GStreamer Plugin Writer's Guide (0.8.2)

Richard John Boulton

Erik Walthinsen

Steve Baker

Leif Johnson

Ronald S. Bultje

GStreamer Plugin Writer's Guide (0.8.2)

by Richard John Boulton, Erik Walthinsen, Steve Baker, Leif Johnson, and Ronald S. Bultje

Table of Contents

Chapter 1. Preface

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 documentation available on the GStreamer web site[1].

Since GStreamer adheres to the GObject programming model, this guide also assumes that you understand the basics of GObject[2] programming. There are several good introductions to the GObject library, including the *GTK+ Tutorial*[3].

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. The discussion begins by giving examples of generating the basic structures with Constructing the Boilerplate. Then you will learn how to write the code to get a basic filter plugin working: These steps include chapters on Chapter 13, Chapter 4, Chapter 5, and (WRITEME: building state).

  After you have finished the first steps, you will be able to create a working plugin, but your new plugin might not have all the functionality you need. To provide some standard functionality, you will learn how to add more features to a new plugin. These features are described in the chapters on (WRITEME) and Chapter 8. Finally, you will see in (WRITEME) how to write a short test application to try out your new plugin.

- 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 Chapter 14, . Since these features are more advanced, the chapters can basically be read in any order, as you find that your plugins require these features.

- Other 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 part includes chapters on Writing a Source, Writing a Sink, and Writing an Autoplugger.

- Appendices - Further information for plugin developers.

  The appendices contain some information that stubbornly refuses to fit cleanly in other sections of the guide. This information includes (WRITEME) and FIXME: organize better.

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://www.gtk.org/tutorial/

Chapter 2. Basic Concepts

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 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 link 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[1] and GstPlugin[2].

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[3].

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 20. 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 GstData[4], GstBuffer[5] and GstEvent[6].

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 Basic Types

Mime Type	Description	Property	Property Type	Property Values	Property Description

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.
audio/x-raw-int	Unstructured and uncompressed raw integer audio data.	endianness	integer	G_BIG_ENDIAN (1234) or G_LITTLE_ENDIAN (4321)	The order of bytes in a sample. The value G_LITTLE_ENDIAN (4321) means "little-endian" (byte-order is "least significant byte first"). The value G_BIG_ENDIAN (1234) 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.
		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.

Mime Type	Description	Property	Property Type	Property Values	Property Description
		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/gstelement.html
2. ../../gstreamer/gstreamer-gstplugin.html
3. ../../gstreamer/gstreamer-gstpad.html
4. ../../gstreamer/gstreamer-gstdata.html
5. ../../gstreamer/gstreamer-gstbuffer.html
6. ../../gstreamer/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` CVS 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 $ cvs -d:pserver:anoncvs@cvs.freedesktop.org/cvs/gstreamer co login
Logging in to :pserver:anoncvs@cvs.freedesktop.org:2401/cvs/gstreamer
CVS password: [ENTER]

shell $ cvs -z3 -d:pserver:anoncvs@cvs.freedesktop.org:/cvs/gstreamer co gst-template
U gst-template/README
U gst-template/gst-app/AUTHORS
U gst-template/gst-app/ChangeLog
U gst-template/gst-app/Makefile.am
U gst-template/gst-app/NEWS
U gst-template/gst-app/README
U gst-template/gst-app/autogen.sh
U gst-template/gst-app/configure.ac
U gst-template/gst-app/src/Makefile.am
...
```

After the first command, you will have to press **ENTER** to log in to the CVS server. (You might have to log in twice.) The second command will check out a series of files and directories into `./gst-template`. The template you will be using is in `./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.

!!! FIXME !!! Using the Project Stamp

This section needs some fixing from someone that is aware of how this works. The only tool that looks like the ones cited there is `gst-plugins/tools/filterstamp.sh`

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 are some tools in the `./gst-plugins/tools/` directory. One tool, `gst-quick-stamp`, is a quick command line tool. The other, `gst-project-stamp`, is

a full GNOME druid application that takes you through the steps of creating a new project (either a plugin or an application).

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

1. the name of the plugin, and
2. the directory that should hold a new subdirectory for the source tree of the plugin.

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 a new directory called `~/src/examplefilter/`:

