains how to write new modules for GStreamer. The guide is relevant to several groups of people:

- Anyone who wants to add support for new ways of processing data in GStreamer. For example, a person in this group might want to create a new data format converter, a new visualization tool, or a new decoder or encoder.
- Anyone who wants to add support for new input and output devices. For example, people in this group might want to add the ability to write to a new video output system or read data from a digital camera or special microphone.
- Anyone who wants to extend GStreamer in any way. You need to have an understanding of how the plugin system works before you can understand the constraints that the plugin system places on the rest of the code. Also, you might be surprised after reading this at how much can be done with plugins.

This guide is not relevant to you if you only want to use the existing functionality of GStreamer, or if you just want to use an application that uses GStreamer. If you are only interested in using existing plugins to write a new application - and there are quite a lot of plugins already - you might want to check the *GStreamer Application Development Manual*. If you are just trying to get help with a GStreamer application, then you should check with the user manual for that particular application.

Preliminary Reading

This guide assumes that you are somewhat familiar with the basic workings of GStreamer. For a gentle introduction to programming concepts in GStreamer, you may wish to read the *GStreamer Application Development Manual* first. Also check out the other documentation available on the GStreamer web site[1].

In order to understand this manual, you will need to have a basic understanding of the C language. Since GStreamer adheres to the GObject programming model, this guide also assumes that you understand the basics of GObject[2] programming. You may also want to have a look at Eric Harlow's book *Developing Linux Applications with GTK+ and GDK*.

Structure of This Guide

To help you navigate through this guide, it is divided into several large parts. Each part addresses a particular broad topic concerning GStreamer plugin development. The parts of this guide are laid out in the following order:

- Building a Plugin - Introduction to the structure of a plugin, using an example audio filter for illustration.

  This part covers all the basic steps you generally need to perform to build a plugin, such as registering the element with GStreamer and setting up the basics so it can receive data from and send data to neighbour elements. The discussion begins by giving examples of generating the basic structures and registering an element in Constructing the Boilerplate. Then, you will learn how to write the code to get a basic filter plugin working in Chapter 4, Chapter 5 and Chapter 6.

  After that, we will show some of the GObject concepts on how to make an element configurable for applications and how to do application-element interaction in Adding Arguments and Chapter 8. Next, you will learn to build a quick test application to test all that you've just learned in Chapter 9. We will just touch upon basics here. For full-blown application development, you should look at the Application Development Manual[3].

- Advanced Filter Concepts - Information on advanced features of GStreamer plugin development.

  After learning about the basic steps, you should be able to create a functional audio or video filter plugin with some nice features. However, GStreamer offers more for plugin writers. This part of the guide includes chapters on more advanced topics, such as scheduling, media type definitions in GStreamer, clocks, interfaces and

tagging. Since these features are purpose-specific, you can read them in any order, most of them don't require knowledge from other sections.

The first chapter, named Different scheduling modes, will explain some of the basics of element scheduling. It is not very in-depth, but is mostly some sort of an introduction on why other things work as they do. Read this chapter if you're interested in `GStreamer` internals. Next, we will apply this knowledge and discuss another type of data transmission than what you learned in Chapter 5: Different scheduling modes. Loop-based elements will give you more control over input rate. This is useful when writing, for example, muxers or demuxers.

Next, we will discuss media identification in `GStreamer` in Chapter 12. You will learn how to define new media types and get to know a list of standard media types defined in `GStreamer`.

In the next chapter, you will learn the concept of request- and sometimes-pads, which are pads that are created dynamically, either because the application asked for it (request) or because the media stream requires it (sometimes). This will be in Chapter 13.

The next chapter, Chapter 14, will explain the concept of clocks in `GStreamer`. You need this information when you want to know how elements should achieve audio/video synchronization.

The next few chapters will discuss advanced ways of doing application-element interaction. Previously, we learned on the GObject-ways of doing this in Adding Arguments and Chapter 8. We will discuss dynamic parameters, which are a way of defining element behaviour over time in advance, in Chapter 15. Next, you will learn about interfaces in Chapter 17. Interfaces are very target-specific ways of application-element interaction, based on GObject's GInterface. Lastly, you will learn about how metadata is handled in `GStreamer` in Chapter 18.

The last chapter, Chapter 19, will discuss the concept of events in `GStreamer`. Events are, on the one hand, another way of doing application-element interaction. It takes care of seeking, for example. On the other hand, it is also a way in which elements interact with each other, such as letting each other know about media stream discontinuities, forwarding tags inside a pipeline and so on.

- Creating special element types - Explanation of writing other plugin types.

  Because the first two parts of the guide use an audio filter as an example, the concepts introduced apply to filter plugins. But many of the concepts apply equally to other plugin types, including sources, sinks, and autopluggers. This part of the guide presents the issues that arise when working on these more specialized plugin types. The chapter starts with a special focus on elements that can be written using a base-class (Pre-made base classes), and later also goes into writing special types of elements in Writing a Demuxer or Parser, Writing a N-to-1 Element or Muxer and Writing a Manager.

- Appendices - Further information for plugin developers.

  The appendices contain some information that stubbornly refuses to fit cleanly in other sections of the guide. Most of this section is not yet finished.

The remainder of this introductory part of the guide presents a short overview of the basic concepts involved in `GStreamer` plugin development. Topics covered include Elements and Plugins, Pads, Data, Buffers and Events and Types and Properties. If

you are already familiar with this information, you can use this short overview to refresh your memory, or you can skip to Building a Plugin.

As you can see, there a lot to learn, so let's get started!

- Creating compound and complex elements by extending from a GstBin. This will allow you to create plugins that have other plugins embedded in them.
- Adding new mime-types to the registry along with typedetect functions. This will allow your plugin to operate on a completely new media type.

Notes

1. http://gstreamer.freedesktop.org/documentation/
2. http://developer.gnome.org/doc/API/2.0/gobject/index.html
3. http://gstreamer.freedesktop.org/data/doc/gstreamer/head/manual/html/index.html

Chapter 2. Foundations

This chapter of the guide introduces the basic concepts of GStreamer. Understanding these concepts will help you grok the issues involved in extending GStreamer. Many of these concepts are explained in greater detail in the *GStreamer Application Development Manual*; the basic concepts presented here serve mainly to refresh your memory.

Elements and Plugins

Elements are at the core of GStreamer. In the context of plugin development, an *element* is an object derived from the GstElement[1] class. Elements provide some sort of functionality when linked with other elements: For example, a source element provides data to a stream, and a filter element acts on the data in a stream. Without elements, GStreamer is just a bunch of conceptual pipe fittings with nothing to link. A large number of elements ship with GStreamer, but extra elements can also be written.

Just writing a new element is not entirely enough, however: You will need to encapsulate your element in a *plugin* to enable GStreamer to use it. A plugin is essentially a loadable block of code, usually called a shared object file or a dynamically linked library. A single plugin may contain the implementation of several elements, or just a single one. For simplicity, this guide concentrates primarily on plugins containing one element.

A *filter* is an important type of element that processes a stream of data. Producers and consumers of data are called *source* and *sink* elements, respectively. *Bin* elements contain other elements. One type of bin is responsible for scheduling the elements that they contain so that data flows smoothly. Another type of bin, called *autoplugger* elements, automatically add other elements to the bin and links them together so that they act as a filter between two arbitary stream types.

The plugin mechanism is used everywhere in GStreamer, even if only the standard packages are being used. A few very basic functions reside in the core library, and all others are implemented in plugins. A plugin registry is used to store the details of the plugins in an XML file. This way, a program using GStreamer does not have to load all plugins to determine which are needed. Plugins are only loaded when their provided elements are requested.

See the *GStreamer Library Reference* for the current implementation details of GstElement[2] and GstPlugin[3].

Pads

Pads are used to negotiate links and data flow between elements in GStreamer. A pad can be viewed as a "place" or "port" on an element where links may be made with other elements, and through which data can flow to or from those elements. Pads have specific data handling capabilities: A pad can restrict the type of data that flows through it. Links are only allowed between two pads when the allowed data types of the two pads are compatible.

An analogy may be helpful here. A pad is similar to a plug or jack on a physical device. Consider, for example, a home theater system consisting of an amplifier, a DVD player, and a (silent) video projector. Linking the DVD player to the amplifier is allowed because both devices have audio jacks, and linking the projector to the DVD player is allowed because both devices have compatible video jacks. Links between the projector and the amplifier may not be made because the projector and amplifier have different types of jacks. Pads in `GStreamer` serve the same purpose as the jacks in the home theater system.

For the most part, all data in `GStreamer` flows one way through a link between elements. Data flows out of one element through one or more *source pads*, and elements accept incoming data through one or more *sink pads*. Source and sink elements have only source and sink pads, respectively.

See the *GStreamer Library Reference* for the current implementation details of a `GstPad`[4].

Data, Buffers and Events

All streams of data in `GStreamer` are chopped up into chunks that are passed from a source pad on one element to a sink pad on another element. *Data* are structures used to hold these chunks of data.

Data contains the following important types:

- An exact type indicating what type of data (control, content, ...) this Data is.
- A reference count indicating the number of elements currently holding a reference to the buffer. When the buffer reference count falls to zero, the buffer will be unlinked, and its memory will be freed in some sense (see below for more details).

There are two types of data defined: events (control) and buffers (content).

Buffers may contain any sort of data that the two linked pads know how to handle. Normally, a buffer contains a chunk of some sort of audio or video data that flows from one element to another.

Buffers also contain metadata describing the buffer's contents. Some of the important types of metadata are:

- A pointer to the buffer's data.
- An integer indicating the size of the buffer's data.
- A timestamp indicating the preferred display timestamp of the content in the buffer.

Events contain information on the state of the stream flowing between the two linked pads. Events will only be sent if the element explicitely supports them, else the core will (try to) handle the events automatically. Events are used to indicate, for example, a clock discontinuity, the end of a media stream or that the cache should be flushed.

Events may contain several of the following items:

- A subtype indicating the type of the contained event.
- The other contents of the event depend on the specific event type.

Events will be discussed extensively in Chapter 19. Until then, the only event that will be used is the *EOS* event, which is used to indicate the end-of-stream (usually end-of-file).

See the *GStreamer Library Reference* for the current implementation details of a `GstMiniObject`[5], `GstBuffer`[6] and `GstEvent`[7].

Buffer Allocation

Buffers are able to store chunks of memory of several different types. The most generic type of buffer contains memory allocated by malloc(). Such buffers, although convenient, are not always very fast, since data often needs to be specifically copied into the buffer.

Many specialized elements create buffers that point to special memory. For example, the filesrc element usually maps a file into the address space of the application (using mmap()), and creates buffers that point into that address range. These buffers created by filesrc act exactly like generic buffers, except that they are read-only. The buffer freeing code automatically determines the correct method of freeing the underlying memory. Downstream elements that recieve these kinds of buffers do not need to do anything special to handle or unreference it.

Another way an element might get specialized buffers is to request them from a downstream peer. These are called downstream-allocated buffers. Elements can ask a peer connected to a source pad to create an empty buffer of a given size. If a downstream element is able to create a special buffer of the correct size, it will do so. Otherwise `GStreamer` will automatically create a generic buffer instead. The element that requested the buffer can then copy data into the buffer, and push the buffer to the source pad it was allocated from.

Many sink elements have accelerated methods for copying data to hardware, or have direct access to hardware. It is common for these elements to be able to create downstream-allocated buffers for their upstream peers. One such example is ximagesink. It creates buffers that contain XImages. Thus, when an upstream peer copies data into the buffer, it is copying directly into the XImage, enabling ximagesink to draw the image directly to the screen instead of having to copy data into an XImage first.

Filter elements often have the opportunity to either work on a buffer in-place, or work while copying from a source buffer to a destination buffer. It is optimal to implement both algorithms, since the `GStreamer` framework can choose the fastest algorithm as appropriate. Naturally, this only makes sense for strict filters -- elements that have exactly the same format on source and sink pads.

Mimetypes and Properties

GStreamer uses a type system to ensure that the data passed between elements is in a recognized format. The type system is also important for ensuring that the parameters required to fully specify a format match up correctly when linking pads between elements. Each link that is made between elements has a specified type and optionally a set of properties.

The Basic Types

GStreamer already supports many basic media types. Following is a table of a few of the the basic types used for buffers in GStreamer. The table contains the name ("mime type") and a description of the type, the properties associated with the type, and the meaning of each property. A full list of supported types is included in List of Defined Types.

Table 2-1. Table of Example Types

Mime Type	Description	Property	Property Type	Property Values	Property Description
audio/*	*All audio types*	rate	integer	greater than 0	The sample rate of the data, in samples (per channel) per second.
		channels	integer	greater than 0	The number of channels of audio data.

Mime Type	Description	Property	Property Type	Property Values	Property Description
audio/x-raw-int	Unstructured and uncompressed raw integer audio data.	endianness	integer	G_BIG_ENDIAN (4321) or G_LITTLE_ENDIAN (1234)	The order of bytes in a sample. The value G_LITTLE_ENDIAN (1234) means "little-endian" (byte-order is "least significant byte first"). The value G_BIG_ENDIAN (4321) means "big-endian" (byte order is "most significant byte first").
		signed	boolean	TRUE or FALSE	Whether the values of the integer samples are signed or not. Signed samples use one bit to indicate sign (negative or positive) of the value. Unsigned samples are always positive.
		width	integer	greater than 0	Number of bits allocated per sample.

Mime Type	Description	Property	Property Type	Property Values	Property Description
		depth	integer	greater than 0	The number of bits used per sample. This must be less than or equal to the width: If the depth is less than the width, the low bits are assumed to be the ones used. For example, a width of 32 and a depth of 24 means that each sample is stored in a 32 bit word, but only the low 24 bits are actually used.
audio/mpeg	Audio data compressed using the MPEG audio encoding scheme.	mpegversion	integer	1, 2 or 4	The MPEG-version used for encoding the data. The value 1 refers to MPEG-1, -2 and -2.5 layer 1, 2 or 3. The values 2 and 4 refer to the MPEG-AAC audio encoding schemes.

Mime Type	Description	Property	Property Type	Property Values	Property Description
		framed	boolean	0 or 1	A true value indicates that each buffer contains exactly one frame. A false value indicates that frames and buffers do not necessarily match up.
		layer	integer	1, 2, or 3	The compression scheme layer used to compress the data *(only if mpegversion=1)*.
		bitrate	integer	greater than 0	The bitrate, in bits per second. For VBR (variable bitrate) MPEG data, this is the average bitrate.
audio/x-vorbis	Vorbis audio data				There are currently no specific properties defined for this type.

Notes

1. ../../gstreamer/html/GstElement.html

2. ../../gstreamer/html/GstElement.html
3. ../../gstreamer/html/GstPlugin.html
4. ../../gstreamer/html/GstPad.html
5. ../../gstreamer/html/gstreamer-GstMiniObject.html
6. ../../gstreamer/html/gstreamer-GstBuffer.html
7. ../../gstreamer/html/gstreamer-GstEvent.html

Chapter 3. Constructing the Boilerplate

In this chapter you will learn how to construct the bare minimum code for a new plugin. Starting from ground zero, you will see how to get the `GStreamer` template source. Then you will learn how to use a few basic tools to copy and modify a template plugin to create a new plugin. If you follow the examples here, then by the end of this chapter you will have a functional audio filter plugin that you can compile and use in `GStreamer` applications.

Getting the GStreamer Plugin Templates

There are currently two ways to develop a new plugin for `GStreamer`: You can write the entire plugin by hand, or you can copy an existing plugin template and write the plugin code you need. The second method is by far the simpler of the two, so the first method will not even be described here. (Errm, that is, "it is left as an exercise to the reader.")

The first step is to check out a copy of the `gst-template` git module to get an important tool and the source code template for a basic `GStreamer` plugin. To check out the `gst-template` module, make sure you are connected to the internet, and type the following commands at a command console:

```
shell $ git clone git://anongit.freedesktop.org/gstreamer/gst-template.git
Initialized empty Git repository in /some/path/gst-template/.git/
remote: Counting objects: 373, done.
remote: Compressing objects: 100% (114/114), done.
remote: Total 373 (delta 240), reused 373 (delta 240)
Receiving objects: 100% (373/373), 75.16 KiB | 78 KiB/s, done.
Resolving deltas: 100% (240/240), done.
```

This command will check out a series of files and directories into `gst-template`. The template you will be using is in the `gst-template/gst-plugin/` directory. You should look over the files in that directory to get a general idea of the structure of a source tree for a plugin.

Using the Project Stamp

The first thing to do when making a new element is to specify some basic details about it: what its name is, who wrote it, what version number it is, etc. We also need to define an object to represent the element and to store the data the element needs. These details are collectively known as the *boilerplate*.

The standard way of defining the boilerplate is simply to write some code, and fill in some structures. As mentioned in the previous section, the easiest way to do this is to copy a template and add functionality according to your needs. To help you do so, there is a tool in the `./gst-plugins/tools/` directory. This tool, `make_element`, is a command line utility that creates the boilerplate code for you.

To use **make_element**, first open up a terminal window. Change to the `gst-template/gst-plugin/src` directory, and then run the **make_element** command. The arguments to the **make_element** are:

1. the name of the plugin, and
2. the source file that the tool will use. By default, `gstplugin` is used.

Note that capitalization is important for the name of the plugin. Under some operating systems, capitalization is also important when specifying directory names. For example, the following commands create the ExampleFilter plugin based on the plugin template and put the output files in the `gst-template/gst-plugin/src` directory:

```
shell $ cd gst-template/gst-plugin/src
shell $ ../tools/make_element ExampleFilter
```

The last command creates two files: `gstexamplefilter.c` and `gstexamplefilter.h`.

> **Note:** It is recommended that you create a copy of the `gst-plugin` directory before continuing.

Examining the Basic Code

First we will examine the code you would be likely to place in a header file (although since the interface to the code is entirely defined by the plugin system, and doesn't depend on reading a header file, this is not crucial.) The code here can be found in `examples/pwg/examplefilter/boiler/gstexamplefilter.h`.

Example 3-1. Example Plugin Header File

```
#include <gst/gst.h>

/* Definition of structure storing data for this element. */
typedef struct _GstMyFilter {
  GstElement element;

  GstPad *sinkpad, *srcpad;

  gboolean silent;



} GstMyFilter;

/* Standard definition defining a class for this element. */
typedef struct _GstMyFilterClass {
  GstElementClass parent_class;
```

```
} GstMyFilterClass;

/* Standard macros for defining types for this element.  */
#define GST_TYPE_MY_FILTER (gst_my_filter_get_type())
#define GST_MY_FILTER(obj) \
  (G_TYPE_CHECK_INSTANCE_CAST((obj),GST_TYPE_MY_FILTER,GstMyFilter))
#define GST_MY_FILTER_CLASS(klass) \
  (G_TYPE_CHECK_CLASS_CAST((klass),GST_TYPE_MY_FILTER,GstMyFilterClass))
#define GST_IS_MY_FILTER(obj) \
  (G_TYPE_CHECK_INSTANCE_TYPE((obj),GST_TYPE_MY_FILTER))
#define GST_IS_MY_FILTER_CLASS(klass) \
  (G_TYPE_CHECK_CLASS_TYPE((klass),GST_TYPE_MY_FILTER))

/* Standard function returning type information. */
GType gst_my_filter_get_type (void);
```

Using this header file, you can use the following macro to setup the `GObject` basics in your source file so that all functions will be called appropriately:

```
#include "filter.h"

GST_BOILERPLATE (GstMyFilter, gst_my_filter, GstElement, GST_TYPE_ELEMENT);
```

GstElementDetails

The GstElementDetails structure gives a hierarchical type for the element, a human-readable description of the element, as well as author and version data. The entries are:

- A long, english, name for the element.
- The type of the element, see the docs/design/draft-klass.txt document in the GStreamer core source tree for details and examples.
- A brief description of the purpose of the element.
- The name of the author of the element, optionally followed by a contact email address in angle brackets.