```
shell $ cd gst-template
shell $ tools/pluginstamp.sh ExampleFilter ~/src
```

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

```
/* Definition of structure storing data for this element. */
typedef struct _GstExample GstExample;

struct _GstExample {
  GstElement element;

  GstPad *sinkpad, *srcpad;

  gboolean silent;
};

/* Standard definition defining a class for this element. */
typedef struct _GstExampleClass GstExampleClass;
struct _GstExampleClass {
  GstElementClass parent_class;
};

/* Standard macros for defining types for this element.  */
#define GST_TYPE_EXAMPLE \
  (gst_example_get_type())
#define GST_EXAMPLE(obj) \
  (GTK_CHECK_CAST((obj),GST_TYPE_EXAMPLE,GstExample))
#define GST_EXAMPLE_CLASS(klass) \
  (GTK_CHECK_CLASS_CAST((klass),GST_TYPE_EXAMPLE,GstExample))
#define GST_IS_EXAMPLE(obj) \
  (GTK_CHECK_TYPE((obj),GST_TYPE_EXAMPLE))
#define GST_IS_EXAMPLE_CLASS(obj) \
  (GTK_CHECK_CLASS_TYPE((klass),GST_TYPE_EXAMPLE))

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

GstElementDetails

The GstElementDetails structure gives a heirarchical 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, as a heirarchy. The heirarchy is defined by specifying the top level category, followed by a "/", followed by the next level category, etc. The type should be defined according to the guidelines elsewhere in this document. (FIXME: write the guidelines, and give a better reference to them)
- 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 GstElementDetails example_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 `_base_init ()`.

```
static void
gst_my_filter_base_init (GstMyFilterClass *klass)
{
  static GstElementDetails my_filter_details = {
[..]
  };
  GstElementClass *element_class = GST_ELEMENT_CLASS (klass);

[..]
  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.

```
static void
gst_my_filter_base_init (GstMyFilterClass *klass)
{
  static GstStaticPadTemplate sink_factory =
[..]
  , src_factory =
[..]
  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 {} are lists, values surrounded by [] 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 `_link ()` and `_getcaps ()` functions to let other elements know their capabilities 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 `_link ()` function pointer and a `_getcaps ()` function pointer. Optionally, you can set a `_chain ()` function pointer (on sink pads in filter and sink elements) through which data will come in to the element, or (on source pads in source elements) a `_get ()` function pointer through which data will be pulled from the element. After that, you have to register the pad with the element. This happens like this:

```
static GstPadLinkReturn gst_my_filter_link (GstPad        *pad,
       const GstCaps *caps);
static GstCaps * gst_my_filter_getcaps (GstPad        *pad);
static void  gst_my_filter_chain (GstPad        *pad,
       GstData       *data);

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

  /* 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_link_function (filter->sinkpad, gst_my_filter_link);
  gst_pad_set_getcaps_function (filter->sinkpad, gst_my_filter_getcaps);
  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_pad_set_link_function (filter->srcpad, gst_my_filter_link);
  gst_pad_set_getcaps_function (filter->srcpad, gst_my_filter_getcaps);
  gst_element_add_pad (GST_ELEMENT (filter), filter->srcpad);
[..]
}
```

The link function

The `_link ()` is called during 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 13. A `_link ()` receives a pointer to a `GstCaps` struct that defines the proposed streamtype, and can respond with either "yes" (`GST_PAD_LINK_OK`), "no" (`GST_PAD_LINK_REFUSED`) or "don't know yet" (`GST_PAD_LINK_DELAYED`). If the element responds positively towards the streamtype, that type will be used on the pad. An example:

```
static GstPadLinkReturn
gst_my_filter_link (GstPad        *pad,
      const GstCaps *caps)
{
  GstStructure *structure = gst_caps_get_structure (caps, 0);
  GstMyFilter *filter = GST_MY_FILTER (gst_pad_get_parent (pad));
```

```
  GstPad *otherpad = (pad == filter->srcpad) ? filter->sinkpad :
            filter->srcpad;
  GstPadLinkReturn ret;
  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 GST_PAD_LINK_REFUSED;
  }

  /* 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. */
  ret = gst_pad_try_set_caps (otherpad, gst_caps_copy (caps));
  if (GST_PAD_LINK_FAILED (ret))
    return ret;

  /* 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 ret;
}
```

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` internally. A `GstCaps` 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`.