For example:

```
static const GstElementDetails my_filter_details = {
  "An example plugin",
  "Example/FirstExample",
  "Shows the basic structure of a plugin",
  "your name <your.name@your.isp>"
};
```

The element details are registered with the plugin during the `_base_init ()` function, which is part of the GObject system. The `_base_init ()` function should be set for this GObject in the function where you register the type with GLib.

```
static void
gst_my_filter_base_init (gpointer klass)
{
  GstElementClass *element_class = GST_ELEMENT_CLASS (klass);

  static const GstElementDetails my_filter_details = {
[..]
  };

[..]
  gst_element_class_set_details (element_class, &my_filter_details);

}
```

GstStaticPadTemplate

A GstStaticPadTemplate is a description of a pad that the element will (or might) create and use. It contains:

- A short name for the pad.
- Pad direction.
- Existence property. This indicates whether the pad exists always (an “always” pad), only in some cases (a “sometimes” pad) or only if the application requested such a pad (a “request” pad).
- Supported types by this element (capabilities).

For example:

```
static GstStaticPadTemplate sink_factory =
GST_STATIC_PAD_TEMPLATE (
  "sink",
  GST_PAD_SINK,
  GST_PAD_ALWAYS,
  GST_STATIC_CAPS ("ANY")
);
```

Those pad templates are registered during the `_base_init ()` function. Pads are created from these templates in the element’s `_init ()` function using `gst_pad_new_from_template ()`. The template can be retrieved from the element class using `gst_element_class_get_pad_template ()`. See below for more details on this. In order to create a new pad from this template using `gst_pad_new_from_template ()`, you will need to declare the pad template as a global variable. More on this subject in Chapter 4.

```
static GstStaticPadTemplate sink_factory = [..],
    src_factory = [..];
```

```
static void
gst_my_filter_base_init (gpointer klass)
{
  GstElementClass *element_class = GST_ELEMENT_CLASS (klass);
[..]

  gst_element_class_add_pad_template (element_class,
 gst_static_pad_template_get (&src_factory));
  gst_element_class_add_pad_template (element_class,
 gst_static_pad_template_get (&sink_factory));
}
```

The last argument in a template is its type or list of supported types. In this example, we use 'ANY', which means that this element will accept all input. In real-life situations, you would set a mimetype and optionally a set of properties to make sure that only supported input will come in. This representation should be a string that starts with a mimetype, then a set of comma-separates properties with their supported values. In case of an audio filter that supports raw integer 16-bit audio, mono or stereo at any samplerate, the correct template would look like this:

```
static GstStaticPadTemplate sink_factory =
GST_STATIC_PAD_TEMPLATE (
  "sink",
  GST_PAD_SINK,
  GST_PAD_ALWAYS,
  GST_STATIC_CAPS (
    "audio/x-raw-int, "
      "width = (int) 16, "
      "depth = (int) 16, "
      "endianness = (int) BYTE_ORDER, "
      "channels = (int) { 1, 2 }, "
      "rate = (int) [ 8000, 96000 ]"
  )
);
```

Values surrounded by curly brackets ("{" and "}") are lists, values surrounded by square brackets ("[" and "]") are ranges. Multiple sets of types are supported too, and should be separated by a semicolon (";"). Later, in the chapter on pads, we will see how to use types to know the exact format of a stream: Chapter 4.

Constructor Functions

Each element has three functions which are used for construction of an element. These are the `_base_init()` function which is meant to initialize class and child class properties during each new child class creation; the `_class_init()` function, which is used to initialise the class only once (specifying what signals, arguments and virtual functions the class has and setting up global state); and the `_init()` function, which is used to initialise a specific instance of this type.

The plugin_init function

Once we have written code defining all the parts of the plugin, we need to write the plugin_init() function. This is a special function, which is called as soon as the plugin is loaded, and should return TRUE or FALSE depending on whether it loaded initialized any dependencies correctly. Also, in this function, any supported element type in the plugin should be registered.

```
static gboolean
plugin_init (GstPlugin *plugin)
{
  return gst_element_register (plugin, "my_filter",
          GST_RANK_NONE,
          GST_TYPE_MY_FILTER);
}

GST_PLUGIN_DEFINE (
  GST_VERSION_MAJOR,
  GST_VERSION_MINOR,
  "my_filter",
  "My filter plugin",
  plugin_init,
  VERSION,
  "LGPL",
  "GStreamer",
  "http://gstreamer.net/"
)
```

Note that the information returned by the plugin_init() function will be cached in a central registry. For this reason, it is important that the same information is always returned by the function: for example, it must not make element factories available based on runtime conditions. If an element can only work in certain conditions (for example, if the soundcard is not being used by some other process) this must be reflected by the element being unable to enter the READY state if unavailable, rather than the plugin attempting to deny existence of the plugin.

Chapter 4. Specifying the pads

As explained before, pads are the port through which data goes in and out of your element, and that makes them a very important item in the process of element creation. In the boilerplate code, we have seen how static pad templates take care of registering pad templates with the element class. Here, we will see how to create actual elements, use a `_setcaps ()`-functions to configure for a particular format and how to register functions to let data flow through the element.

In the element `_init ()` function, you create the pad from the pad template that has been registered with the element class in the `_base_init ()` function. After creating the pad, you have to set a `_setcaps ()` function pointer and optionally a `_getcaps ()` function pointer. Also, you have to set a `_chain ()` function pointer. Alternatively, pads can also operate in looping mode, which mans that they can pull data themselves. More on this topic later. After that, you have to register the pad with the element. This happens like this:

```
static gboolean  gst_my_filter_setcaps (GstPad        *pad,
       GstCaps       *caps);
static GstFlowReturn gst_my_filter_chain (GstPad        *pad,
       GstBuffer     *buf);
```

```
static void
gst_my_filter_init (GstMyFilter *filter, GstMyFilterClass *filter_klass)
{
  GstElementClass *klass = GST_ELEMENT_CLASS (filter_klass);

  /* pad through which data comes in to the element */
  filter->sinkpad = gst_pad_new_from_template (
 gst_element_class_get_pad_template (klass, "sink"), "sink");
  gst_pad_set_setcaps_function (filter->sinkpad, gst_my_filter_setcaps);
  gst_pad_set_chain_function (filter->sinkpad, gst_my_filter_chain);


  gst_element_add_pad (GST_ELEMENT (filter), filter->sinkpad);

  /* pad through which data goes out of the element */
  filter->srcpad = gst_pad_new_from_template (
 gst_element_class_get_pad_template (klass, "src"), "src");


  gst_element_add_pad (GST_ELEMENT (filter), filter->srcpad);

  /* properties initial value */
  filter->silent = FALSE;
}
```

The setcaps-function

The _setcaps ()-function is called during caps negotiation, which is discussed in great detail in Caps negotiation. This is the process where the linked pads decide on the streamtype that will transfer between them. A full list of type-definitions can be found in Chapter 12. A _link () receives a pointer to a GstCaps[1] struct that defines the proposed streamtype, and can respond with either "yes" (TRUE) or "no" (FALSE). If the element responds positively towards the streamtype, that type will be used on the pad. An example:

```
static gboolean
gst_my_filter_setcaps (GstPad  *pad,
         GstCaps *caps)
{
  GstStructure *structure = gst_caps_get_structure (caps, 0);
  GstMyFilter *filter = GST_MY_FILTER (GST_OBJECT_PARENT (pad));
  const gchar *mime;

  /* Since we're an audio filter, we want to handle raw audio
   * and from that audio type, we need to get the samplerate and
   * number of channels. */
  mime = gst_structure_get_name (structure);
  if (strcmp (mime, "audio/x-raw-int") != 0) {
    GST_WARNING ("Wrong mimetype %s provided, we only support %s",
   mime, "audio/x-raw-int");
    return FALSE;
  }

  /* we're a filter and don't touch the properties of the data.
   * That means we can set the given caps unmodified on the next
   * element, and use that negotiation return value as ours. */
  if (!gst_pad_set_caps (filter->srcpad, caps))
    return FALSE;

  /* Capsnego succeeded, get the stream properties for internal
   * usage and return success. */
  gst_structure_get_int (structure, "rate", &filter->samplerate);
  gst_structure_get_int (structure, "channels", &filter->channels);

  g_print ("Caps negotiation succeeded with %d Hz @ %d channels\n",
    filter->samplerate, filter->channels);

  return TRUE;
}
```

In here, we check the mimetype of the provided caps. Normally, you don't need to do that in your own plugin/element, because the core does that for you. We simply use it to show how to retrieve the mimetype from a provided set of caps. Types are stored in GstStructure [2] internally. A GstCaps [3] is nothing more than a small wrapper for 0 or more structures/types. From the structure, you can also retrieve properties, as is shown above with the function gst_structure_get_int ().

If your _link () function does not need to perform any specific operation (i.e. it will only forward caps), you can set it to gst_pad_proxy_link (). This is a link

forwarding function implementation provided by the core. It is useful for elements such as `identity`.

Notes

1. ../../gstreamer/html/gstreamer-GstCaps.html
2. ../../gstreamer/html/gstreamer-GstStructure.html
3. ../../gstreamer/html/gstreamer-GstCaps.html

Chapter 5. The chain function

The chain function is the function in which all data processing takes place. In the case of a simple filter, `_chain ()` functions are mostly linear functions - so for each incoming buffer, one buffer will go out, too. Below is a very simple implementation of a chain function:

```
static GstFlowReturn
gst_my_filter_chain (GstPad    *pad,
       GstBuffer *buf)
{
  GstMyFilter *filter = GST_MY_FILTER (GST_OBJECT_PARENT (pad));

  if (!filter->silent)
    g_print ("Have data of size %u bytes!\n", GST_BUFFER_SIZE (buf));

  return gst_pad_push (filter->srcpad, buf);
}
```

Obviously, the above doesn't do much useful. Instead of printing that the data is in, you would normally process the data there. Remember, however, that buffers are not always writable. In more advanced elements (the ones that do event processing), you may want to additionally specify an event handling function, which will be called when stream-events are sent (such as end-of-stream, discontinuities, tags, etc.).

```
static void
gst_my_filter_init (GstMyFilter * filter)
{
[..]
  gst_pad_set_event_function (filter->sinkpad,
      gst_my_filter_event);
[..]
}



static gboolean
gst_my_filter_event (GstPad   *pad,
       GstEvent *event)
{
  GstMyFilter *filter = GST_MY_FILTER (GST_OBJECT_PARENT (pad));

  switch (GST_EVENT_TYPE (event)) {
    case GST_EVENT_EOS:
      /* end-of-stream, we should close down all stream leftovers here */
      gst_my_filter_stop_processing (filter);
      break;
    default:
      break;
  }

  return gst_pad_event_default (pad, event);
}
```

```
static GstFlowReturn
gst_my_filter_chain (GstPad    *pad,
       GstBuffer *buf)
{
  GstMyFilter *filter = GST_MY_FILTER (gst_pad_get_parent (pad));
  GstBuffer *outbuf;

  outbuf = gst_my_filter_process_data (filter, buf);
  gst_buffer_unref (buf);
  if (!outbuf) {
    /* something went wrong - signal an error */
    GST_ELEMENT_ERROR (GST_ELEMENT (filter), STREAM, FAILED, (NULL), (NULL));
    return GST_FLOW_ERROR;
  }

  return gst_pad_push (filter->srcpad, outbuf);
}
```

In some cases, it might be useful for an element to have control over the input data rate, too. In that case, you probably want to write a so-called *loop-based* element. Source elements (with only source pads) can also be *get-based* elements. These concepts will be explained in the advanced section of this guide, and in the section that specifically discusses source pads.

Chapter 6. What are states?

A state describes whether the element instance is initialized, whether it is ready to transfer data and whether it is currently handling data. There are four states defined in `GStreamer`:

- GST_STATE_NULL
- GST_STATE_READY
- GST_STATE_PAUSED
- GST_STATE_PLAYING

which will from now on be referred to simply as “NULL”, “READY”, “PAUSED” and “PLAYING”.

GST_STATE_NULL is the default state of an element. In this state, it has not allocated any runtime resources, it has not loaded any runtime libraries and it can obviously not handle data.

GST_STATE_READY is the next state that an element can be in. In the READY state, an element has all default resources (runtime-libraries, runtime-memory) allocated. However, it has not yet allocated or defined anything that is stream-specific. When going from NULL to READY state (GST_STATE_CHANGE_NULL_TO_READY), an element should allocate any non-stream-specific resources and should load runtime-loadable libraries (if any). When going the other way around (from READY to NULL, GST_STATE_CHANGE_READY_TO_NULL), an element should unload these libraries and free all allocated resources. Examples of such resources are hardware devices. Note that files are generally streams, and these should thus be considered as stream-specific resources; therefore, they should *not* be allocated in this state.

GST_STATE_PAUSED is the state in which an element is ready to accept and handle data. For most elements this state is the same as PLAYING. The only exception to this rule are sink elements. Sink elements only accept one single buffer of data and then block. At this point the pipeline is ’prerolled’ and ready to render data immediately.

GST_STATE_PLAYING is the highest state that an element can be in. For most elements this state is exactly the same as PAUSED, they accept and process events and buffers with data. Only sink elements need to differentiate between PAUSED and PLAYING state. In PLAYING state, sink elements actually render incoming data, e.g. output audio to a sound card or render video pictures to an image sink.

Managing filter state

If at all possible, your element should derive from one of the new base classes (Pre-made base classes). There are ready-made general purpose base classes for different types of sources, sinks and filter/transformation elements. In addition to those, specialised base classes exist for audio and video elements and others.

If you use a base class, you will rarely have to handle state changes yourself. All you have to do is override the base class’s start() and stop() virtual functions (might be

called differently depending on the base class) and the base class will take care of everything for you.

If, however, you do not derive from a ready-made base class, but from GstElement or some other class not built on top of a base class, you will most likely have to implement your own state change function to be notified of state changes. This is definitively necessary if your plugin is a decoder or an encoder, as there are no base classes for decoders or encoders yet.

An element can be notified of state changes through a virtual function pointer. Inside this function, the element can initialize any sort of specific data needed by the element, and it can optionally fail to go from one state to another.

Do not g_assert for unhandled state changes; this is taken care of by the GstElement base class.

```
static GstStateChangeReturn
gst_my_filter_change_state (GstElement *element, GstStateChange transition);

static void
gst_my_filter_class_init (GstMyFilterClass *klass)
{
  GstElementClass *element_class = GST_ELEMENT_CLASS (klass);

  element_class->change_state = gst_my_filter_change_state;
}



static GstStateChangeReturn
gst_my_filter_change_state (GstElement *element, GstStateChange transition)
{
  GstStateChangeReturn ret = GST_STATE_CHANGE_SUCCESS;
  GstMyFilter *filter = GST_MY_FILTER (element);

  switch (transition) {
    case GST_STATE_CHANGE_NULL_TO_READY:
      if (!gst_my_filter_allocate_memory (filter))
        return GST_STATE_CHANGE_FAILURE;
      break;
    default:
      break;
  }

  ret = GST_ELEMENT_CLASS (parent_class)->change_state (element, transition);
  if (ret == GST_STATE_CHANGE_FAILURE)
    return ret;

  switch (transition) {
    case GST_STATE_CHANGE_READY_TO_NULL:
      gst_my_filter_free_memory (filter);
      break;
    default:
      break;
  }

  return ret;
```

```
}
```

Note that upwards (NULL=>READY, READY=>PAUSED, PAUSED=>PLAYING) and downwards (PLAYING=>PAUSED, PAUSED=>READY, READY=>NULL) state changes are handled in two separate blocks with the downwards state change handled only after we have chained up to the parent class's state change function. This is necessary in order to safely handle concurrent access by multiple threads.

The reason for this is that in the case of downwards state changes you don't want to destroy allocated resources while your plugin's chain function (for example) is still accessing those resources in another thread. Whether your chain function might be running or not depends on the state of your plugin's pads, and the state of those pads is closely linked to the state of the element. Pad states are handled in the GstElement class's state change function, including proper locking, that's why it is essential to chain up before destroying allocated resources.

Chapter 7. Adding Arguments

The primary and most important way of controlling how an element behaves, is through GObject properties. GObject properties are defined in the `_class_init ()` function. The element optionally implements a `_get_property ()` and a `_set_property ()` function. These functions will be notified if an application changes or requests the value of a property, and can then fill in the value or take action required for that property to change value internally.

```
/* properties */
enum {
  ARG_0,
  ARG_SILENT
  /* FILL ME */
};

static void gst_my_filter_set_property (GObject      *object,
       guint         prop_id,
       const GValue *value,
       GParamSpec   *pspec);
static void gst_my_filter_get_property (GObject      *object,
       guint         prop_id,
       GValue       *value,
       GParamSpec   *pspec);

static void
gst_my_filter_class_init (GstMyFilterClass *klass)
{
  GObjectClass *object_class = G_OBJECT_CLASS (klass);

  /* define properties */
  g_object_class_install_property (object_class, ARG_SILENT,
    g_param_spec_boolean ("silent", "Silent",
     "Whether to be very verbose or not",
     FALSE, G_PARAM_READWRITE | G_PARAM_STATIC_STRINGS));

  /* define virtual function pointers */
  object_class->set_property = gst_my_filter_set_property;
  object_class->get_property = gst_my_filter_get_property;
}

static void
gst_my_filter_set_property (GObject      *object,
       guint         prop_id,
       const GValue *value,
       GParamSpec   *pspec)
{
  GstMyFilter *filter = GST_MY_FILTER (object);

  switch (prop_id) {
    case ARG_SILENT:
      filter->silent = g_value_get_boolean (value);
      g_print ("Silent argument was changed to %s\n",
        filter->silent ? "true" : "false");
```

```
      break;
    default:
      G_OBJECT_WARN_INVALID_PROPERTY_ID (object, prop_id, pspec);
      break;
  }
}

static void
gst_my_filter_get_property (GObject    *object,
       guint       prop_id,
       GValue     *value,
       GParamSpec *pspec)
{
  GstMyFilter *filter = GST_MY_FILTER (object);

  switch (prop_id) {
    case ARG_SILENT:
      g_value_set_boolean (value, filter->silent);
      break;
    default:
      G_OBJECT_WARN_INVALID_PROPERTY_ID (object, prop_id, pspec);
      break;
  }
}
```

The above is a very simple example of how arguments are used. Graphical applications - for example GStreamer Editor - will use these properties and will display a user-controlleable widget with which these properties can be changed. This means that - for the property to be as user-friendly as possible - you should be as exact as possible in the definition of the property. Not only in defining ranges in between which valid properties can be located (for integers, floats, etc.), but also in using very descriptive (better yet: internationalized) strings in the definition of the property, and if possible using enums and flags instead of integers. The GObject documentation describes these in a very complete way, but below, we'll give a short example of where this is useful. Note that using integers here would probably completely confuse the user, because they make no sense in this context. The example is stolen from videotestsrc.

```
typedef enum {
  GST_VIDEOTESTSRC_SMPTE,
  GST_VIDEOTESTSRC_SNOW,
  GST_VIDEOTESTSRC_BLACK
} GstVideotestsrcPattern;

[..]

#define GST_TYPE_VIDEOTESTSRC_PATTERN (gst_videotestsrc_pattern_get_type ())
static GType
gst_videotestsrc_pattern_get_type (void)
{
  static GType videotestsrc_pattern_type = 0;

  if (!videotestsrc_pattern_type) {
    static GEnumValue pattern_types[] = {
```

```
      { GST_VIDEOTESTSRC_SMPTE, "smpte", "SMPTE 100% color bars" },
      { GST_VIDEOTESTSRC_SNOW,  "snow",  "Random (television snow)" },
      { GST_VIDEOTESTSRC_BLACK, "black", "0% Black" },
      { 0, NULL, NULL },
    };

    videotestsrc_pattern_type =
 g_enum_register_static ("GstVideotestsrcPattern",
    pattern_types);
  }

  return videotestsrc_pattern_type;
}

[..]

static void
gst_videotestsrc_class_init (GstvideotestsrcClass *klass)
{
[..]
  g_object_class_install_property (G_OBJECT_CLASS (klass), ARG_TYPE,
    g_param_spec_enum ("pattern", "Pattern",
         "Type of test pattern to generate",
         GST_TYPE_VIDEOTESTSRC_PATTERN, 1, G_PARAM_READWRITE |
                         G_PARAM_STATIC_STRINGS));
[..]
}
```

Chapter 8. Signals

GObject signals can be used to notify applications of events specific to this object. Note, however, that the application needs to be aware of signals and their meaning, so if you're looking for a generic way for application-element interaction, signals are probably not what you're looking for. In many cases, however, signals can be very useful. See the GObject documentation[1] for all internals about signals.