The getcaps function

The `_getcaps ()` funtion is used to request the list of supported formats and properties from the element. In some cases, this will be equal to the formats provided by the pad template, in which case this function can be omitted. In some cases, too, it will not depend on anything inside this element, but it will rather depend on the input from another element linked to this element's sink or source pads. In that case, you can use `gst_pad_proxy_getcaps` as implementation, it provides getcaps forwarding in the core. However, in many cases, the format supported by this element cannot be defined externally, but is more specific than those provided by the pad template. In this case, you should use a `_getcaps ()` function. In the case as specified below, we assume that our filter is able to resample sound, so it would be able to provide any samplerate (indifferent from the samplerate specified on the other pad) on both pads. It explains how a `_getcaps ()` can be used to do this.

```
static GstCaps *
gst_my_filter_getcaps (GstPad *pad)
```

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

  if (gst_caps_is_empty (othercaps))
    return othercaps;

  /* 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_free (othercaps);

  return caps;
}
```

Explicit caps

Obviously, many elements will not need this complex mechanism, because they are much simpler than that. They only support one format, or their format is fixed but the contents of the format depend on the stream or something else. In those cases, *explicit caps* are an easy way of handling caps. Explicit caps are an easy way of specifying one, fixed, supported format on a pad. Pads using explicit caps do not implement their own `_getcaps ()` or `_link ()` functions. When the exact format is known, an elements uses `gst_pad_set_explicit_caps ()` to specify the exact format. This is very useful for demuxers, for example.

```
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_use_explicit_caps (filter->srcpad);
[..]
}

static void
gst_my_filter_somefunction (GstMyFilter *filter)
{
  GstCaps *caps = ..;
[..]
  gst_pad_set_explicit_caps (filter->srcpad, caps);
[..]
}
```

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 lineair functions - so for each incoming buffer, one buffer will go out, too. Below is a very simple implementation of a chain function:

```
static void
gst_my_filter_chain (GstPad  *pad,
       GstData *data)
{
  GstMyFilter *filter = GST_MY_FILTER (gst_pad_get_parent (pad));
  GstBuffer *buf = GST_BUFFER (data);

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

  gst_pad_push (filter->srcpad, GST_DATA (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), the incoming data might not even be a buffer.

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

  if (GST_IS_EVENT (data)) {
    GstEvent *event = GST_EVENT (data);

    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);
        /* fall-through to default event handling */
      default:
        gst_pad_event_default (pad, event);
        break;
    }
    return;
  }

  buf = GST_BUFFER (data);
  outbuf = gst_my_filter_process_data (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_pad_push (filter->srcpad, GST_DATA (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 and GST_STATE_PLAYING.

GST_STATE_NULL (from now on referred to as "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 (from now on referred to as "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_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_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 (from now on referred to as "PAUSED") is a state in which an element is by all means able to handle data; the only 'but' here is that it doesn't actually handle any data. When going from the READY state into the PAUSED state (GST_STATE_READY_TO_PAUSED), the element will usually not do anything at all: all stream-specific info is generally handled in the _link (), which is called during caps negotiation. Exceptions to this rule are, for example, files: these are considered stream-specific data (since one file is one stream), and should thus be opened in this state change. When going from the PAUSED back to READY (GST_STATE_PAUSED_TO_READY), all stream-specific data should be discarded.

GST_STATE_PLAYING (from now on referred to as "PLAYING") is the highest state that an element can be in. It is similar to PAUSED, except that now, data is actually passing over the pipeline. The transition from PAUSED to PLAYING (GST_STATE_PAUSED_TO_PLAYING) should be as small as possible and would ideally cause no delay at all. The same goes for the reverse transition (GST_STATE_PLAYING_TO_PAUSED).