Notes

1. http://www.le-hacker.org/papers/gobject/index.html

Chapter 9. Building a Test Application

Often, you will want to test your newly written plugin in an as small setting as possible. Usually, `gst-launch` is a good first step at testing a plugin. If you have not installed your plugin in a directory that GStreamer searches, then you will need to set the plugin path. Either set GST_PLUGIN_PATH to the directory containing your plugin, or use the command-line option --gst-plugin-path. If you based your plugin off of the gst-plugin template, then this will look something like **gst-launch --gst-plugin-path=$HOME/gst-template/gst-plugin/src/.libs TESTPIPELINE** However, you will often need more testing features than gst-launch can provide, such as seeking, events, interactivity and more. Writing your own small testing program is the easiest way to accomplish this. This section explains - in a few words - how to do that. For a complete application development guide, see the Application Development Manual[1].

At the start, you need to initialize the `GStreamer` core library by calling `gst_init ()`. You can alternatively call `gst_init_with_popt_tables ()`, which will return a pointer to popt tables. You can then use libpopt to handle the given argument table, and this will finish the `GStreamer` intialization.

You can create elements using `gst_element_factory_make ()`, where the first argument is the element type that you want to create, and the second argument is a free-form name. The example at the end uses a simple filesource - decoder - soundcard output pipeline, but you can use specific debugging elements if that's necessary. For example, an `identity` element can be used in the middle of the pipeline to act as a data-to-application transmitter. This can be used to check the data for misbehaviours or correctness in your test application. Also, you can use a `fakesink` element at the end of the pipeline to dump your data to the stdout (in order to do this, set the `dump` property to TRUE). Lastly, you can use the `efence` element (indeed, an eletric fence memory debugger wrapper element) to check for memory errors.

During linking, your test application can use fixation or filtered caps as a way to drive a specific type of data to or from your element. This is a very simple and effective way of checking multiple types of input and output in your element.

Running the pipeline happens through the `gst_bin_iterate ()` function. Note that during running, you should connect to at least the "error" and "eos" signals on the pipeline and/or your plugin/element to check for correct handling of this. Also, you should add events into the pipeline and make sure your plugin handles these correctly (with respect to clocking, internal caching, etc.).

Never forget to clean up memory in your plugin or your test application. When going to the NULL state, your element should clean up allocated memory and caches. Also, it should close down any references held to possible support libraries. Your application should `unref ()` the pipeline and make sure it doesn't crash.

```
#include <gst/gst.h>

static gboolean
bus_call (GstBus     *bus,
   GstMessage *msg,
   gpointer    data)
{
  GMainLoop *loop = data;
```

```
  switch (GST_MESSAGE_TYPE (msg)) {
    case GST_MESSAGE_EOS:
      g_print ("End-of-stream\n");
      g_main_loop_quit (loop);
      break;
    case GST_MESSAGE_ERROR: {
      gchar *debug = NULL;
      GError *err = NULL;

      gst_message_parse_error (msg, &err, &debug);

      g_print ("Error: %s\n", err->message);
      g_error_free (err);

      if (debug) {
        g_print ("Debug deails: %s\n", debug);
        g_free (debug);
      }

      g_main_loop_quit (loop);
      break;
    }
    default:
      break;
  }

  return TRUE;
}

gint
main (gint   argc,
      gchar *argv[])
{
  GstStateChangeReturn ret;
  GstElement *pipeline, *filesrc, *decoder, *filter, *sink;
  GstElement *convert1, *convert2, *resample;
  GMainLoop *loop;
  GstBus *bus;

  /* initialization */
  gst_init (&argc, &argv);
  loop = g_main_loop_new (NULL, FALSE);
  if (argc != 2) {
    g_print ("Usage: %s <mp3 filename>\n", argv[0]);
    return 01;
  }

  /* create elements */
  pipeline = gst_pipeline_new ("my_pipeline");

  /* watch for messages on the pipeline's bus (note that this will only
   * work like this when a GLib main loop is running) */
  bus = gst_pipeline_get_bus (GST_PIPELINE (pipeline));
  gst_bus_add_watch (bus, bus_call, loop);
  gst_object_unref (bus);
```

```
filesrc  = gst_element_factory_make ("filesrc", "my_filesource");
decoder  = gst_element_factory_make ("mad", "my_decoder");

/* putting an audioconvert element here to convert the output of the
 * decoder into a format that my_filter can handle (we are assuming it
 * will handle any sample rate here though) */
convert1 = gst_element_factory_make ("audioconvert", "audioconvert1");

/* use "identity" here for a filter that does nothing */
filter   = gst_element_factory_make ("my_filter", "my_filter");

/* there should always be audioconvert and audioresample elements before
 * the audio sink, since the capabilities of the audio sink usually vary
 * depending on the environment (output used, sound card, driver etc.) */
convert2 = gst_element_factory_make ("audioconvert", "audioconvert2");
resample = gst_element_factory_make ("audioresample", "audioresample");
sink     = gst_element_factory_make ("osssink", "audiosink");

if (!sink || !decoder) {
  g_print ("Decoder or output could not be found - check your install\n");
  return -1;
} else if (!convert1 || !convert2 || !resample) {
  g_print ("Could not create audioconvert or audioresample element, "
           "check your installation\n");
  return -1;
} else if (!filter) {
  g_print ("Your self-written filter could not be found. Make sure it "
           "is installed correctly in $(libdir)/gstreamer-0.10/ or "
           "~/.gstreamer-0.10/plugins/ and that gst-inspect-0.10 lists it. "
           "If it doesn't, check with 'GST_DEBUG=*:2 gst-inspect-0.10' for "
           "the reason why it is not being loaded.");
  return -1;
}

g_object_set (G_OBJECT (filesrc), "location", argv[1], NULL);

gst_bin_add_many (GST_BIN (pipeline), filesrc, decoder, convert1, filter,
                  convert2, resample, sink, NULL);

/* link everything together */
if (!gst_element_link_many (filesrc, decoder, convert1, filter, convert2,
                            resample, sink, NULL)) {
  g_print ("Failed to link one or more elements!\n");
  return -1;
}

/* run */
ret = gst_element_set_state (pipeline, GST_STATE_PLAYING);
if (ret == GST_STATE_CHANGE_FAILURE) {
  GstMessage *msg;

  g_print ("Failed to start up pipeline!\n");

  /* check if there is an error message with details on the bus */
  msg = gst_bus_poll (bus, GST_MESSAGE_ERROR, 0);
```

```
    if (msg) {
      GError *err = NULL;

      gst_message_parse_error (msg, &err, NULL);
      g_print ("ERROR: %s\n", err->message);
      g_error_free (err);
      gst_message_unref (msg);
    }
    return -1;
  }

  g_main_loop_run (loop);

  /* clean up */
  gst_element_set_state (pipeline, GST_STATE_NULL);
  gst_object_unref (pipeline);

  return 0;
}
```

Notes

1. ../../manual/html/index.html

Chapter 10. Caps negotiation

Caps negotiation is the process where elements configure themselves and each other for streaming a particular media format over their pads. Since different types of elements have different requirements for the media formats they can negotiate to, it is important that this process is generic and implements all those use cases correctly.

In this chapter, we will discuss downstream negotiation and upstream negotiation from a pipeline perspective, implicating the responsibilities of different types of elements in a pipeline, and we will introduce the concept of *fixed caps*.

Caps negotiation use cases

Let's take the case of a file source, linked to a demuxer, linked to a decoder, linked to a converter with a caps filter and finally an audio output. When dataflow originally starts, the demuxer will parse the file header (e.g. the Ogg headers), and notice that there is, for example, a Vorbis stream in this Ogg file. Noticing that, it will create an output pad for the Vorbis elementary stream and set a Vorbis-caps on it. Lastly, it adds the pad. As of this point, the pad is ready to be used to stream data, and so the Ogg demuxer is now done. This pad is *not* re-negotiatable, since the type of the data stream is embedded within the data.

The Vorbis decoder will decode the Vorbis headers and the Vorbis data coming in on its sinkpad. Now, some decoders may be able to output in multiple output formats, for example both 16-bit integer output and floating-point output, whereas other decoders may be able to only decode into one specific format, e.g. only floating-point (32-bit) audio. Those two cases have consequences for how caps negotiation should be implemented in this decoder element. In the one case, it is possible to use fixed caps, and you're done. In the other case, however, you should implement the possibility for *renegotiation* in this element, which is the possibility for the data format to be changed to another format at some point in the future. We will discuss how to do this in one of the sections further on in this chapter.

The filter can be used by applications to force, for example, a specific channel configuration (5.1/surround or 2.0/stereo), on the pipeline, so that the user can enjoy sound coming from all its speakers. The audio sink, in this example, is a standard ALSA output element (alsasink). The converter element supports any-to-any, and the filter will make sure that only a specifically wanted channel configuration streams through this link (as provided by the user's channel configuration preference). By changing this preference while the pipeline is running, some elements will have to renegotiate *while the pipeline is running*. This is done through upstream caps renegotiation. That, too, will be discussed in detail in a section further below.

In order for caps negotiation on non-fixed links to work correctly, pads can optionally implement a function that tells peer elements what formats it supports and/or preferes. When upstream renegotiation is triggered, this becomes important.

Downstream elements are notified of a newly set caps only when data is actually passing their pad. This is because caps is attached to buffers during dataflow. So when the vorbis decoder sets a caps on its source pad (to configure the output format), the converter will not yet be notified. Instead, the converter will only be notified when the decoder pushes a buffer over its source pad to the converter. Right before calling the chain-function in the converter, `GStreamer` will check whether the for-

mat that was previously negotiated still applies to this buffer. If not, it first calls the setcaps-function of the converter to configure it for the new format. Only after that will it call the chain function of the converter.

Fixed caps

The simplest way in which to do caps negotiation is setting a fixed caps on a pad. After a fixed caps has been set, the pad can not be renegotiated from the outside. The only way to reconfigure the pad is for the element owning the pad to set a new fixed caps on the pad. Fixed caps is a setup property for pads, called when creating the pad:

```
[..]
  pad = gst_pad_new_from_template (..);
  gst_pad_use_fixed_caps (pad);
[..]
```

The fixed caps can then be set on the pad by calling `gst_pad_set_caps ()`.

```
[..]
  caps = gst_caps_new_simple ("audio/x-raw-float",
      "width", G_TYPE_INT, 32,
      "endianness", G_TYPE_INT, G_BYTE_ORDER,
      "buffer-frames", G_TYPE_INT, <bytes-per-frame>,
      "rate", G_TYPE_INT, <samplerate>,
      "channels", G_TYPE_INT, <num-channels>, NULL);
  if (!gst_pad_set_caps (pad, caps)) {
    GST_ELEMENT_ERROR (element, CORE, NEGOTIATION, (NULL),
        ("Some debug information here"));
    return GST_FLOW_ERROR;
  }
[..]
```

Elements that could implement fixed caps (on their source pads) are, in general, all elements that are not renegotiatable. Examples include:

- A typefinder, since the type found is part of the actual data stream and can thus not be re-negotiated.
- Pretty much all demuxers, since the contained elementary data streams are defined in the file headers, and thus not renegotiatable.
- Some decoders, where the format is embedded in the datastream and not part of the peercaps *and* where the decoder itself is not reconfigureable, too.

All other elements that need to be configured for the format should implement full caps negotiation, which will be explained in the next few sections.

Downstream caps negotiation

Downstream negotiation takes place when a format needs to be set on a source pad to configure the output format, but this element allows renegotiation because its format is configured on the sinkpad caps, or because it supports multiple formats. The requirements for doing the actual negotiation differ slightly.

Negotiating caps embedded in input caps

Many elements, particularly effects and converters, will be able to parse the format of the stream from their input caps, and decide the output format right at that time already. When renegotiation takes place, some may merely need to "forward" the renegotiation backwards upstream (more on that later). For those elements, all (downstream) caps negotiation can be done in something that we call the `_setcaps ()` function. This function is called when a buffer is pushed over a pad, but the format on this buffer is not the same as the format that was previously negotiated (or, similarly, no format was negotiated yet so far).

In the `_setcaps ()`-function, the element can forward the caps to the next element and, if that pad accepts the format too, the element can parse the relevant parameters from the caps and configure itself internally. The caps passed to this function is *always* a subset of the template caps, so there's no need for extensive safety checking. The following example should give a clear indication of how such a function can be implemented:

```
static gboolean
gst_my_filter_setcaps (GstPad  *pad,
         GstCaps *caps)
{
  GstMyFilter *filter = GST_MY_FILTER (GST_OBJECT_PARENT (pad));
  GstStructure *s;

  /* forward-negotiate */
  if (!gst_pad_set_caps (filter->srcpad, caps))
    return FALSE;

  /* negotiation succeeded, so now configure ourselves */
  s = gst_caps_get_structure (caps, 0);
  gst_structure_get_int (s, "rate", &filter->samplerate);
  gst_structure_get_int (s, "channels", &filter->channels);

  return TRUE;
}
```

There may also be cases where the filter actually is able to *change* the format of the stream. In those cases, it will negotiate a new format. Obviously, the element should first attempt to configure "pass-through", which means that it does not change the stream's format. However, if that fails, then it should call `gst_pad_get_allowed_caps ()` on its sourcepad to get a list of supported formats on the outputs, and pick the first. The return value of that function is guaranteed to be a subset of the template caps.

Let's look at the example of an element that can convert between samplerates, so where input and output samplerate don't have to be the same:

```
static gboolean
gst_my_filter_setcaps (GstPad  *pad,
        GstCaps *caps)
{
  GstMyFilter *filter = GST_MY_FILTER (GST_OBJECT_PARENT (pad));

  if (gst_pad_set_caps (filter->sinkpad, caps)) {
    filter->passthrough = TRUE;
  } else {
    GstCaps *othercaps, *newcaps;
    GstStructure *s = gst_caps_get_structure (caps, 0), *others;

    /* no passthrough, setup internal conversion */
    gst_structure_get_int (s, "channels", &filter->channels);
    othercaps = gst_pad_get_allowed_caps (filter->srcpad);
    others = gst_caps_get_structure (othercaps, 0);
    gst_structure_set (others,
        "channels", G_TYPE_INT, filter->channels, NULL);

    /* now, the samplerate value can optionally have multiple values, so
     * we "fixate" it, which means that one fixed value is chosen */
    newcaps = gst_caps_copy_nth (othercaps, 0);
    gst_caps_unref (othercaps);
    gst_pad_fixate_caps (filter->srcpad, newcaps);
    if (!gst_pad_set_caps (filter->srcpad, newcaps))
      return FALSE;

    /* we are now set up, configure internally */
    filter->passthrough = FALSE;
    gst_structure_get_int (s, "rate", &filter->from_samplerate);
    others = gst_caps_get_structure (newcaps, 0);
    gst_structure_get_int (others, "rate", &filter->to_samplerate);
  }

  return TRUE;
}

static GstFlowReturn
gst_my_filter_chain (GstPad    *pad,
       GstBuffer *buf)
{
  GstMyFilter *filter = GST_MY_FILTER (GST_OBJECT_PARENT (pad));
  GstBuffer *out;

  /* push on if in passthrough mode */
  if (filter->passthrough)
    return gst_pad_push (filter->srcpad, buf);

  /* convert, push */
  out = gst_my_filter_convert (filter, buf);
  gst_buffer_unref (buf);
```

```
  return gst_pad_push (filter->srcpad, out);
}
```

Parsing and setting caps

Other elements, such as certain types of decoders, will not be able to parse the caps from their input, simply because the input format does not contain the information required to know the output format yet; rather, the data headers need to be parsed, too. In many cases, fixed-caps will be enough, but in some cases, particularly in cases where such decoders are renegotiatable, it is also possible to use full caps negotiation.

Fortunately, the code required to do so is very similar to the last code example in Negotiating caps embedded in input caps, with the difference being that the caps is selected in the `_chain ()`-function rather than in the `_setcaps ()`-function. The rest, as for getting all allowed caps from the source pad, fixating and such, is all the same. Re-negotiation, which will be handled in the next section, is very different for such elements, though.

Upstream caps (re)negotiation

Upstream negotiation's primary use is to renegotiate (part of) an already-negotiated pipeline to a new format. Some practical examples include to select a different video size because the size of the video window changed, and the video output itself is not capable of rescaling, or because the audio channel configuration changed.

Upstream caps renegotiation is done in the `gst_pad_alloc_buffer ()`-function. The idea here is that an element requesting a buffer from downstream, has to specify the type of that buffer. If renegotiation is to take place, this type will no longer apply, and the downstream element will set a new caps on the provided buffer. The element should then reconfigure itself to push buffers with the returned caps. The source pad's setcaps will be called once the buffer is pushed.

It is important to note here that different elements actually have different responsibilities here:

- Elements should implement a "padalloc"-function in order to be able to change format on renegotiation. This is also true for filters and converters.
- Elements should allocate new buffers using `gst_pad_alloc_buffer ()`.
- Elements that are renegotiatable should implement a "setcaps"-function on their sourcepad as well.

Unfortunately, not all details here have been worked out yet, so this documentation is incomplete. FIXME.

Implementing a getcaps function

A `_getcaps ()`-function is called when a peer element would like to know which formats this element supports, and in what order of preference. The return value should be all formats that this elements supports, taking into account limitations of peer elements further downstream or upstream, sorted by order of preference, highest preference first.

```
static GstCaps *
gst_my_filter_getcaps (GstPad *pad)
{
  GstMyFilter *filter = GST_MY_FILTER (GST_OBJECT_PARENT (pad));
  GstPad *otherpad = (pad == filter->srcpad) ? filter->sinkpad :
        filter->srcpad;
  GstCaps *othercaps = gst_pad_get_allowed_caps (otherpad), *caps;
  gint i;

  /* We support *any* samplerate, indifferent from the samplerate
   * supported by the linked elements on both sides. */
  for (i = 0; i < gst_caps_get_size (othercaps); i++) {
    GstStructure *structure = gst_caps_get_structure (othercaps, i);

    gst_structure_remove_field (structure, "rate");
  }
  caps = gst_caps_intersect (othercaps, gst_pad_get_pad_template_caps (pad));
  gst_caps_unref (othercaps);

  return caps;
}
```

Using all the knowledge you've acquired by reading this chapter, you should be able to write an element that does correct caps negotiation. If in doubt, look at other elements of the same type in our git repository to get an idea of how they do what you want to do.

Chapter 11. Different scheduling modes

Scheduling is, in short, a method for making sure that every element gets called once in a while to process data and prepare data for the next element. Likewise, a kernel has a scheduler for processes, and your brain is a very complex scheduler too in a way. Randomly calling elements' chain functions won't bring us far, however, so you'll understand that the schedulers in `GStreamer` are a bit more complex than this. However, as a start, it's a nice picture.

So far, we have only discussed `_chain ()`-operating elements, i.e. elements that have a chain-function set on their sink pad and push buffers on their source pad(s). Pads (or elements) can also operate in two other scheduling modes, however. In this chapter, we will discuss what those scheduling modes are, how they can be enabled and in what cases they are useful. The other two scheduling modes are random access (`_getrange ()`-based) or task-runner (which means that this element is the driving force in the pipeline) mode.

The pad activation stage

The stage in which `GStreamer` decides in what scheduling mode the various elements will operate, is called the pad-activation stage. In this stage, `GStreamer` will query the scheduling capabilities (i.e. it will see in what modes each particular element/pad can operate) and decide on the optimal scheduling composition for the pipeline. Next, each pad will be notified of the scheduling mode that was assigned to it, and after that the pipeline will start running.

Pads can be assigned one of three modes, each mode putting several prerequisites on the pads. Pads should implement a notification function (`gst_pad_set_activatepull_function ()` and `gst_pad_set_activatepush_function ()`) to be notified of the scheduling mode assignment. Also, sinkpads assigned to do pull-based scheduling mode should start and stop their task in this function.

- If all pads of an element are assigned to do "push"-based scheduling, then this means that data will be pushed by upstream elements to this element using the sinkpads `_chain ()`-function. Prerequisites for this scheduling mode are that a chain-function was set for each sinkpad using`gst_pad_set_chain_function ()` and that all downstream elements operate in the same mode. Pads are assigned to do push-based scheduling in sink-to-source element order, and within an element first sourcepads and then sinkpads. Sink elements can operate in this mode if their sinkpad is activated for push-based scheduling. Source elements cannot be chain-based.
- Alternatively, sinkpads can be the driving force behind a pipeline by operating in "pull"-based mode, while the sourcepads of the element still operate in push-based mode. In order to be the driving force, those pads start a `GstTask` when their pads are being activated. This task is a thread, which will call a function specified by the element. When called, this function will have random data access (through `gst_pad_get_range ()`) over all sinkpads, and can push data over the sourcepads, which effectively means that this element controls dataflow in the pipeline. Prerequisites for this mode are that all downstream elements can act in

chain-based mode, and that all upstream elements allow random access (see below). Source elements can be told to act in this mode if their sourcepads are activated in push-based fashion. Sink elements can be told to act in this mode when their sinkpads are activated in pull-mode.

- lastly, all pads in an element can be assigned to act in pull-mode. too. However, contrary to the above, this does not mean that they start a task on their own. Rather, it means that they are pull slave for the downstream element, and have to provide random data access to it from their `_get_range ()`-function. Requiremenents are that the a `_get_range ()`-function was set on this pad using the function `gst_pad_set_getrange_function ()`. Also, if the element has any sinkpads, all those pads (and thereby their peers) need to operate in random access mode, too. Note that the element is supposed to activate those elements itself! `GStreamer` will not do that for you.

In the next two sections, we will go closer into pull-based scheduling (elements/pads driving the pipeline, and elements/pads providing random access), and some specific use cases will be given.

Pads driving the pipeline

Sinkpads assigned to operate in pull-based mode, while none of its sourcepads operate in pull-based mode (or it has no sourcepads), can start a task that will drive the pipeline dataflow. Within this function, those elements have random access over all of their sinkpads, and push data over their sourcepads. This can come in useful for several different kinds of elements:

- Demuxers, parsers and certain kinds of decoders where data comes in unparsed (such as MPEG-audio or video streams), since those will prefer byte-exact (random) access from their input. If possible, however, such elements should be prepared to operate in chain-based mode, too.
- Certain kind of audio outputs, which require control over their input dataflow, such as the Jack sound server.

In order to start this task, you will need to create it in the activation function.

```
#include "filter.h"
#include <string.h>

static gboolean gst_my_filter_activate (GstPad      * pad);
static gboolean gst_my_filter_activate_pull (GstPad  * pad,
      gboolean      active);
static void gst_my_filter_loop (GstMyFilter * filter);

GST_BOILERPLATE (GstMyFilter, gst_my_filter, GstElement, GST_TYPE_ELEMENT);



static void
gst_my_filter_init (GstMyFilter * filter)
{
```

```
[..]

  gst_pad_set_activate_function (filter->sinkpad, gst_my_filter_activate);
  gst_pad_set_activatepull_function (filter->sinkpad,
      gst_my_filter_activate_pull);


[..]
}

[..]

static gboolean
gst_my_filter_activate (GstPad * pad)
{
  if (gst_pad_check_pull_range (pad)) {
    return gst_pad_activate_pull (pad, TRUE);
  } else {
    return FALSE;
  }
}

static gboolean
gst_my_filter_activate_pull (GstPad  *pad,
        gboolean active)
{
  GstMyFilter *filter = GST_MY_FILTER (GST_OBJECT_PARENT (pad));

  if (active) {
    filter->offset = 0;
    return gst_pad_start_task (pad,
        (GstTaskFunction) gst_my_filter_loop, filter);
  } else {
    return gst_pad_stop_task (pad);
  }
}
```

Once started, your task has full control over input and output. The most simple case of a task function is one that reads input and pushes that over its source pad. It's not all that useful, but provides some more flexibility than the old chain-based case that we've been looking at so far.

```
#define BLOCKSIZE 2048

static void
gst_my_filter_loop (GstMyFilter * filter)
{
  GstFlowReturn ret;
  guint64 len;
  GstFormat fmt = GST_FORMAT_BYTES;
  GstBuffer *buf = NULL;

  if (!gst_pad_query_duration (filter->sinkpad, &fmt, &len)) {
    GST_DEBUG_OBJECT (filter, "failed to query duration, pausing");
    goto stop;
```

```
  }

   if (filter->offset >= len) {
    GST_DEBUG_OBJECT (filter, "at end of input, sending EOS, pausing");
    gst_pad_push_event (filter->srcpad, gst_event_new_eos ());
    goto stop;
  }

  /* now, read BLOCKSIZE bytes from byte offset filter->offset */
  ret = gst_pad_pull_range (filter->sinkpad, filter->offset,
      BLOCKSIZE, &buf);

  if (ret != GST_FLOW_OK) {
    GST_DEBUG_OBJECT (filter, "pull_range failed: %s", gst_flow_get_name (ret));
    goto stop;
  }

  /* now push buffer downstream */
  ret = gst_pad_push (filter->srcpad, buf);

  buf = NULL; /* gst_pad_push() took ownership of buffer */

  if (ret != GST_FLOW_OK) {
    GST_DEBUG_OBJECT (filter, "pad_push failed: %s", gst_flow_get_name (ret));
    goto stop;
  }

  /* everything is fine, increase offset and wait for us to be called again */
  filter->offset += BLOCKSIZE;
  return;

stop:
  GST_DEBUG_OBJECT (filter, "pausing task");
  gst_pad_pause_task (filter->sinkpad);
}
```

Providing random access

In the previous section, we have talked about how elements (or pads) that are assigned to drive the pipeline using their own task, have random access over their sinkpads. This means that all elements linked to those pads (recursively) need to provide random access functions. Requesting random access is done using the function `gst_pad_pull_range ()`, which requests a buffer of a specified size and offset. Source pads implementing and assigned to do random access will have a `_get_range ()`-function set using `gst_pad_set_getrange_function ()`, and that function will be called when the peer pad requests some data. The element is then responsible for seeking to the right offset and providing the requested data. Several elements can implement random access:

- Data sources, such as a file source, that can provide data from any offset with reasonable low latency.

- Filters that would like to provide a pull-based-like scheduling mode over the whole pipeline. Note that elements assigned to do random access-based scheduling are themselves responsible for assigning this scheduling mode to their upstream peers! `GStreamer` will not do that for you.
- Parsers who can easily provide this by skipping a small part of their input and are thus essentially "forwarding" random access requests literally without any own processing involved. Examples include tag readers (e.g. ID3) or single output parsers, such as a WAVE parser.

The following example will show how a `_get_range ()`-function can be implemented in a source element:

```
#include "filter.h"
static GstFlowReturn
  gst_my_filter_get_range (GstPad     * pad,
      guint64      offset,
      guint        length,
      GstBuffer ** buf);

GST_BOILERPLATE (GstMyFilter, gst_my_filter, GstElement, GST_TYPE_ELEMENT);



static void
gst_my_filter_init (GstMyFilter * filter)
{
  GstElementClass *klass = GST_ELEMENT_GET_CLASS (filter);

  filter->srcpad = gst_pad_new_from_template (
        gst_element_class_get_pad_template (klass, "src"), "src");
  gst_pad_set_getrange_function (filter->srcpad,
      gst_my_filter_get_range);
  gst_element_add_pad (GST_ELEMENT (filter), filter->srcpad);

[..]
}

static gboolean
gst_my_filter_get_range (GstPad     * pad,
    guint64      offset,
    guint        length,
    GstBuffer ** buf)
{

  GstMyFilter *filter = GST_MY_FILTER (GST_OBJECT_PARENT (pad));

  [.. here, you would fill *buf ..]

  return GST_FLOW_OK;
}
```

In practice, many elements that could theoretically do random access, may in practice often be assigned to do push-based scheduling anyway, since there is no downstream

element able to start its own task. Therefore, in practice, those elements should implement both a `_get_range ()`-function and a `_chain ()`-function (for filters and parsers) or a `_get_range ()`-function and be prepared to start their own task by providing `_activate_* ()`-functions (for source elements), so that `GStreamer` can decide for the optimal scheduling mode and have it just work fine in practice.

Chapter 12. Types and Properties

There is a very large set of possible types that may be used to pass data between elements. Indeed, each new element that is defined may use a new data format (though unless at least one other element recognises that format, it will be most likely be useless since nothing will be able to link with it).

In order for types to be useful, and for systems like autopluggers to work, it is necessary that all elements agree on the type definitions, and which properties are required for each type. The `GStreamer` framework itself simply provides the ability to define types and parameters, but does not fix the meaning of types and parameters, and does not enforce standards on the creation of new types. This is a matter for a policy to decide, not technical systems to enforce.

For now, the policy is simple:

- Do not create a new type if you could use one which already exists.
- If creating a new type, discuss it first with the other `GStreamer` developers, on at least one of: IRC, mailing lists.
- Try to ensure that the name for a new format is as unlikely to conflict with anything else created already, and is not a more generalised name than it should be. For example: "audio/compressed" would be too generalised a name to represent audio data compressed with an mp3 codec. Instead "audio/mp3" might be an appropriate name, or "audio/compressed" could exist and have a property indicating the type of compression used.
- Ensure that, when you do create a new type, you specify it clearly, and get it added to the list of known types so that other developers can use the type correctly when writing their elements.

Building a Simple Format for Testing

If you need a new format that has not yet been defined in our List of Defined Types, you will want to have some general guidelines on mimetype naming, properties and such. A mimetype would ideally be one defined by IANA; else, it should be in the form type/x-name, where type is the sort of data this mimetype handles (audio, video, ...) and name should be something specific for this specific type. Audio and video mimetypes should try to support the general audio/video properties (see the list), and can use their own properties, too. To get an idea of what properties we think are useful, see (again) the list.

Take your time to find the right set of properties for your type. There is no reason to hurry. Also, experimenting with this is generally a good idea. Experience learns that theoretically thought-out types are good, but they still need practical use to assure that they serve their needs. Make sure that your property names do not clash with similar properties used in other types. If they match, make sure they mean the same thing; properties with different types but the same names are *not* allowed.

Typefind Functions and Autoplugging

With only *defining* the types, we're not yet there. In order for a random data file to be recognized and played back as such, we need a way of recognizing their type out of the blue. For this purpose, "typefinding" was introduced. Typefinding is the process of detecting the type of a datastream. Typefinding consists of two separate parts: first, there's an unlimited number of functions that we call *typefind functions*, which are each able to recognize one or more types from an input stream. Then, secondly, there's a small engine which registers and calls each of those functions. This is the typefind core. On top of this typefind core, you would normally write an autoplugger, which is able to use this type detection system to dynamically build a pipeline around an input stream. Here, we will focus only on typefind functions.

A typefind function usually lives in `gst-plugins-base/gst/typefind/gsttypefindfunctions.c`, unless there's a good reason (like library dependencies) to put it elsewhere. The reason for this centralization is to reduce the number of plugins that need to be loaded in order to detect a stream's type. Below is an example that will recognize AVI files, which start with a "RIFF" tag, then the size of the file and then an "AVI " tag:

```
static void
gst_my_typefind_function (GstTypeFind *tf,
     gpointer     data)
{
  guint8 *data = gst_type_find_peek (tf, 0, 12);

  if (data &&
      GUINT32_FROM_LE (&((guint32 *) data)[0]) == GST_MAKE_FOURCC ('R','I','F','F')
      GUINT32_FROM_LE (&((guint32 *) data)[2]) == GST_MAKE_FOURCC ('A','V','I',' '))
    gst_type_find_suggest (tf, GST_TYPE_FIND_MAXIMUM,
      gst_caps_new_simple ("video/x-msvideo", NULL));
  }
}

static gboolean
plugin_init (GstPlugin *plugin)
{
  static gchar *exts[] = { "avi", NULL };
  if (!gst_type_find_register (plugin, "", GST_RANK_PRIMARY,
          gst_my_typefind_function, exts,
          gst_caps_new_simple ("video/x-msvideo",
          NULL), NULL))
    return FALSE;
}
```

Note that `gst-plugins/gst/typefind/gsttypefindfunctions.c` has some simplification macros to decrease the amount of code. Make good use of those if you want to submit typefinding patches with new typefind functions.

Autoplugging has been discussed in great detail in the Application Development Manual.

List of Defined Types

Below is a list of all the defined types in `GStreamer`. They are split up in separate tables for audio, video, container, subtitle and other types, for the sake of readability. Below each table might follow a list of notes that apply to that table. In the definition of each type, we try to follow the types and rules as defined by IANA[1] for as far as possible.

Jump directly to a specific table:

- Table of Audio Types
- Table of Video Types
- Table of Container Types
- Table of Subtitle Types
- Table of Other Types

Note that many of the properties are not *required*, but rather *optional* properties. This means that most of these properties can be extracted from the container header, but that - in case the container header does not provide these - they can also be extracted by parsing the stream header or the stream content. The policy is that your element should provide the data that it knows about by only parsing its own content, not another element's content. Example: the AVI header provides samplerate of the contained audio stream in the header. MPEG system streams don't. This means that an AVI stream demuxer would provide samplerate as a property for MPEG audio streams, whereas an MPEG demuxer would not. A decoder needing this data would require a stream parser in between two extract this from the header or calculate it from the stream.

Table 12-1. Table of Audio Types

Mime Type	Description	Property	Property Type	Property Values	Property Description
All audio types.					
audio/*	*All audio types*	rate	integer	greater than 0	The sample rate of the data, in samples (per channel) per second.
		channels	integer	greater than 0	The number of channels of audio data.
All raw audio types.					

Mime Type	Description	Property	Property Type	Property Values	Property Description
audio/x-raw-int	Unstructured and uncompressed raw fixed-integer audio data.	endianness	integer	G_BIG_ENDIAN (4321) or G_LITTLE_ENDIAN (1234)	The order of bytes in a sample. The value G_LITTLE_ENDIAN (1234) means "little-endian" (byte-order is "least significant byte first"). The value G_BIG_ENDIAN (4321) means "big-endian" (byte order is "most significant byte first").
		signed	boolean	TRUE or FALSE	Whether the values of the integer samples are signed or not. Signed samples use one bit to indicate sign (negative or positive) of the value. Unsigned samples are always positive.
		width	integer	greater than 0	Number of bits allocated per sample.
		depth	integer	greater than 0	The number of bits used per sample. This must be less than or equal to the width: If the depth is less than the width, the low bits are assumed to be the ones used. For example, a width of 32 and a depth of 24 means that each sample is stored in a 32 bit word, but only the low 24 bits are actually used.
audio/x-raw-float	Unstructured and uncompressed raw floating-point audio data.	endianness	integer	G_BIG_ENDIAN (4321) or G_LITTLE_ENDIAN (1234)	The order of bytes in a sample. The value G_LITTLE_ENDIAN (1234) means "little-endian" (byte-order is "least significant byte first"). The value G_BIG_ENDIAN (4321) means "big-endian" (byte order is "most significant byte first").

Mime Type	Description	Property	Property Type	Property Values	Property Description
		width	integer	greater than 0	The amount of bits used and allocated per sample.
All encoded audio types.					
audio/x-ac3	AC-3 or A52 audio streams.				There are currently no specific properties defined or needed for this type.
audio/x-adpcm	ADPCM Audio streams.	layout	string	"quicktime", "dvi", "microsoft" or "4xm".	The layout defines the packing of the samples in the stream. In ADPCM, most formats store multiple samples per channel together. This number of samples differs per format, hence the different layouts. On the long term, we probably want this variable to die and use something more descriptive, but this will do for now.
		block_align	integer	Any	Chunk buffer size.
audio/x-cinepak	Audio as provided in a Cinepak (Quicktime) stream.				There are currently no specific properties defined or needed for this type.

Mime Type	Description	Property	Property Type	Property Values	Property Description
audio/x-dv	Audio as provided in a Digital Video stream.				There are currently no specific properties defined or needed for this type.
audio/x-flac	Free Lossless Audio codec (FLAC).				There are currently no specific properties defined or needed for this type.
audio/x-gsm	Data encoded by the GSM codec.				There are currently no specific properties defined or needed for this type.
audio/x-alaw	A-Law Audio.				There are currently no specific properties defined or needed for this type.
audio/x-mulaw	Mu-Law Audio.				There are currently no specific properties defined or needed for this type.
audio/x-mace	MACE Audio (used in Quicktime).	maceversion	integer	3 or 6	The version of the MACE audio codec used to encode the stream.

Mime Type	Description	Property	Property Type	Property Values	Property Description
audio/mpeg	Audio data compressed using the MPEG audio encoding schem.	mpegversion	integer	1, 2 or 4	The MPEG-version used for encoding the data. The value 1 refers to MPEG-1, -2 and -2.5 layer 1, 2 or 3. The values 2 and 4 refer to the MPEG-AAC audio encoding schemes.
		framed	boolean	0 or 1	A true value indicates that each buffer contains exactly one frame. A false value indicates that frames and buffers do not necessarily match up.
		layer	integer	1, 2, or 3	The compression scheme layer used to compress the data *(only if mpegversion=1)*.
		bitrate	integer	greater than 0	The bitrate, in bits per second. For VBR (variable bitrate) MPEG data, this is the average bitrate.
audio/x-qdm2	Data encoded by the QDM version 2 codec.				There are currently no specific properties defined or needed for this type.
audio/x-pn-realaudio	Realmedia Audio data.	raversion	integer	1 or 2	The version of the Real Audio codec used to encode the stream. 1 stands for a 14k4 stream, 2 stands for a 28k8 stream.
audio/x-speex	Data encoded by the Speex audio codec				There are currently no specific properties defined or needed for this type.
audio/x-vorbis	Vorbis audio data				There are currently no specific properties defined or needed for this type.

Mime Type	Description	Property	Property Type	Property Values	Property Description
audio/x-wma	Windows Media Audio	wmaversion	integer	1,2 or 3	The version of the WMA codec used to encode the stream.
audio/x-paris	Ensoniq PARIS audio				There are currently no specific properties defined or needed for this type.
audio/x-svx	Amiga IFF / SVX8 / SV16 audio				There are currently no specific properties defined or needed for this type.
audio/x-nist	Sphere NIST audio				There are currently no specific properties defined or needed for this type.
audio/x-voc	Sound Blaster VOC audio				There are currently no specific properties defined or needed for this type.
audio/x-ircam	Berkeley/IRCAM/CARL audio				There are currently no specific properties defined or needed for this type.
audio/x-w64	Sonic Foundry's 64 bit RIFF/WAV				There are currently no specific properties defined or needed for this type.

Table 12-2. Table of Video Types

Mime Type	Description	Property	Property Type	Property Values	Property Description
All video types.					
video/*	*All video types*	width	integer	greater than 0	The width of the video image

Mime Type	Description	Property	Property Type	Property Values	Property Description
		height	integer	greater than 0	The height of the video image
		framerate	fraction	greater or equal 0	The (average) framerate in frames per second. Note that this property does not guarantee in *any* way that it will actually come close to this value. If you need a fixed framerate, please use an element that provides that (such as "videodrop"). 0 means a variable framerate.
All raw video types.					
video/x-raw-yuv	YUV (or Y'Cb'Cr) video format.	format	fourcc	YUY2, YVYU, UYVY, Y41P, IYU2, Y42B, YV12, I420, Y41B, YUV9, YVU9, Y800	The layout of the video. See FourCC definition site[2] for references and definitions. YUY2, YVYU and UYVY are 4:2:2 packed-pixel, Y41P is 4:1:1 packed-pixel and IYU2 is 4:4:4 packed-pixel. Y42B is 4:2:2 planar, YV12 and I420 are 4:2:0 planar, Y41B is 4:1:1 planar and YUV9 and YVU9 are 4:1:0 planar. Y800 contains Y-samples only (black/white).
video/x-raw-rgb	Red-Green-Blue (RBG) video.	bpp	integer	greater than 0	The number of bits allocated per pixel. This is usually 16, 24 or 32.
		depth	integer	greater than 0	The number of bits used per pixel by the R/G/B components. This is usually 15, 16 or 24.
		endianness	integer	G_BIG_ENDIAN (4321) or G_LITTLE_ENDIAN (1234)	The order of bytes in a sample. The value G_LITTLE_ENDIAN (1234) means "little-endian" (byte-order is "least significant byte first"). The value G_BIG_ENDIAN (4321) means "big-endian" (byte order is "most significant byte first"). For 24/32bpp, this should always be big endian because the byte order can be given in both.