Mangaging filter state

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.

```
static GstElementStateReturn
  gst_my_filter_change_state (GstElement *element);

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 GstElementStateReturn
gst_my_filter_change_state (GstElement *element)
{
  GstMyFilter *filter = GST_MY_FILTER (element);

  switch (GST_STATE_TRANSITION (element)) {
```

```
    case GST_STATE_NULL_TO_READY:
      if (!gst_my_filter_allocate_memory (filter))
        return GST_STATE_FAILURE;
      break;
    case GST_STATE_READY_TO_NULL:
      gst_my_filter_free_memory (filter);
      break;
    default:
      break;
  }

  if (GST_ELEMENT_CLASS (parent_class)->change_state)
    return GST_ELEMENT_CLASS (parent_class)->change_state (element);

  return GST_STATE_SUCCESS;
}
```

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));

  /* 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));
[..]
}
```

Chapter 8. Signals

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 for all internals about signals.

Chapter 9. Building a Test Application

Often, you will want to test your newly written plugin in an as small setting as possible. Ususally, `gst-launch` is a good first step at testing a plugin. 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>

gint
main (gint   arcg,
      gchar *argv[])
{
  GstElement *pipeline, *filesrc, *decoder, *filter, *sink;

  /* initialization */
  gst_init (&argc, &argv);

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

  filesrc  = gst_element_factory_make ("filesrc", "my_filesource");
  decoder  = gst_element_factory_make ("mad", "my_decoder");
  filter   = gst_element_factory_make ("my_filter", "my_filter");
  sink     = gst_element_factory_make ("osssink", "audiosink");

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

  /* link everything together */
  gst_element_link_many (filesrc, decoder, filter, sink, NULL);
  gst_bin_add_many (GST_BIN (pipeline), filesrc, decoder, filter, sink, NULL);
```

```
  /* run */
  gst_element_set_state (pipeline, GST_STATE_PLAYING);
  while (gst_bin_iterate (GST_BIN (pipeline)));

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

  return 0;
}
```

Notes

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

Chapter 10. Creating a Filter with a Filter Factory

A plan for the future is to create a FilterFactory, to make the process of making a new filter a simple process of specifying a few details, and writing a small amount of code to perform the actual data processing. Ideally, a FilterFactory would perform the tasks of boilerplate creation, code functionality implementation, and filter registration.

Unfortunately, this has not yet been implemented. Even when someone eventually does write a FilterFactory, this element will not be able to cover all the possibilities available for filter writing. Thus, some plugins will always need to be manually coded and registered.

Here is a rough outline of what is planned: You run the FilterFactory and give the factory a list of appropriate function pointers and data structures to define a filter. With a reasonable measure of preprocessor magic, you just need to provide a name for the filter and definitions of the functions and data structures desired. Then you call a macro from within plugin_init() that registers the new filter. All the fluff that goes into the definition of a filter is thus be hidden from view.

Chapter 11. How scheduling works

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 to 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. `GStreamer` currently provides two schedulers: a *basic* scheduler and an *optimal* scheduler. As the name says, the basic scheduler ("basic") is an unoptimized, but very complete and simple scheduler. The optimal scheduler ("opt"), on the other hand, is optimized for media processing, but therefore also more complex.

Note that schedulers only operate on one thread. If your pipeline contains multiple threads, each thread will run with a separate scheduler. That is the reason why two elements running in different threads need a queue-like element (a `DECOUPLED` element) in between them.

The Basic Scheduler

The *basic* scheduler assumes that each element is its own process. We don't use UNIX processes or POSIX threads for this, however; instead, we use so-called *co-threads*. Co-threads are threads that run besides each other, but only one is active at a time. The advantage of co-threads over normal threads is that they're lightweight. The disadvantage is that UNIX or POSIX do not provide such a thing, so we need to include our own co-threads stack for this to run.

The task of the scheduler here is to control which co-thread runs at what time. A well-written scheduler based on co-threads will let an element run until it outputs one piece of data. Upon pushing one piece of data to the next element, it will let the next element run, and so on. Whenever a running element requires data from the previous element, the scheduler will switch to that previous element and run that element until it has provided data for use in the next element.

This method of running elements as needed has the disadvantage that a lot of data will often be queued in between two elements, as the one element has provided data but the other element hasn't actually used it yet. These storages of in-between-data are called *bufpens*, and they can be visualized as a light "queue".

Note that since every element runs in its own (co-)thread, this scheduler is rather heavy on your system for larger pipelines.