Mime Type	Description	Property	Property Type	Property Values	Property Description
		red_mask, green_mask and blue_mask	integer	any	The masks that cover all the bits used by each of the samples. The mask should be given in the endianness specified above. This means that for 24/32bpp, the masks might be opposite to host byte order (if you are working on little-endian computers).
All encoded video types.					
video/x-3ivx	3ivx video.				There are currently no specific properties defined or needed for this type.
video/x-divx	DivX video.	divxversion	integer	3, 4 or 5	Version of the DivX codec used to encode the stream.
video/x-dv	Digital Video.	systemstream	boolean	FALSE	Indicates that this stream is *not* a system container stream.
video/x-ffv	FFMpeg video.	ffvversion	integer	1	Version of the FFMpeg video codec used to encode the stream.
video/x-h263	H-263 video.	variant	string	itu, lead, microsoft, vdolive, vivo, xirlink	Vendor specific variant of the format. 'itu' is the standard.
		h263version	string	h263, h263p, h263pp	Enhanced versions of the h263 codec.
video/x-h264	H-264 video.	variant	string	itu, videosoft	Vendor specific variant of the format. 'itu' is the standard.
video/x-huffyuv	Huffyuv video.				There are currently no specific properties defined or needed for this type.
video/x-indeo	Indeo video.	indeoversion	integer	3	Version of the Indeo codec used to encode this stream.

Mime Type	Description	Property	Property Type	Property Values	Property Description
video/x-intel-h263	H-263 video.	variant	string	intel	Vendor specific variant of the format.
video/x-jpeg	Motion-JPEG video.				There are currently no specific properties defined or needed for this type. Note that video/x-jpeg only applies to Motion-JPEG pictures (YUY2 colourspace). RGB colourspace JPEG images are referred to as image/jpeg (JPEG image).
video/mpeg	MPEG video.	mpegversion	integer	1, 2 or 4	Version of the MPEG codec that this stream was encoded with. Note that we have different mimetypes for 3ivx, XviD, DivX and "standard" ISO MPEG-4. This is *not* a good thing and we're fully aware of this. However, we do not have a solution yet.
		systemstream	boolean	FALSE	Indicates that this stream is *not* a system container stream.
video/x-msmpeg	Microsoft MPEG-4 video deviations.	msmpegversion	integer	41, 42 or 43	Version of the MS-MPEG-4-like codec that was used to encode this version. A value of 41 refers to MS MPEG 4.1, 42 to 4.2 and 43 to version 4.3.
video/x-msvideocodec	Microsoft Video 1 (old-ish codec).	msvideoversion	integer	1	Version of the codec - always 1.
video/x-pn-realvideo	Realmedia video.	rmversion	integer	1, 2 or 3	Version of the Real Video codec that this stream was encoded with.
video/x-rle	RLE animation format.	layout	string	"microsoft" or "quicktime"	The RLE format inside the Microsoft AVI container has a different byte layout than the RLE format inside Apple's Quicktime container; this property keeps track of the layout.

Mime Type	Description	Property	Property Type	Property Values	Property Description
		depth	integer	1 to 64	Bitdepth of the used palette. This means that the palette that belongs to this format defines 2^depth colors.
		palette_data	GstBuffer		Buffer containing a color palette (in native-endian RGBA) used by this format. The buffer is of size 4*2^depth.
video/x-svq	Sorenson Video.	svqversion	integer	1 or 3	Version of the Sorensen codec that the stream was encoded with.
video/x-tarkin	Tarkin video.				There are currently no specific properties defined or needed for this type.
video/x-theora	Theora video.				There are currently no specific properties defined or needed for this type.
video/x-vp3	VP-3 video.				There are currently no specific properties defined or needed for this type. Note that we have different mimetypes for VP-3 and Theora, which is not necessarily a good idea. This could probably be improved.
video/x-wmv	Windows Media Video	wmvversion	integer	1,2 or 3	Version of the WMV codec that the stream was encoded with.
video/x-xvid	XviD video.				There are currently no specific properties defined or needed for this type.
All image types.					
image/gif	Graphics Interchange Format.				There are currently no specific properties defined or needed for this type.

Mime Type	Description	Property	Property Type	Property Values	Property Description
image/jpeg	Joint Picture Expert Group Image.				There are currently no specific properties defined or needed for this type. Note that image/jpeg only applies to RGB-colourspace JPEG images; YUY2-colourspace JPEG pictures are referred to as video/x-jpeg ("Motion JPEG").
image/png	Portable Network Graphics Image.				There are currently no specific properties defined or needed for this type.
image/tiff	Tagged Image File Format.				There are currently no specific properties defined or needed for this type.

Table 12-3. Table of Container Types

Mime Type	Description	Property	Property Type	Property Values	Property Description
video/x-ms-asf	Advanced Streaming Format (ASF).				There are currently no specific properties defined or needed for this type.
video/x-msvideo	AVI.				There are currently no specific properties defined or needed for this type.
video/x-dv	Digital Video.	systemstream	boolean	TRUE	Indicates that this is a container system stream rather than an elementary video stream.

Mime Type	Description	Property	Property Type	Property Values	Property Description
video/x-matroska	Matroska.				There are currently no specific properties defined or needed for this type.
video/mpeg	Motion Pictures Expert Group System Stream.	systemstream	boolean	TRUE	Indicates that this is a container system stream rather than an elementary video stream.
application/ogg	Ogg.	ogg			There are currently no specific properties defined or needed for this type.
video/quicktime	Quicktime.				There are currently no specific properties defined or needed for this type.
application/vnd.rn-realmedia	RealMedia.				There are currently no specific properties defined or needed for this type.
audio/x-wav	WAV.				There are currently no specific properties defined or needed for this type.

Table 12-4. Table of Subtitle Types

Mime Type	Description	Property	Property Type	Property Values	Property Description
					None defined yet.

Table 12-5. Table of Other Types

Mime Type	Description	Property	Property Type	Property Values	Property Description
					None defined yet.

Notes

1. http://www.iana.org/assignments/media-types

Chapter 13. Request and Sometimes pads

Until now, we've only dealt with pads that are always available. However, there's also pads that are only being created in some cases, or only if the application requests the pad. The first is called a *sometimes*; the second is called a *request* pad. The availability of a pad (always, sometimes or request) can be seen in a pad's template. This chapter will discuss when each of the two is useful, how they are created and when they should be disposed.

Sometimes pads

A "sometimes" pad is a pad that is created under certain conditions, but not in all cases. This mostly depends on stream content: demuxers will generally parse the stream header, decide what elementary (video, audio, subtitle, etc.) streams are embedded inside the system stream, and will then create a sometimes pad for each of those elementary streams. At its own choice, it can also create more than one instance of each of those per element instance. The only limitation is that each newly created pad should have a unique name. Sometimes pads are disposed when the stream data is disposed, too (i.e. when going from PAUSED to the READY state). You should *not* dispose the pad on EOS, because someone might re-activate the pipeline and seek back to before the end-of-stream point. The stream should still stay valid after EOS, at least until the stream data is disposed. In any case, the element is always the owner of such a pad.

The example code below will parse a text file, where the first line is a number (n). The next lines all start with a number (0 to n-1), which is the number of the source pad over which the data should be sent.

```
3
0: foo
1: bar
0: boo
2: bye
```

The code to parse this file and create the dynamic "sometimes" pads, looks like this:

```
typedef struct _GstMyFilter {
[..]
  gboolean firstrun;
  GList *srcpadlist;
} GstMyFilter;

static void
gst_my_filter_base_init (GstMyFilterClass *klass)
{
  GstElementClass *element_class = GST_ELEMENT_CLASS (klass);
  static GstStaticPadTemplate src_factory =
  GST_STATIC_PAD_TEMPLATE (
    "src_%02d",
    GST_PAD_SRC,
    GST_PAD_SOMETIMES,
```

```
    GST_STATIC_CAPS ("ANY")
  );
[..]
  gst_element_class_add_pad_template (element_class,
 gst_static_pad_template_get (&src_factory));
[..]
}

static void
gst_my_filter_init (GstMyFilter *filter)
{
[..]
  filter->firstrun = TRUE;
  filter->srcpadlist = NULL;
}

/*
 * Get one line of data - without newline.
 */

static GstBuffer *
gst_my_filter_getline (GstMyFilter *filter)
{
  guint8 *data;
  gint n, num;

  /* max. line length is 512 characters - for safety */
  for (n = 0; n < 512; n++) {
    num = gst_bytestream_peek_bytes (filter->bs, &data, n + 1);
    if (num != n + 1)
      return NULL;

    /* newline? */
    if (data[n] == '\n') {
      GstBuffer *buf = gst_buffer_new_and_alloc (n + 1);

      gst_bytestream_peek_bytes (filter->bs, &data, n);
      memcpy (GST_BUFFER_DATA (buf), data, n);
      GST_BUFFER_DATA (buf)[n] = '\0';
      gst_bytestream_flush_fast (filter->bs, n + 1);

      return buf;
    }
  }
}

static void
gst_my_filter_loopfunc (GstElement *element)
{
  GstMyFilter *filter = GST_MY_FILTER (element);
  GstBuffer *buf;
  GstPad *pad;
  gint num, n;

  /* parse header */
  if (filter->firstrun) {
```

```
  GstElementClass *klass;
  GstPadTemplate *templ;
  gchar *padname;

  if (!(buf = gst_my_filter_getline (filter))) {
    gst_element_error (element, STREAM, READ, (NULL),
  ("Stream contains no header"));
    return;
  }
  num = atoi (GST_BUFFER_DATA (buf));
  gst_buffer_unref (buf);

  /* for each of the streams, create a pad */
  klass = GST_ELEMENT_GET_CLASS (filter);
  templ = gst_element_class_get_pad_template (klass, "src_%02d");
  for (n = 0; n < num; n++) {
    padname = g_strdup_printf ("src_%02d", n);
    pad = gst_pad_new_from_template (templ, padname);
    g_free (padname);

    /* here, you would set _getcaps () and _link () functions */

    gst_element_add_pad (element, pad);
    filter->srcpadlist = g_list_append (filter->srcpadlist, pad);
  }
}

/* and now, simply parse each line and push over */
if (!(buf = gst_my_filter_getline (filter))) {
  GstEvent *event = gst_event_new (GST_EVENT_EOS);
  GList *padlist;

  for (padlist = srcpadlist;
       padlist != NULL; padlist = g_list_next (padlist)) {
    pad = GST_PAD (padlist->data);
    gst_event_ref (event);
    gst_pad_push (pad, GST_DATA (event));
  }
  gst_event_unref (event);
  gst_element_set_eos (element);

  return;
}

/* parse stream number and go beyond the ':' in the data */
num = atoi (GST_BUFFER_DATA (buf));
if (num >= 0 && num < g_list_length (filter->srcpadlist)) {
  pad = GST_PAD (g_list_nth_data (filter->srcpadlist, num);

  /* magic buffer parsing foo */
  for (n = 0; GST_BUFFER_DATA (buf)[n] != ':' &&
              GST_BUFFER_DATA (buf)[n] != '\0'; n++) ;
  if (GST_BUFFER_DATA (buf)[n] != '\0') {
    GstBuffer *sub;

    /* create subbuffer that starts right past the space. The reason
```

```
       * that we don't just forward the data pointer is because the
       * pointer is no longer the start of an allocated block of memory,
       * but just a pointer to a position somewhere in the middle of it.
       * That cannot be freed upon disposal, so we'd either crash or have
       * a memleak. Creating a subbuffer is a simple way to solve that. */
      sub = gst_buffer_create_sub (buf, n + 1, GST_BUFFER_SIZE (buf) - n - 1);
      gst_pad_push (pad, GST_DATA (sub));
    }
  }
  gst_buffer_unref (buf);
}
```

Note that we use a lot of checks everywhere to make sure that the content in the file is valid. This has two purposes: first, the file could be erronous, in which case we prevent a crash. The second and most important reason is that - in extreme cases - the file could be used maliciously to cause undefined behaviour in the plugin, which might lead to security issues. *Always* assume that the file could be used to do bad things.

Request pads

"Request" pads are similar to sometimes pads, except that request are created on demand of something outside of the element rather than something inside the element. This concept is often used in muxers, where - for each elementary stream that is to be placed in the output system stream - one sink pad will be requested. It can also be used in elements with a variable number of input or outputs pads, such as the `tee` (multi-output), `switch` or `aggregator` (both multi-input) elements. At the time of writing this, it is unclear to me who is responsible for cleaning up the created pad and how or when that should be done. Below is a simple example of an aggregator based on request pads.

```
static GstPad * gst_my_filter_request_new_pad (GstElement     *element,
       GstPadTemplate *templ,
       const gchar    *name);

static void
gst_my_filter_base_init (GstMyFilterClass *klass)
{
  GstElementClass *element_class = GST_ELEMENT_CLASS (klass);
  static GstStaticPadTemplate sink_factory =
  GST_STATIC_PAD_TEMPLATE (
    "sink_%d",
    GST_PAD_SINK,
    GST_PAD_REQUEST,
    GST_STATIC_CAPS ("ANY")
  );
[..]
  gst_element_class_add_pad_template (klass,
 gst_static_pad_template_get (&sink_factory));
}
```

```
static void
gst_my_filter_class_init (GstMyFilterClass *klass)
{
  GstElementClass *element_class = GST_ELEMENT_CLASS (klass);
[..]
  element_class->request_new_pad = gst_my_filter_request_new_pad;
}

static GstPad *
gst_my_filter_request_new_pad (GstElement     *element,
          GstPadTemplate *templ,
          const gchar    *name)
{
  GstPad *pad;
  GstMyFilterInputContext *context;

  context = g_new0 (GstMyFilterInputContext, 1);
  pad = gst_pad_new_from_template (templ, name);
  gst_pad_set_element_private (pad, context);

  /* normally, you would set _link () and _getcaps () functions here */

  gst_element_add_pad (element, pad);

  return pad;
}
```

Chapter 14. Clocking

When playing complex media, each sound and video sample must be played in a specific order at a specific time. For this purpose, GStreamer provides a synchronization mechanism.

Types of time

There are two kinds of time in GStreamer. *Clock time* is an absolute time. By contrast, *element time* is the relative time, usually to the start of the current media stream. The element time represents the time that should have a media sample that is being processed by the element at this time. The element time is calculated by adding an offset to the clock time.

Clocks

GStreamer can use different clocks. Though the system time can be used as a clock, soundcards and other devices provides a better time source. For this reason some elements provide a clock. The method `get_clock` is implemented in elements that provide one.

As clocks return an absolute measure of time, they are not usually used directly. Instead, a reference to a clock is stored in any element that needs it, and it is used internaly by GStreamer to calculate the element time.

Flow of data between elements and time

Now we will see how time information travels the pipeline in different states.

The pipeline starts playing. The source element typically knows the time of each sample. [1] First, the source element sends a discontinous event. This event carries information about the current relative time of the next sample. This relative time is arbitrary, but it must be consistent with the timestamp that will be placed in buffers. It is expected to be the relative time to the start of the media stream, or whatever makes sense in the case of each media. When receiving it, the other elements adjust their offset of the element time so that this time matches the time written in the event.

Then the source element sends media samples in buffers. This element places a timestamp in each buffer saying when the sample should be played. When the buffer reachs the sink pad of the last element, this element compares the current element time with the timestamp of the buffer. If the timestamp is higher or equal it plays the buffer, otherwise it waits until the time to place the buffer arrives with `gst_element_wait()`.

If the stream is seeked, the next samples sent will have a timestamp that is not adjusted with the element time. Therefore, the source element must send a discontinous event.

Obligations of each element.

Let us clarify the contract between GStreamer and each element in the pipeline.

Source elements

Source elements (or parsers of formats that provide notion of time, such as MPEG, as explained above) must place a timestamp in each buffer that they deliver. The origin of the time used is arbitrary, but it must match the time delivered in the discontinous event (see below). However, it is expected that the origin is the origin of the media stream.

In order to initialize the element time of the rest of the pipeline, a source element must send a discontinous event before starting to play. In addition, after seeking, a discontinious event must be sent, because the timestamp of the next element does not match the element time of the rest of the pipeline.

Sink elements

If the element is intended to emit samples at a specific time (real time playing), the element should require a clock, and thus implement the method `set_clock`.

In addition, before playing each sample, if the current element time is less than the timestamp in the sample, it wait until the current time arrives should call `gst_element_wait()` [2]

Notes

1. Sometimes it is a parser element the one that knows the time, for instance if a pipeline contains a filesrc element connected to a MPEG decoder element, the former is the one that knows the time of each sample, because the knowledge of when to play each sample is embedded in the MPEG format. In this case this element will be regarded as the source element for this discussion.
2. With some schedulers, `gst_element_wait()` blocks the pipeline. For instance, if there is one audio sink element and one video sink element, while the audio element is waiting for a sample the video element cannot play other sample. This behaviour is under discussion, and might change in a future release.

Chapter 15. Supporting Dynamic Parameters

Sometimes object properties are not powerful enough to control the parameters that affect the behaviour of your element. When this is the case you can mark these parameters as beeing Controllable. Aware appliations can use the controller subsystem to dynamically adjust the property values over time.

Getting Started

The controller subsystem is contained within the `gstcontroller` library. You need to include the header in your element's source file:

```
...
#include <gst/gst.h>
#include <gst/controller/gstcontroller.h>
...
```

Even though the `gstcontroller` library may be linked into the host application, you should make sure it is initialized in your `plugin_init` function:

```
static gboolean
plugin_init (GstPlugin *plugin)
{
  ...
  /* initialize library */
  gst_controller_init (NULL, NULL);
  ...
}
```

It makes not sense for all GObject parameter to be real-time controlled. Therefore the next step is to mark controllable parameters. This is done by using the special flag `GST_PARAM_CONTROLLABLE`. when setting up GObject params in the `_class_init` method.

```
g_object_class_install_property (gobject_class, PROP_FREQ,
    g_param_spec_double ("freq", "Frequency", "Frequency of test signal",
        0.0, 20000.0, 440.0,
        G_PARAM_READWRITE | GST_PARAM_CONTROLLABLE | G_PARAM_STATIC_STRINGS));
```

The Data Processing Loop

In the last section we learned how to mark GObject params as controllable. Application developers can then queue parameter changes for these parameters. The approach the controller subsystem takes is to make plugins responsible for pulling the changes in. This requires just one action:

```
gst_object_sync_values(element,timestamp);
```

This call makes all parameter-changes for the given timestamp active by adjusting the GObject properties of the element. Its up to the element to determine the synchronisation rate.

The Data Processing Loop for Video Elements

For video processing elements it is the best to synchonise for every frame. That means one would add the `gst_object_sync_values()` call described in the previous section to the data processing function of the element.

The Data Processing Loop for Audio Elements

For audio processing elements the case is not as easy as for video processing elements. The problem here is that audio has a much higher rate. For PAL video one will e.g. process 25 full frames per second, but for standard audio it will be 44100 samples. It is rarely useful to synchronise controllable parameters that often. The easiest solution is also to have just one synchronisation call per buffer processing. This makes the control-rate dependend on the buffer size.

Elements that need a specific control-rate need to break their data processing loop to synchronise every n-samples.

Chapter 16. MIDI

WRITEME

Chapter 17. Interfaces

Previously, in the chapter Adding Arguments, we have introduced the concept of GObject properties of controlling an element's behaviour. This is very powerful, but it has two big disadvantages: first of all, it is too generic, and second, it isn't dynamic.

The first disadvantage is related to the customizability of the end-user interface that will be built to control the element. Some properties are more important than others. Some integer properties are better shown in a spin-button widget, whereas others would be better represented by a slider widget. Such things are not possible because the UI has no actual meaning in the application. A UI widget that represents a bitrate property is the same as a UI widget that represents the size of a video, as long as both are of the same `GParamSpec` type. Another problem, is that things like parameter grouping, function grouping, or parameter coupling are not really possible.

The second problem with parameters are that they are not dynamic. In many cases, the allowed values for a property are not fixed, but depend on things that can only be detected at runtime. The names of inputs for a TV card in a video4linux source element, for example, can only be retrieved from the kernel driver when we've opened the device; this only happens when the element goes into the READY state. This means that we cannot create an enum property type to show this to the user.

The solution to those problems is to create very specialized types of controls for certain often-used controls. We use the concept of interfaces to achieve this. The basis of this all is the glib `GTypeInterface` type. For each case where we think it's useful, we've created interfaces which can be implemented by elements at their own will. We've also created a small extension to `GTypeInterface` (which is static itself, too) which allows us to query for interface availability based on runtime properties. This extension is called `GstImplementsInterface`[1].

One important note: interfaces do *not* replace properties. Rather, interfaces should be built *next to* properties. There are two important reasons for this. First of all, properties can be saved in XML files. Second, properties can be specified on the command-line (`gst-launch`).

How to Implement Interfaces

Implementing interfaces is intiated in the `_get_type ()` of your element. You can register one or more interfaces after having registered the type itself. Some interfaces have dependencies on other interfaces or can only be registered by certain types of elements. You will be notified of doing that wrongly when using the element: it will quit with failed assertions, which will explain what went wrong. In the case of GStreamer, the only dependency that *some* interfaces have is `GstImplementsInterface`[2]. Per interface, we will indicate clearly when it depends on this extension. If it does, you need to register support for *that* interface before registering support for the interface that you're wanting to support. The example below explains how to add support for a simple interface with no further dependencies. For a small explanation on `GstImplementsInterface`[3], see the next section about the mixer interface: Mixer Interface.

```
static void gst_my_filter_some_interface_init (GstSomeInterface *iface);

GType
```

```
gst_my_filter_get_type (void)
{
  static GType my_filter_type = 0;

  if (!my_filter_type) {
    static const GTypeInfo my_filter_info = {
      sizeof (GstMyFilterClass),
      (GBaseInitFunc) gst_my_filter_base_init,
      NULL,
      (GClassInitFunc) gst_my_filter_class_init,
      NULL,
      NULL,
      sizeof (GstMyFilter),
      0,
      (GInstanceInitFunc) gst_my_filter_init
    };
    static const GInterfaceInfo some_interface_info = {
      (GInterfaceInitFunc) gst_my_filter_some_interface_init,
      NULL,
      NULL
    };

    my_filter_type =
 g_type_register_static (GST_TYPE_MY_FILTER,
    "GstMyFilter",
    &my_filter_info, 0);
    g_type_add_interface_static (my_filter_type,
     GST_TYPE_SOME_INTERFACE,
                                 &some_interface_info);
  }

  return my_filter_type;
}

static void
gst_my_filter_some_interface_init (GstSomeInterface *iface)
{
  /* here, you would set virtual function pointers in the interface */
}
```

URI interface

WRITEME

Mixer Interface

The goal of the mixer interface is to provide a simple yet powerful API to applications for audio hardware mixer/volume control. Most soundcards have hardware mixers, where volume can be changed, they can be muted, inputs can be modified to mix their content into what will be read from the device by applications (in our case: audio source plugins). The mixer interface is the way to control those. The mixer

interface can also be used for volume control in software (e.g. the "volume" element). The end goal of this interface is to allow development of hardware volume control applications and for the control of audio volume and input/output settings.

The mixer interface requires the `GstImplementsInterface`[4] interface to be implemented by the element. The example below will feature both, so it serves as an example for the `GstImplementsInterface`[5], too. In this interface, it is required to set a function pointer for the `supported ()` function. If you don't, this function will always return FALSE (default implementation) and the mixer interface implementation will not work. For the mixer interface, the only required function is `list_tracks ()`. All other function pointers in the mixer interface are optional, although it is strongly recommended to set function pointers for at least the `get_volume ()` and `set_volume ()` functions. The API reference for this interface documents the goal of each function, so we will limit ourselves to the implementation here.

The following example shows a mixer implementation for a software N-to-1 element. It does not show the actual process of stream mixing, that is far too complicated for this guide.

```
#include <gst/mixer/mixer.h>

typedef struct _GstMyFilter {
[..]
  gint volume;
  GList *tracks;
} GstMyFilter;

static void gst_my_filter_implements_interface_init (GstImplementsInterfaceClass *if
static void gst_my_filter_mixer_interface_init (GstMixerClass *iface);

GType
gst_my_filter_get_type (void)
{
[..]
    static const GInterfaceInfo implements_interface_info = {
      (GInterfaceInitFunc) gst_my_filter_implements_interface_init,
      NULL,
      NULL
    };
    static const GInterfaceInfo mixer_interface_info = {
      (GInterfaceInitFunc) gst_my_filter_mixer_interface_init,
      NULL,
      NULL
    };
[..]
    g_type_add_interface_static (my_filter_type,
     GST_TYPE_IMPLEMENTS_INTERFACE,
     &implements_interface_info);
    g_type_add_interface_static (my_filter_type,
     GST_TYPE_MIXER,
     &mixer_interface_info);
[..]
}

static void
gst_my_filter_init (GstMyFilter *filter)
```

```
{
  GstMixerTrack *track = NULL;
[..]
  filter->volume = 100;
  filter->tracks = NULL;
  track = g_object_new (GST_TYPE_MIXER_TRACK, NULL);
  track->label = g_strdup ("MyTrack");
  track->num_channels = 1;
  track->min_volume = 0;
  track->max_volume = 100;
  track->flags = GST_MIXER_TRACK_SOFTWARE;
  filter->tracks = g_list_append (filter->tracks, track);
}

static gboolean
gst_my_filter_interface_supported (GstImplementsInterface *iface,
       GType                   iface_type)
{
  g_return_val_if_fail (iface_type == GST_TYPE_MIXER, FALSE);

  /* for the sake of this example, we'll always support it. However, normally,
   * you would check whether the device you've opened supports mixers. */
  return TRUE;
}

static void
gst_my_filter_implements_interface_init (GstImplementsInterfaceClass *iface)
{
  iface->supported = gst_my_filter_interface_supported;
}

/*
 * This function returns the list of support tracks (inputs, outputs)
 * on this element instance. Elements usually build this list during
 * _init () or when going from NULL to READY.
 */

static const GList *
gst_my_filter_mixer_list_tracks (GstMixer *mixer)
{
  GstMyFilter *filter = GST_MY_FILTER (mixer);

  return filter->tracks;
}

/*
 * Set volume. volumes is an array of size track->num_channels, and
 * each value in the array gives the wanted volume for one channel
 * on the track.
 */

static void
gst_my_filter_mixer_set_volume (GstMixer      *mixer,
    GstMixerTrack *track,
    gint          *volumes)
{
```

```
  GstMyFilter *filter = GST_MY_FILTER (mixer);

  filter->volume = volumes[0];

  g_print ("Volume set to %d\n", filter->volume);
}

static void
gst_my_filter_mixer_get_volume (GstMixer      *mixer,
    GstMixerTrack *track,
    gint          *volumes)
{
  GstMyFilter *filter = GST_MY_FILTER (mixer);

  volumes[0] = filter->volume;
}

static void
gst_my_filter_mixer_interface_init (GstMixerClass *iface)
{
  /* the mixer interface requires a definition of the mixer type:
   * hardware or software? */
  GST_MIXER_TYPE (iface) = GST_MIXER_SOFTWARE;

  /* virtual function pointers */
  iface->list_tracks = gst_my_filter_mixer_list_tracks;
  iface->set_volume  = gst_my_filter_mixer_set_volume;
  iface->get_volume  = gst_my_filter_mixer_get_volume;
}
```

The mixer interface is very audio-centric. However, with the software flag set, the mixer can be used to mix any kind of stream in a N-to-1 element to join (not aggregate!) streams together into one output stream. Conceptually, that's called mixing too. You can always use the element factory's "category" to indicate type of your element. In a software element that mixes random streams, you would not be required to implement the `_get_volume ()` or `_set_volume ()` functions. Rather, you would only implement the `_set_record ()` to enable or disable tracks in the output stream. to make sure that a mixer-implementing element is of a certain type, check the element factory's category.

Tuner Interface

As opposed to the mixer interface, that's used to join together N streams into one output stream by mixing all streams together, the tuner interface is used in N-to-1 elements too, but instead of mixing the input streams, it will select one stream and push the data of that stream to the output stream. It will discard the data of all other streams. There is a flag that indicates whether this is a software-tuner (in which case it is a pure software implementation, with N sink pads and 1 source pad) or a hardware-tuner, in which case it only has one source pad, and the whole stream selection process is done in hardware. The software case can be used in elements such as *switch*. The hardware case can be used in elements with channel selection, such as video source elements (v4lsrc, v4l2src, etc.). If you need a specific element type, use

the element factory's "category" to make sure that the element is of the type that you need. Note that the interface itself is highly analog-video-centric.

This interface requires the `GstImplemensInterface`[6] interface to work correctly.

The following example shows how to implement the tuner interface in an element. It does not show the actual process of stream selection, that is irrelevant for this section.

```
#include <gst/tuner/tuner.h>

typedef struct _GstMyFilter {
[..]
  gint active_input;
  GList *channels;
} GstMyFilter;

static void gst_my_filter_implements_interface_init (GstImplementsInterfaceClass *if
static void gst_my_filter_tuner_interface_init (GstTunerClass *iface);

GType
gst_my_filter_get_type (void)
{
[..]
    static const GInterfaceInfo implements_interface_info = {
      (GInterfaceInitFunc) gst_my_filter_implements_interface_init,
      NULL,
      NULL
    };
    static const GInterfaceInfo tuner_interface_info = {
      (GInterfaceInitFunc) gst_my_filter_tuner_interface_init,
      NULL,
      NULL
    };
[..]
    g_type_add_interface_static (my_filter_type,
     GST_TYPE_IMPLEMENTS_INTERFACE,
     &implements_interface_info);
    g_type_add_interface_static (my_filter_type,
     GST_TYPE_TUNER,
     &tunerr_interface_info);
[..]
}

static void
gst_my_filter_init (GstMyFilter *filter)
{
  GstTunerChannel *channel = NULL;
[..]
  filter->active_input = 0;
  filter->channels = NULL;
  channel = g_object_new (GST_TYPE_TUNER_CHANNEL, NULL);
  channel->label = g_strdup ("MyChannel");
  channel->flags = GST_TUNER_CHANNEL_INPUT;
  filter->channels = g_list_append (filter->channels, channel);
}

static gboolean
```

```
gst_my_filter_interface_supported (GstImplementsInterface *iface,
       GType                   iface_type)
{
  g_return_val_if_fail (iface_type == GST_TYPE_TUNER, FALSE);

  /* for the sake of this example, we'll always support it. However, normally,
   * you would check whether the device you've opened supports tuning. */
  return TRUE;
}

static void
gst_my_filter_implements_interface_init (GstImplementsInterfaceClass *iface)
{
  iface->supported = gst_my_filter_interface_supported;
}

static const GList *
gst_my_filter_tuner_list_channels (GstTuner *tuner)
{
  GstMyFilter *filter = GST_MY_FILTER (tuner);

  return filter->channels;
}

static GstTunerChannel *
gst_my_filter_tuner_get_channel (GstTuner *tuner)
{
  GstMyFilter *filter = GST_MY_FILTER (tuner);

  return g_list_nth_data (filter->channels,
     filter->active_input);
}

static void
gst_my_filter_tuner_set_channel (GstTuner        *tuner,
     GstTunerChannel *channel)
{
  GstMyFilter *filter = GST_MY_FILTER (tuner);

  filter->active_input = g_list_index (filter->channels, channel);
  g_assert (filter->active_input >= 0);
}

static void
gst_my_filter_tuner_interface_init (GstTunerClass *iface)
{
  iface->list_channels = gst_my_filter_tuner_list_channels;
  iface->get_channel   = gst_my_filter_tuner_get_channel;
  iface->set_channel   = gst_my_filter_tuner_set_channel;
}
```

As said, the tuner interface is very analog video-centric. It features functions for selecting an input or output, and on inputs, it features selection of a tuning frequency if the channel supports frequency-tuning on that input. Likewise, it allows signal-strength-acquiring if the input supports that. Frequency tuning can be used for radio

or cable-TV tuning. Signal-strength is an indication of the signal and can be used for visual feedback to the user or for autodetection. Next to that, it also features norm selection, which is only useful for analog video elements.

Color Balance Interface

WRITEME

Property Probe Interface

Property probing is a generic solution to the problem that properties' value lists in an enumeration are static. We've shown enumerations in Adding Arguments. Property probing tries to accomplish a goal similar to enumeration lists: to have a limited, explicit list of allowed values for a property. There are two differences between enumeration lists and probing. Firstly, enumerations only allow strings as values; property probing works for any value type. Secondly, the contents of a probed list of allowed values may change during the life of an element. The contents of an enumeration list are static. Currently, property probing is being used for detection of devices (e.g. for OSS elements, Video4linux elements, etc.). It could - in theory - be used for any property, though.

Property probing stores the list of allowed (or recommended) values in a `GValueArray` and returns that to the user. NULL is a valid return value, too. The process of property probing is separated over two virtual functions: one for probing the property to create a `GValueArray`, and one to retrieve the current `GValueArray`. Those two are separated because probing might take a long time (several seconds). Also, this simpliies interface implementation in elements. For the application, there are functions that wrap those two. For more information on this, have a look at the API reference for the `GstPropertyProbe` interface.

Below is a example of property probing for the audio filter element; it will probe for allowed values for the "silent" property. Indeed, this value is a gboolean so it doesn't make much sense. Then again, it's only an example.

```
#include <gst/propertyprobe/propertyprobe.h>

static void gst_my_filter_probe_interface_init (GstPropertyProbeInterface *iface);

GType
gst_my_filter_get_type (void)
{
[..]
    static const GInterfaceInfo probe_interface_info = {
      (GInterfaceInitFunc) gst_my_filter_probe_interface_init,
      NULL,
      NULL
    };
[..]
    g_type_add_interface_static (my_filter_type,
     GST_TYPE_PROPERTY_PROBE,
     &probe_interface_info);
```

```
[..]
}

static const GList *
gst_my_filter_probe_get_properties (GstPropertyProbe *probe)
{
  GObjectClass *klass = G_OBJECT_GET_CLASS (probe);
  static GList *props = NULL;

  if (!props) {
    GParamSpec *pspec;

    pspec = g_object_class_find_property (klass, "silent");
    props = g_list_append (props, pspec);
  }

  return props;
}

static gboolean
gst_my_filter_probe_needs_probe (GstPropertyProbe *probe,
     guint             prop_id,
     const GParamSpec *pspec)
{
  gboolean res = FALSE;

  switch (prop_id) {
    case ARG_SILENT:
      res = FALSE;
      break;
    default:
      G_OBJECT_WARN_INVALID_PROPERTY_ID (probe, prop_id, pspec);
      break;
  }

  return res;
}

static void
gst_my_filter_probe_probe_property (GstPropertyProbe *probe,
        guint             prop_id,
        const GParamSpec *pspec)
{
  switch (prop_id) {
    case ARG_SILENT:
      /* don't need to do much here... */
      break;
    default:
      G_OBJECT_WARN_INVALID_PROPERTY_ID (probe, prop_id, pspec);
      break;
  }
}

static GValueArray *
gst_my_filter_get_silent_values (GstMyFilter *filter)
{
```

```
  GValueArray *array = g_value_array_new (2);
  GValue value = { 0 };

  g_value_init (&value, G_TYPE_BOOLEAN);

  /* add TRUE */
  g_value_set_boolean (&value, TRUE);
  g_value_array_append (array, &value);

  /* add FALSE */
  g_value_set_boolean (&value, FALSE);
  g_value_array_append (array, &value);

  g_value_unset (&value);

  return array;
}

static GValueArray *
gst_my_filter_probe_get_values (GstPropertyProbe *probe,
    guint             prop_id,
    const GParamSpec *pspec)
{
  GstMyFilter *filter = GST_MY_FILTER (probe);
  GValueArray *array = NULL;

  switch (prop_id) {
    case ARG_SILENT:
      array = gst_my_filter_get_silent_values (filter);
      break;
    default:
      G_OBJECT_WARN_INVALID_PROPERTY_ID (probe, prop_id, pspec);
      break;
  }

  return array;
}

static void
gst_my_filter_probe_interface_init (GstPropertyProbeInterface *iface)
{
  iface->get_properties = gst_my_filter_probe_get_properties;
  iface->needs_probe    = gst_my_filter_probe_needs_probe;
  iface->probe_property = gst_my_filter_probe_probe_property;
  iface->get_values     = gst_my_filter_probe_get_values;
}
```

You don't need to support any functions for getting or setting values. All that is handled via the standard `GObject` `_set_property ()` and `_get_property ()` functions.

X Overlay Interface

An X Overlay is basically a video output in a XFree86 drawable. Elements implementing this interface will draw video in a X11 window. Through this interface, applications will be proposed 2 different modes to work with a plugin implemeting it. The first mode is a passive mode where the plugin owns, creates and destroys the X11 window. The second mode is an active mode where the application handles the X11 window creation and then tell the plugin where it should output video. Let's get a bit deeper in those modes...

A plugin drawing video output in a X11 window will need to have that window at one stage or another. Passive mode simply means that no window has been given to the plugin before that stage, so the plugin created the window by itself. In that case the plugin is responsible of destroying that window when it's not needed anymore and it has to tell the applications that a window has been created so that the application can use it. This is done using the `have_xwindow_id` signal that can be emitted from the plugin with the `gst_x_overlay_got_xwindow_id` method.

As you probably guessed already active mode just means sending a X11 window to the plugin so that video output goes there. This is done using the `gst_x_overlay_set_xwindow_id` method.

It is possible to switch from one mode to another at any moment, so the plugin implementing this interface has to handle all cases. There are only 2 methods that plugins writers have to implement and they most probably look like that :

```
static void
gst_my_filter_set_xwindow_id (GstXOverlay *overlay, XID xwindow_id)
{
  GstMyFilter *my_filter = GST_MY_FILTER (overlay);

  if (my_filter->window)
    gst_my_filter_destroy_window (my_filter->window);

  my_filter->window = xwindow_id;
}

static void
gst_my_filter_get_desired_size (GstXOverlay *overlay,
                                guint *width, guint *height)
{
  GstMyFilter *my_filter = GST_MY_FILTER (overlay);

  *width = my_filter->width;
  *height = my_filter->height;
}

static void
gst_my_filter_xoverlay_init (GstXOverlayClass *iface)
{
  iface->set_xwindow_id = gst_my_filter_set_xwindow_id;
  iface->get_desired_size = gst_my_filter_get_desired_size;
}
```

You will also need to use the interface methods to fire signals when needed such as in the pad link function where you will know the video geometry and maybe create the window.

```
static MyFilterWindow *
gst_my_filter_window_create (GstMyFilter *my_filter, gint width, gint height)
{
  MyFilterWindow *window = g_new (MyFilterWindow, 1);
  ...
  gst_x_overlay_got_xwindow_id (GST_X_OVERLAY (my_filter), window->win);
}

static GstPadLinkReturn
gst_my_filter_sink_link (GstPad *pad, const GstCaps *caps)
{
  GstMyFilter *my_filter = GST_MY_FILTER (overlay);
  gint width, height;
  gboolean ret;
  ...
  ret = gst_structure_get_int (structure, "width", &width);
  ret &= gst_structure_get_int (structure, "height", &height);
  if (!ret) return GST_PAD_LINK_REFUSED;

  if (!my_filter->window)
    my_filter->window = gst_my_filter_create_window (my_filter, width, height);

  gst_x_overlay_got_desired_size (GST_X_OVERLAY (my_filter),
                                  width, height);
  ...
}
```

Navigation Interface

WRITEME

Notes

1. ../../gstreamer/html/GstImplementsInterface.html
2. ../../gstreamer/html/GstImplementsInterface.html
3. ../../gstreamer/html/GstImplementsInterface.html
4. ../../gstreamer/html/GstImplementsInterface.html
5. ../../gstreamer/html/GstImplementsInterface.html
6. ../../gstreamer/html/GstImplementsInterface.html

Chapter 18. Tagging (Metadata and Streaminfo)

Overview

Tags are pieces of information stored in a stream that are not the content itself, but they rather *describe* the content. Most media container formats support tagging in one way or another. Ogg uses VorbisComment for this, MP3 uses ID3, AVI and WAV use RIFF's INFO list chunk, etc. GStreamer provides a general way for elements to read tags from the stream and expose this to the user. The tags (at least the metadata) will be part of the stream inside the pipeline. The consequence of this is that transcoding of files from one format to another will automatically preserve tags, as long as the input and output format elements both support tagging.

Tags are separated in two categories in GStreamer, even though applications won't notice anything of this. The first are called *metadata*, the second are called *streaminfo*. Metadata are tags that describe the non-technical parts of stream content. They can be changed without needing to re-encode the stream completely. Examples are "author", "title" or "album". The container format might still need to be re-written for the tags to fit in, though. Streaminfo, on the other hand, are tags that describe the stream contents technically. To change them, the stream needs to be re-encoded. Examples are "codec" or "bitrate". Note that some container formats (like ID3) store various streaminfo tags as metadata in the file container, which means that they can be changed so that they don't match the content in the file anymore. Still, they are called metadata because *technically*, they can be changed without re-encoding the whole stream, even though that makes them invalid. Files with such metadata tags will have the same tag twice: once as metadata, once as streaminfo.

There is no special name for tag reading elements in `GStreamer`. There are specialised elements (e.g. id3demux) that do nothing besides tag reading, but any `GStreamer` element may extract tags while processing data, and most decoders, demuxers and parsers do.

A tag writer is called `TagSetter`[1]. An element supporting both can be used in a tag editor for quick tag changing (note: in-place tag editing is still poorly supported at the time of writing and usually requires tag extraction/stripping and remuxing of the stream with new tags).

Reading Tags from Streams

The basic object for tags is a `GstTagList` [2]. An element that is reading tags from a stream should create an empty taglist and fill this with individual tags. Empty tag lists can be created with `gst_tag_list_new ()`. Then, the element can fill the list using `gst_tag_list_add ()` or `gst_tag_list_add_values ()`. Note that elements often read metadata as strings, but the values in the taglist might not necessarily be strings - they need to be of the type the tag was registered as (the API documentation for each predefined tag should contain the type). Be sure to use functions like `gst_value_transform ()` to make sure that your data is of the right type. After data reading, the application can be notified of the new taglist by calling `gst_element_found_tags ()` or `gst_element_found_tags_for_pad ()` (if they only refer to a specific sub-stream). These functions will post a tag message

on the pipeline's GstBus for the application to pick up, but also send tag events downstream, either over all source pad or the pad specified.

The following example program will parse a file and parse the data as metadata/tags rather than as actual content-data. It will parse each line as "name:value", where name is the type of metadata (title, author, ...) and value is the metadata value. The `_getline ()` is the same as the one given in Sometimes pads.

```
static void
gst_my_filter_task_func (GstElement *element)
{
  GstMyFilter *filter = GST_MY_FILTER (element);
  GstBuffer *buf;
  GstTagList *taglist = gst_tag_list_new ();

  /* get each line and parse as metadata */
  while ((buf = gst_my_filter_getline (filter))) {
    gchar *line = GST_BUFFER_DATA (buf), *colon_pos, *type = NULL;a

    /* get the position of the ':' and go beyond it */
    if (!(colon_pos = strchr (line, ':')))
      goto next:

    /* get the string before that as type of metadata */
    type = g_strndup (line, colon_pos - line);

    /* content is one character beyond the ':' */
    colon_pos = &colon_pos[1];
    if (*colon_pos == '\0')
      goto next;

    /* get the metadata category, it's value type, store it in that
     * type and add it to the taglist. */
    if (gst_tag_exists (type)) {
      GValue from = { 0 }, to = { 0 };
      GType to_type;

      to_type = gst_tag_get_type (type);
      g_value_init (&from, G_TYPE_STRING);
      g_value_set_string (&from, colon_pos);
      g_value_init (&to, to_type);
      g_value_transform (&from, &to);
      g_value_unset (&from);
      gst_tag_list_add_values (taglist, GST_TAG_MERGE_APPEND,
          type, &to, NULL);
      g_value_unset (&to);
    }

next:
    g_free (type);
    gst_buffer_unref (buf);
  }

  /* signal metadata */
  gst_element_found_tags_for_pad (element, filter->srcpad, 0, taglist);
```

```
  gst_tag_list_free (taglist);

  /* send EOS */
  gst_pad_send_event (filter->srcpad, GST_DATA (gst_event_new (GST_EVENT_EOS)));
  gst_element_set_eos (element);
}
```

We currently assume the core to already *know* the mimetype (`gst_tag_exists ()`). You can add new tags to the list of known tags using `gst_tag_register ()`. If you think the tag will be useful in more cases than just your own element, it might be a good idea to add it to `gsttag.c` instead. That's up to you to decide. If you want to do it in your own element, it's easiest to register the tag in one of your class init functions, preferrably `_class_init ()`.

```
static void
gst_my_filter_class_init (GstMyFilterClass *klass)
{
[..]
  gst_tag_register ("my_tag_name", GST_TAG_FLAG_META,
      G_TYPE_STRING,
      _("my own tag"),
      _("a tag that is specific to my own element"),
      NULL);
[..]
}
```

Writing Tags to Streams

Tag writers are the opposite of tag readers. Tag writers only take metadata tags into account, since that's the only type of tags that have to be written into a stream. Tag writers can receive tags in three ways: internal, application and pipeline. Internal tags are tags read by the element itself, which means that the tag writer is - in that case - a tag reader, too. Application tags are tags provided to the element via the TagSetter interface (which is just a layer). Pipeline tags are tags provided to the element from within the pipeline. The element receives such tags via the GST_EVENT_TAG event, which means that tags writers should automatically be event aware. The tag writer is responsible for combining all these three into one list and writing them to the output stream.

The example below will receive tags from both application and pipeline, combine them and write them to the output stream. It implements the tag setter so applications can set tags, and retrieves pipeline tags from incoming events.

```
GType
gst_my_filter_get_type (void)
{
[..]
```

```
    static const GInterfaceInfo tag_setter_info = {
      NULL,
      NULL,
      NULL
    };
[..]
    g_type_add_interface_static (my_filter_type,
     GST_TYPE_TAG_SETTER,
     &tag_setter_info);
[..]
}

static void
gst_my_filter_init (GstMyFilter *filter)
{
  GST_FLAG_SET (filter, GST_ELEMENT_EVENT_AWARE);
[..]
}

/*
 * Write one tag.
 */

static void
gst_my_filter_write_tag (const GstTagList *taglist,
    const gchar      *tagname,
    gpointer          data)
{
  GstMyFilter *filter = GST_MY_FILTER (data);
  GstBuffer *buffer;
  guint num_values = gst_tag_list_get_tag_size (list, tag_name), n;
  const GValue *from;
  GValue to = { 0 };

  g_value_init (&to, G_TYPE_STRING);

  for (n = 0; n < num_values; n++) {
    from = gst_tag_list_get_value_index (taglist, tagname, n);
    g_value_transform (from, &to);

    buf = gst_buffer_new ();
    GST_BUFFER_DATA (buf) = g_strdup_printf ("%s:%s", tagname,
          g_value_get_string (&to));
    GST_BUFFER_SIZE (buf) = strlen (GST_BUFFER_DATA (buf));
    gst_pad_push (filter->srcpad, GST_DATA (buf));
  }

  g_value_unset (&to);
}

static void
gst_my_filter_task_func (GstElement *element)
{
  GstMyFilter *filter = GST_MY_FILTER (element);
  GstTagSetter *tagsetter = GST_TAG_SETTER (element);
  GstData *data;
```

```
  GstEvent *event;
  gboolean eos = FALSE;
  GstTagList *taglist = gst_tag_list_new ();

  while (!eos) {
    data = gst_pad_pull (filter->sinkpad);

    /* We're not very much interested in data right now */
    if (GST_IS_BUFFER (data))
      gst_buffer_unref (GST_BUFFER (data));
    event = GST_EVENT (data);

    switch (GST_EVENT_TYPE (event)) {
      case GST_EVENT_TAG:
        gst_tag_list_insert (taglist, gst_event_tag_get_list (event),
        GST_TAG_MERGE_PREPEND);
        gst_event_unref (event);
        break;
      case GST_EVENT_EOS:
        eos = TRUE;
        gst_event_unref (event);
        break;
      default:
        gst_pad_event_default (filter->sinkpad, event);
        break;
    }
  }

  /* merge tags with the ones retrieved from the application */
  if ((gst_tag_setter_get_tag_list (tagsetter)) {
    gst_tag_list_insert (taglist,
    gst_tag_setter_get_tag_list (tagsetter),
    gst_tag_setter_get_tag_merge_mode (tagsetter));
  }

  /* write tags */
  gst_tag_list_foreach (taglist, gst_my_filter_write_tag, filter);

  /* signal EOS */
  gst_pad_push (filter->srcpad, GST_DATA (gst_event_new (GST_EVENT_EOS)));
  gst_element_set_eos (element);
}
```

Note that normally, elements would not read the full stream before processing tags. Rather, they would read from each sinkpad until they've received data (since tags usually come in before the first data buffer) and process that.

Notes

1. ../../gstreamer/html/GstTagSetter.html
2. ../../gstreamer/html/gstreamer-GstTagList.html

Chapter 19. Events: Seeking, Navigation and More

There are many different event types but only two ways they can travel in the pipeline: downstream or upstream. It is very important to understand how both of these methods work because if one element in the pipeline is not handling them correctly the whole event system of the pipeline is broken. We will try to explain here how these methods work and how elements are supposed to implement them.

Downstream events

Downstream events are received through the sink pad's event handler, as set using `gst_pad_set_event_function ()` when the pad was created.

Downstream events can travel in two ways: they can be in-band (serialised with the buffer flow) or out-of-band (travelling through the pipeline instantly, possibly not in the same thread as the streaming thread that is processing the buffers, skipping ahead of buffers being processed or queued in the pipeline). The most common downstream events (NEWSEGMENT, EOS, TAG) are all serialised with the buffer flow.

Here is a typical event function:

```
static gboolean
gst_my_filter_sink_event (GstPad  *pad, GstEvent * event)
{
  GstMyFilter *filter;
  gboolean ret;

  filter = GST_MY_FILTER (gst_pad_get_parent (pad));
  ...

  switch (GST_EVENT_TYPE (event)) {
    case GST_EVENT_NEWSEGMENT:
      /* maybe save and/or update the current segment (e.g. for output
       * clipping) or convert the event into one in a different format
       * (e.g. BYTES to TIME) or drop it and set a flag to send a newsegment
       * event in a different format later */
      ret = gst_pad_push_event (filter->src_pad, event);
      break;
    case GST_EVENT_EOS:
      /* end-of-stream, we should close down all stream leftovers here */
      gst_my_filter_stop_processing (filter);
      ret = gst_pad_push_event (filter->src_pad, event);
      break;
    case GST_EVENT_FLUSH_STOP:
      gst_my_filter_clear_temporary_buffers (filter);
      ret = gst_pad_push_event (filter->src_pad, event);
      break;
    default:
      ret = gst_pad_event_default (pad, event);
      break;
  }

  ...
  gst_object_unref (filter);
  return ret;
```

```
}
```

If your element is chain-based, you will almost always have to implement a sink event function, since that is how you are notified about new segments and the end of the stream.

If your element is exclusively loop-based, you may or may not want a sink event function (since the element is driving the pipeline it will know the length of the stream in advance or be notified by the flow return value of `gst_pad_pull_range()`. In some cases even loop-based element may receive events from upstream though (for example audio decoders with an id3demux or apedemux element in front of them, or demuxers that are being fed input from sources that send additional information about the stream in custom events, as DVD sources do).

Upstream events

Upstream events are generated by an element somewhere downstream in the pipeline (example: a video sink may generate navigation events that informs upstream elements about the current position of the mouse pointer). This may also happen indirectly on request of the application, for example when the application executes a seek on a pipeline this seek request will be passed on to a sink element which will then in turn generate an upstream seek event.

The most common upstream events are seek events and Quality-of-Service (QoS) events.

An upstream event can be sent using the `gst_pad_send_event` function. This function simply call the default event handler of that pad. The default event handler of pads is `gst_pad_event_default`, and it basically sends the event to the peer pad. So upstream events always arrive on the src pad of your element and are handled by the default event handler except if you override that handler to handle it yourself. There are some specific cases where you have to do that :

- If you have multiple sink pads in your element. In that case you will have to decide which one of the sink pads you will send the event to (if not all of them).
- If you need to handle that event locally. For example a navigation event that you will want to convert before sending it upstream, or a QoS event that you want to handle.

The processing you will do in that event handler does not really matter but there are important rules you have to absolutely respect because one broken element event handler is breaking the whole pipeline event handling. Here they are :

- Always forward events you won't handle upstream using the default `gst_pad_event_default` method.
- If you are generating some new event based on the one you received don't forget to gst_event_unref the event you received.

- Event handler function are supposed to return TRUE or FALSE indicating if the event has been handled or not. Never simply return TRUE/FALSE in that handler except if you really know that you have handled that event.
- Remember that the event handler might be called from a different thread than the streaming thread, so make sure you use appropriate locking everywhere and at the beginning of the function obtain a reference to your element via the `gst_pad_get_parent()` (and release it again at the end of the function with `gst_object_unref ()`.

All Events Together

In this chapter follows a list of all defined events that are currently being used, plus how they should be used/interpreted. You can check the what type a certain event is using the GST_EVENT_TYPE macro (or if you need a string for debugging purposes you can use GST_EVENT_TYPE_NAME).

In this chapter, we will discuss the following events:

- End of Stream (EOS)
- Flush Start
- Flush Stop
- New Segment
- Seek Request
- Navigation
- Tag (metadata)

For more comprehensive information about events and how they should be used correctly in various circumstances please consult the GStreamer design documentation. This section only gives a general overview.

End of Stream (EOS)

End-of-stream events are sent if the stream that an element sends out is finished. An element receiving this event (from upstream, so it receives it on its sinkpad) will generally just process any buffered data (if there is any) and then forward the event further downstream. The `gst_pad_event_default ()` takes care of all this, so most elements do not need to support this event. Exceptions are elements that explicitly need to close a resource down on EOS, and N-to-1 elements. Note that the stream itself is *not* a resource that should be closed down on EOS! Applications might seek back to a point before EOS and continue playing again.

The EOS event has no properties, which makes it one of the simplest events in `GStreamer`. It is created using the `gst_event_new_eos()` function.

It is important to note that *only elements driving the pipeline should ever send an EOS event*. If your element is chain-based, it is not driving the pipeline. Chain-based elements should just return GST_FLOW_UNEXPECTED from their chain function at

the end of the stream (or the configured segment), the upstream element that is driving the pipeline will then take care of sending the EOS event (or alternatively post a SEGMENT_DONE message on the bus depending on the mode of operation). If you are implementing your own source element, you also do not need to ever manually send an EOS event, you should also just return GST_FLOW_UNEXPECTED in your create function (assuming your element derives from GstBaseSrc or GstPushSrc).

Flush Start

The flush start event is sent downstream if all buffers and caches in the pipeline should be emptied. "Queue" elements will empty their internal list of buffers when they receive this event, for example. File sink elements (e.g. "filesink") will flush the kernel-to-disk cache (`fdatasync ()` or `fflush ()`) when they receive this event. Normally, elements receiving this event will simply just forward it, since most filter or filter-like elements don't have an internal cache of data. `gst_pad_event_default ()` does just that, so for most elements, it is enough to forward the event using the default event handler.

As a side-effect of flushing all data from the pipeline, this event unblocks the streaming thread by making all pads reject data until they receive a Flush Stop signal (elements trying to push data will get a WRONG_STATE flow return and stop processing data).

The flush-start event is created with the `gst_event_new_flush_start ()`. Like the EOS event, it has no properties. This event is usually only created by elements driving the pipeline, like source elements operating in push-mode or pull-range based demuxers/decoders.

Flush Stop

The flush-stop event is sent by an element driving the pipeline after a flush-start and tells pads and elements downstream that they should accept events and buffers again (there will be at least a NEWSEGMENT event before any buffers first though).

If your element keeps temporary caches of stream data, it should clear them when it receives a FLUSH-STOP event (and also whenever its chain function receives a buffer with the DISCONT flag set).

The flush-stop event is created with `gst_event_new_flush_stop ()`. Like the EOS event, it has no properties.

New Segment

A new segment event is sent downstream to either announce a new segment of data in the data stream or to update the current segment with new values. A new segment event must always be sent before the first buffer of data and after a flush (see above).

The first new segment event is created by the element driving the pipeline, like a source operating in push-mode or a demuxer/decoder operating pull-based. This new segment event then travels down the pipeline and may be transformed on the

way (a decoder, for example, might receive a new-segment event in BYTES format and might transform this into a new-segment event in TIMES format based on the average bitrate).

New segment events may also be used to indicate 'gaps' in the stream, like in a subtitle stream for example where there may not be any data at all for a considerable amount of (stream) time. This is done by updating the segment start of the current segment (see the design documentation for more details).

Depending on the element type, the event can simply be forwarded using `gst_pad_event_default ()`, or it should be parsed and a modified event should be sent on. The last is true for demuxers, which generally have a byte-to-time conversion concept. Their input is usually byte-based, so the incoming event will have an offset in byte units (GST_FORMAT_BYTES), too. Elements downstream, however, expect new segment events in time units, so that it can be used to update the pipeline clock. Therefore, demuxers and similar elements should not forward the event, but parse it, free it and send a new newsegment event (in time units, GST_FORMAT_TIME) further downstream.

The newsegment event is created using the function `gst_event_new_new_segment ()`. See the API reference and design document for details about its parameters.

Elements parsing this event can use gst_event_parse_new_segment_full() to extract the event details. Elements may find the GstSegment API useful to keep track of the current segment (if they want to use it for output clipping, for example).

Seek Request

Seek events are meant to request a new stream position to elements. This new position can be set in several formats (time, bytes or "default units" [a term indicating frames for video, channel-independent samples for audio, etc.]). Seeking can be done with respect to the end-of-file, start-of-file or current position, and usually happens in upstream direction (downstream seeking is done by sending a NEWSEGMENT event with the appropriate offsets for elements that support that, like filesink).

Elements receiving seek events should, depending on the element type, either just forward it upstream (filters, decoders), change the format in which the event is given and then forward it (demuxers), or handle the event by changing the file pointer in their internal stream resource (file sources, demuxers/decoders driving the pipeline in pull-mode) or something else.

Seek events are built up using positions in specified formats (time, bytes, units). They are created using the function `gst_event_new_seek ()`. Note that many plugins do not support seeking from the end of the stream or from the current position. An element not driving the pipeline and forwarding a seek request should not assume that the seek succeeded or actually happened, it should operate based on the NEWSEGMENT events it receives.

Elements parsing this event can do this using `gst_event_parse_seek()`.

Navigation

Navigation events are sent upstream by video sinks to inform upstream elements of where the mouse pointer is, if and where mouse pointer clicks have happened, or if keys have been pressed or released.

All this information is contained in the event structure which can be obtained with `gst_event_get_structure ()`.

Check out the navigationtest element in gst-plugins-good for an idea how to extract navigation information from this event.

Tag (metadata)

Tagging events are being sent downstream to indicate the tags as parsed from the stream data. This is currently used to preserve tags during stream transcoding from one format to the other. Tags are discussed extensively in Chapter 18. Most elements will simply forward the event by calling `gst_pad_event_default ()`.

The tag event is created using the function `gst_event_new_tag ()`, but more often elements will use either the `gst_element_found_tags ()` function or the `gst_element_found_tags_for_pad ()`, which will do both: post a tag message on the bus and send a tag event downstream. All of these functions require a filled-in taglist as argument, which they will take ownership of.

Elements parsing this event can use the function `gst_event_parse_tag ()` to acquire the taglist that the event contains.

Chapter 20. Pre-made base classes

So far, we've been looking at low-level concepts of creating any type of GStreamer element. Now, let's assume that all you want is to create an simple audiosink that works exactly the same as, say, "esdsink", or a filter that simply normalizes audio volume. Such elements are very general in concept and since they do nothing special, they should be easier to code than to provide your own scheduler activation functions and doing complex caps negotiation. For this purpose, GStreamer provides base classes that simplify some types of elements. Those base classes will be discussed in this chapter.

Writing a sink

Sinks are special elements in GStreamer. This is because sink elements have to take care of *preroll*, which is the process that takes care that elements going into the GST_STATE_PAUSED state will have buffers ready after the state change. The result of this is that such elements can start processing data immediately after going into the GST_STATE_PLAYING state, without requiring to take some time to initialize outputs or set up decoders; all that is done already before the state-change to GST_STATE_PAUSED successfully completes.

Preroll, however, is a complex process that would require the same code in many elements. Therefore, sink elements can derive from the GstBaseSink base-class, which does preroll and a few other utility functions automatically. The derived class only needs to implement a bunch of virtual functions and will work automatically.

The GstBaseSink base-class specifies some limitations on elements, though:

- It requires that the sink only has one sinkpad. Sink elements that need more than one sinkpad, cannot use this base-class.
- The base-class owns the pad, and specifies caps negotiation, data handling, pad allocation and such functions. If you need more than the ones provided as virtual functions, then you cannot use this base-class.
- By implementing the pad_allocate () function, it is possible for upstream elements to use special memory, such as memory on the X server side that only the sink can allocate, or even hardware memory mmap ()'ed from the kernel. Note that in almost all cases, you will want to subclass the GstBuffer object, so that your own set of functions will be called when the buffer loses its last reference.

Sink elements can derive from GstBaseSink using the usual GObject type creation voodoo, or by using the convenience macro GST_BOILERPLATE ():

```
GST_BOILERPLATE_FULL (GstMySink, gst_my_sink, GstBaseSink, GST_TYPE_BASE_SINK);

[..]

static void
gst_my_sink_class_init (GstMySinkClass * klass)
{
  klass->set_caps = [..];
  klass->render = [..];
```

```
[..]
}
```

The advantages of deriving from `GstBaseSink` are numerous:

- Derived implementations barely need to be aware of preroll, and do not need to know anything about the technical implementation requirements of preroll. The base-class does all the hard work.

  Less code to write in the derived class, shared code (and thus shared bugfixes).

There are also specialized base classes for audio and video, let's look at those a bit.

Writing an audio sink

Essentially, audio sink implementations are just a special case of a general sink. There are two audio base classes that you can choose to derive from, depending on your needs: `GstBaseAudiosink` and `GstAudioSink`. The baseaudiosink provides full control over how synchronization and scheduling is handled, by using a ringbuffer that the derived class controls and provides. The audiosink base-class is a derived class of the baseaudiosink, implementing a standard ringbuffer implementing default synchronization and providing a standard audio-sample clock. Derived classes of this base class merely need to provide a `_open ()`, `_close ()` and a `_write ()` function implementation, and some optional functions. This should suffice for many sound-server output elements and even most interfaces. More demanding audio systems, such as Jack, would want to implement the `GstBaseAudioSink` base-class.

The `GstBaseAusioSink` has little to no limitations and should fit virtually every implementation, but is hard to implement. The `GstAudioSink`, on the other hand, only fits those systems with a simple `open ()` / `close ()` / `write ()` API (which practically means pretty much all of them), but has the advantage that it is a lot easier to implement. The benefits of this second base class are large:

- Automatic synchronization, without any code in the derived class.
- Also automatically provides a clock, so that other sinks (e.g. in case of audio/video playback) are synchronized.
- Features can be added to all audiosinks by making a change in the base class, which makes maintainance easy.
- Derived classes require only three small functions, plus some `GObject` boilerplate code.

In addition to implementing the audio base-class virtual functions, derived classes can (should) also implement the `GstBaseSink` `set_caps ()` and `get_caps ()` virtual functions for negotiation.

Writing a video sink

Writing a videosink can be done using the `GstVideoSink` base-class, which derives from `GstBaseSink` internally. Currently, it does nothing yet but add another compile

dependency, so derived classes will need to implement all base-sink virtual functions. When they do this correctly, this will have some positive effects on the end user experience with the videosink:

- Because of preroll (and the `preroll ()` virtual function), it is possible to display a video frame already when going into the `GST_STATE_PAUSED` state.
- By adding new features to `GstVideoSink`, it will be possible to add extensions to videosinks that affect all of them, but only need to be coded once, which is a huge maintainance benefit.

Writing a source

In the previous part, particularly Providing random access, we have learned that some types of elements can provide random access. This applies most definitely to source elements reading from a randomly seekable location, such as file sources. However, other source elements may be better described as a live source element, such as a camera source, an audio card source and such; those are not seekable and do not provide byte-exact access. For all such use cases, `GStreamer` provides two base classes: `GstBaseSrc` for the basic source functionality, and `GstPushSrc`, which is a non-byte exact source base-class. The pushsource base class itself derives from basesource as well, and thus all statements about the basesource apply to the pushsource, too.

The basesrc class does several things automatically for derived classes, so they no longer have to worry about it:

- Fixes to `GstBaseSrc` apply to all derived classes automatically.
- Automatic pad activation handling, and task-wrapping in case we get assigned to start a task ourselves.

The `GstBaseSrc` may not be suitable for all cases, though; it has limitations:

- There is one and only one sourcepad. Source elements requiring multiple sourcepads cannot use this base-class.
- Since the base-class owns the pad and derived classes can only control it as far as the virtual functions allow, you are limited to the functionality provided by the virtual functions. If you need more, you cannot use this base-class.

It is possible to use special memory, such as X server memory pointers or `mmap ()`'ed memory areas, as data pointers in buffers returned from the `create()` virtual function. In almost all cases, you will want to subclass `GstBuffer` so that your own set of functions can be called when the buffer is destroyed.

Writing an audio source

An audio source is nothing more but a special case of a pushsource. Audio sources would be anything that reads audio, such as a source reading from a

soundserver, a kernel interface (such as ALSA) or a test sound / signal generator. GStreamer provides two base classes, similar to the two audiosinks described in Writing an audio sink; one is ringbuffer-based, and requires the derived class to take care of its own scheduling, synchronization and such. The other is based on this GstBaseAudioSrc and is called GstAudioSrc, and provides a simple open (), close () and read () interface, which is rather simple to implement and will suffice for most soundserver sources and audio interfaces (e.g. ALSA or OSS) out there.

The GstAudioSrc base-class has several benefits for derived classes, on top of the benefits of the GstPushSrc base-class that it is based on:

- Does syncronization and provides a clock.
- New features can be added to it and will apply to all derived classes automatically.

Writing a transformation element

A third base-class that GStreamer provides is the GstBaseTransform. This is a base class for elements with one sourcepad and one sinkpad which act as a filter of some sort, such as volume changing, audio resampling, audio format conversion, and so on and so on. There is quite a lot of bookkeeping that such elements need to do in order for things such as buffer allocation forwarding, passthrough, in-place processing and such to all work correctly. This base class does all that for you, so that you just need to do the actual processing.

Since the GstBaseTransform is based on the 1-to-1 model for filters, it may not apply well to elements such as decoders, which may have to parse properties from the stream. Also, it will not work for elements requiring more than one sourcepad or sinkpad.

Chapter 21. Writing a Demuxer or Parser

Demuxers are the 1-to-N elements that need very special care. They are responsible for timestamping raw, unparsed data into elementary video or audio streams, and there are many things that you can optimize or do wrong. Here, several culprits will be mentioned and common solutions will be offered. Parsers are demuxers with only one source pad. Also, they only cut the stream into buffers, they don't touch the data otherwise.

As mentioned previously in Caps negotiation, demuxers should use fixed caps, since their data type will not change.

As discussed in Different scheduling modes, demuxer elements can be written in multiple ways:

- They can be the driving force of the pipeline, by running their own task. This works particularly well for elements that need random access, for example an AVI demuxer.
- They can also run in push-based mode, which means that an upstream element drives the pipeline. This works particularly well for streams that may come from network, such as Ogg.

In addition, audio parsers with one output can, in theory, also be written in random access mode. Although simple playback will mostly work if your element only accepts one mode, it may be required to implement multiple modes to work in combination with all sorts of applications, such as editing. Also, performance may become better if you implement multiple modes. See Different scheduling modes to see how an element can accept multiple scheduling modes.

Chapter 22. Writing a N-to-1 Element or Muxer

N-to-1 elements have been previously mentioned and discussed in both Chapter 13 and in Different scheduling modes. The main noteworthy thing about N-to-1 elements is that each pad is push-based in its own thread, and the N-to-1 element synchronizes those streams by expected-timestamp-based logic. This means it lets all streams wait except for the one that provides the earliest next-expected timestamp. When that stream has passwed one buffer, the next earliest-expected-timestamp is calculated, and we start back where we were, until all streams have reached EOS. There is a helper base class, called `GstCollectPads`, that will help you to do this.

Note, however, that this helper class will only help you with grabbing a buffer from each input and giving you the one with earliest timestamp. If you need anything more difficult, such as "don't-grab-a-new-buffer until a given timestamp" or something like that, you'll need to do this yourself.

Chapter 23. Writing a Manager

Managers are elements that add a function or unify the function of another (series of) element(s). Managers are generally a `GstBin` with one or more ghostpads. Inside them is/are the actual element(s) that matters. There is several cases where this is useful. For example:

- To add support for private events with custom event handling to another element.
- To add support for custom pad `_query ()` or `_convert ()` handling to another element.
- To add custom data handling before or after another element's data handler function (generally its `_chain ()` function).
- To embed an element, or a series of elements, into something that looks and works like a simple element to the outside world.

Making a manager is about as simple as it gets. You can derive from a `GstBin`, and in most cases, you can embed the required elements in the `_init ()` already, including setup of ghostpads. If you need any custom data handlers, you can connect signals or embed a second element which you control.

Chapter 24. Things to check when writing an element

This chapter contains a fairly random selection of things to take care of when writing an element. It's up to you how far you're going to stick to those guidelines. However, keep in mind that when you're writing an element and hope for it to be included in the mainstream `GStreamer` distribution, it *has to* meet those requirements. As far as possible, we will try to explain why those requirements are set.

About states

- Make sure the state of an element gets reset when going to `NULL`. Ideally, this should set all object properties to their original state. This function should also be called from _init.
- Make sure an element forgets *everything* about its contained stream when going from `PAUSED` to `READY`. In `READY`, all stream states are reset. An element that goes from `PAUSED` to `READY` and back to `PAUSED` should start reading the stream from he start again.
- People that use **gst-launch** for testing have the tendency to not care about cleaning up. This is *wrong*. An element should be tested using various applications, where testing not only means to "make sure it doesn't crash", but also to test for memory leaks using tools such as **valgrind**. Elements have to be reusable in a pipeline after having been reset.

Debugging

- Elements should *never* use their standard output for debugging (using functions such as `printf ()` or `g_print ()`). Instead, elements should use the logging functions provided by `GStreamer`, named `GST_DEBUG ()`, `GST_LOG ()`, `GST_INFO ()`, `GST_WARNING ()` and `GST_ERROR ()`. The various logging levels can be turned on and off at runtime and can thus be used for solving issues as they turn up. Instead of `GST_LOG ()` (as an example), you can also use `GST_LOG_OBJECT ()` to print the object that you're logging output for.
- Ideally, elements should use their own debugging category. Most elements use the following code to do that:

```
GST_DEBUG_CATEGORY_STATIC (myelement_debug);
#define GST_CAT_DEFAULT myelement_debug

[..]

static void
gst_myelement_class_init (GstMyelementClass *klass)
{
[..]
  GST_DEBUG_CATEGORY_INIT (myelement_debug, "myelement",
      0, "My own element");
}
```

At runtime, you can turn on debugging using the commandline option **--gst-debug=myelement:5**.

- Elements should use GST_DEBUG_FUNCPTR when setting pad functions or overriding element class methods, for example:

```
gst_pad_set_event_func (myelement->srcpad,
    GST_DEBUG_FUNCPTR (my_element_src_event));
```

  This makes debug output much easier to read later on.
- Elements that are aimed for inclusion into one of the GStreamer modules should ensure consistent naming of the element name, structures and function names. For example, if the element type is GstYellowFooDec, functions should be prefixed with gst_yellow_foo_dec_ and the element should be registered as 'yellowfoodec'. Separate words should be separate in this scheme, so it should be GstFooDec and gst_foo_dec, and not GstFoodec and gst_foodec.

Querying, events and the like

- All elements to which it applies (sources, sinks, demuxers) should implement query functions on their pads, so that applications and neighbour elements can request the current position, the stream length (if known) and so on.
- Elements should make sure they forward events they do not handle with gst_pad_event_default (pad, event) instead of just dropping them. Events should never be dropped unless specifically intended.
- Elements should make sure they forward queries they do not handle with gst_pad_query_default (pad, query) instead of just dropping them.
- Elements should use gst_pad_get_parent() in event and query functions, so that they hold a reference to the element while they are operating. Note that gst_pad_get_parent() increases the reference count of the element, so you must be very careful to call gst_object_unref (element) before returning from your query or event function, otherwise you will leak memory.

Testing your element

- **gst-launch** is *not* a good tool to show that your element is finished. Applications such as Rhythmbox and Totem (for GNOME) or AmaroK (for KDE) *are*. **gst-launch** will not test various things such as proper clean-up on reset, interrupt event handling, querying and so on.
- Parsers and demuxers should make sure to check their input. Input cannot be trusted. Prevent possible buffer overflows and the like. Feel free to error out on unrecoverable stream errors. Test your demuxer using stream corruption elements such as `breakmydata` (included in gst-plugins). It will randomly insert, delete and

modify bytes in a stream, and is therefore a good test for robustness. If your element crashes when adding this element, your element needs fixing. If it errors out properly, it's good enough. Ideally, it'd just continue to work and forward data as much as possible.

- Demuxers should not assume that seeking works. Be prepared to work with unseekable input streams (e.g. network sources) as well.
- Sources and sinks should be prepared to be assigned another clock then the one they expose themselves. Always use the provided clock for synchronization, else you'll get A/V sync issues.

Chapter 25. Porting 0.8 plug-ins to 0.10

This section of the appendix will discuss shortly what changes to plugins will be needed to quickly and conveniently port most applications from GStreamer-0.8 to GStreamer-0.10, with references to the relevant sections in this Plugin Writer's Guide where needed. With this list, it should be possible to port most plugins to GStreamer-0.10 in less than a day. Exceptions are elements that will require a base class in 0.10 (sources, sinks), in which case it may take a lot longer, depending on the coder's skills (however, when using the GstBaseSink and GstBaseSrc base-classes, it shouldn't be all too bad), and elements requiring the deprecated bytestream interface, which should take 1-2 days with random access. The scheduling parts of muxers will also need a rewrite, which will take about the same amount of time.

List of changes

- Discont events have been replaced by newsegment events. In 0.10, it is essential that you send a newsegment event downstream before you send your first buffer (in 0.8 the scheduler would invent discont events if you forgot them, in 0.10 this is no longer the case).
- In 0.10, buffers have caps attached to them. Elements should allocate new buffers with gst_pad_alloc_buffer (). See Caps negotiation for more details.
- Most functions returning an object or an object property have been changed to return its own reference rather than a constant reference of the one owned by the object itself. The reason for this change is primarily threadsafety. This means effectively that return values of functions such as gst_element_get_pad (), gst_pad_get_name (), gst_pad_get_parent (), gst_object_get_parent (), and many more like these have to be free'ed or unreferenced after use. Check the API references of each function to know for sure whether return values should be free'ed or not.
- In 0.8, scheduling could happen in any way. Source elements could be _get ()-based or _loop ()-based, and any other element could be _chain ()-based or _loop ()-based, with no limitations. Scheduling in 0.10 is simpler for the scheduler, and the element is expected to do some more work. Pads get assigned a scheduling mode, based on which they can either operate in random access-mode, in pipeline driving mode or in push-mode. all this is documented in detail in Different scheduling modes. As a result of this, the bytestream object no longer exists. Elements requiring byte-level access should now use random access on their sinkpads.
- Negotiation is asynchronous. This means that downstream negotiation is done as data comes in and upstream negotiation is done whenever renegotiation is required. All details are described in Caps negotiation.
- For as far as possible, elements should try to use existing base classes in 0.10. Sink and source elements, for example, could derive from GstBaseSrc and GstBaseSink. Audio sinks or sources could even derive from audio-specific base classes. All existing base classes have been discussed in Pre-made base classes and the next few chapters.

- In 0.10, event handling and buffers are separated once again. This means that in order to receive events, one no longer has to set the GST_FLAG_EVENT_AWARE flag, but can simply set an event handling function on the element's sinkpad(s), using the function gst_pad_set_event_function (). The _chain ()-function will only receive buffers.
- Although core will wrap most threading-related locking for you (e.g. it takes the stream lock before calling your data handling functions), you are still responsible for locking around certain functions, e.g. object properties. Be sure to lock properly here, since applications will change those properties in a different thread than the thread which does the actual data passing! You can use the GST_OBJECT_LOCK () and GST_OBJECT_UNLOCK () helpers in most cases, fortunately, which grabs the default property lock of the element.
- GstValueFixedList and all *_fixed_list_* () functions were renamed to GstValueArray and *_array_* ().
- The semantics of GST_STATE_PAUSED and GST_STATE_PLAYING have changed for elements that are not sink elements. Non-sink elements need to be able to accept and process data already in the GST_STATE_PAUSED state now (ie. when prerolling the pipeline). More details can be found in Chapter 6.
- If your plugin's state change function hasn't been superseded by virtual start() and stop() methods of one of the new base classes, then your plugin's state change functions may need to be changed in order to safely handle concurrent access by multiple threads. Your typical state change function will now first handle upwards state changes, then chain up to the state change function of the parent class (usually GstElementClass in these cases), and only then handle downwards state changes. See the vorbis decoder plugin in gst-plugins-base for an example.

  The reason for this is that in the case of downwards state changes you don't want to destroy allocated resources while your plugin's chain function (for example) is still accessing those resources in another thread. Whether your chain function might be running or not depends on the state of your plugin's pads, and the state of those pads is closely linked to the state of the element. Pad states are handled in the GstElement class's state change function, including proper locking, that's why it is essential to chain up before destroying allocated resources.

  As already mentioned above, you should really rewrite your plugin to derive from one of the new base classes though, so you don't have to worry about these things, as the base class will handle it for you. There are no base classes for decoders and encoders yet, so the above paragraphs about state changes definitively apply if your plugin is a decoder or an encoder.
- gst_pad_set_link_function (), which used to set a function that would be called when a format was negotiated between two GstPads, now sets a function that is called when two elements are linked together in an application. For all practical purposes, you most likely want to use the function gst_pad_set_setcaps_function (), nowadays, which sets a function that is called when the format streaming over a pad changes (so similar to _set_link_function () in GStreamer-0.8).

  If the element is derived from a GstBase class, then override the set_caps ().
- gst_pad_use_explicit_caps () has been replaced by gst_pad_use_fixed_caps (). You can then set the fixed caps to use on a pad

with `gst_pad_set_caps ()`.

Chapter 26. GStreamer licensing

How to license the code you write for `GStreamer`

GStreamer is a plugin-based framework licensed under the LGPL. The reason for this choice in licensing is to ensure that everyone can use GStreamer to build applications using licenses of their choice.

To keep this policy viable, the GStreamer community has made a few licensing rules for code to be included in GStreamer's core or GStreamer's official modules, like our plugin packages. We require that all code going into our core package is LGPL. For the plugin code, we require the use of the LGPL for all plugins written from scratch or linking to external libraries. The only exception to this is when plugins contain older code under more liberal licenses (like the MPL or BSD). They can use those licenses instead and will still be considered for inclusion. We do not accept GPL code to be added to our plugins module, but we do accept LGPL-licensed plugins using an external GPL library. The reason for demanding plugins be licensed under the LGPL, even when using a GPL library, is that other developers might want to use the plugin code as a template for plugins linking to non-GPL libraries.

We also plan on splitting out the plugins using GPL libraries into a separate package eventually and implement a system which makes sure an application will not be able to access these plugins unless it uses some special code to do so. The point of this is not to block GPL-licensed plugins from being used and developed, but to make sure people are not unintentionally violating the GPL license of said plugins.

This advisory is part of a bigger advisory with a FAQ which you can find on the GStreamer website[1]

Notes

1. http://gstreamer.freedesktop.org/documentation/licensing.html