The Optimal Scheduler

The *optimal* scheduler takes advantage of the fact that several elements can be linked together in one thread, with one element controlling the other. This works as follows: in a series of chain-based elements, each element has a function that accepts one piece of data, and it calls a function that provides one piece of data to the next element. The optimal scheduler will make sure that the `gst_pad_push ()` function of the first element *directly* calls the chain-function of the second element. This significantly decreases the latency in a pipeline. It takes similar advantage of other possibilities of short-cutting the data path from one element to the next.

The disadvantage of the optimal scheduler is that it is not fully implemented. Also it is badly documented; for most developers, the opt scheduler is one big black box. Features that are not implemented include pad-unlinking within a group while running, pad-selecting (i.e. waiting for data to arrive on a list of pads), and it can't really cope with multi-input/-output elements (with the elements linked to each of these in-/outputs running in the same thread) right now.

Some of our developers are intending to write a new scheduler, similar to the optimal scheduler (but better documented and more completely implemented).

Chapter 12. How a loopfunc works

A `_loop ()` function is a function that is called by the scheduler, but without providing data to the element. Instead, the element will become responsible for acquiring its own data, and it will still be responsible of sending data over to its source pads. This method noticeably complicates scheduling; you should only write loop-based elements when you need to. Normally, chain-based elements are preferred. Examples of elements that *have* to be loop-based are elements with multiple sink pads. Since the scheduler will push data into the pads as it comes (and this might not be synchronous), you will easily get ascynronous data on both pads, which means that the data that arrives on the first pad has a different display timestamp then the data arriving on the second pad at the same time. To get over these issues, you should write such elements in a loop-based form. Other elements that are *easier* to write in a loop-based form than in a chain-based form are demuxers and parsers. It is not required to write such elements in a loop-based form, though.

Below is an example of the easiest loop-function that one can write:

```
static void gst_my_filter_loopfunc (GstElement *element);

static void
gst_my_filter_init (GstMyFilter *filter)
{
[..]
  gst_element_set_loopfunc (GST_ELEMENT (filter), gst_my_filter_loopfunc);
[..]
}

static void
gst_my_filter_loopfunc (GstElement *element)
{
  GstMyFilter *filter = GST_MY_FILTER (element);
  GstData *data;

  /* acquire data */
  data = gst_pad_pull (filter->sinkpad);

  /* send data */
  gst_pad_push (filter->srcpad, data);
}
```

Obviously, this specific example has no single advantage over a chain-based element, so you should never write such elements. However, it's a good introduction to the concept.

Multi-Input Elements

Elements with multiple sink pads need to take manual control over their input to assure that the input is synchronized. The following example code could (should) be used in an aggregator, i.e. an element that takes input from multiple streams and sends it out intermangled. Not really useful in practice, but a good example, again.

```
typedef struct _GstMyFilterInputContext {
  gboolean   eos;
  GstBuffer *lastbuf;
} GstMyFilterInputContext;

[..]

static void
```

```
gst_my_filter_init (GstMyFilter *filter)
{
  GstElementClass *klass = GST_ELEMENT_GET_CLASS (filter);
  GstMyFilterInputContext *context;

  filter->sinkpad1 = gst_pad_new_from_template (
 gst_element_class_get_pad_template (klass, "sink"), "sink_1");
  context = g_new0 (GstMyFilterInputContext, 1);
  gst_pad_set_private_data (filter->sinkpad1, context);
[..]
  filter->sinkpad2 = gst_pad_new_from_template (
 gst_element_class_get_pad_template (klass, "sink"), "sink_2");
  context = g_new0 (GstMyFilterInputContext, 1);
  gst_pad_set_private_data (filter->sinkpad2, context);
[..]
  gst_element_set_loopfunc (GST_ELEMENT (filter),
       gst_my_filter_loopfunc);
}

[..]

static void
gst_my_filter_loopfunc (GstElement *element)
{
  GstMyFilter *filter = GST_MY_FILTER (element);
  GList *padlist;
  GstMyFilterInputContext *first_context = NULL;

  /* Go over each sink pad, update the cache if needed, handle EOS
   * or non-responding streams and see which data we should handle
   * next. */
  for (padlist = gst_element_get_padlist (element);
       padlist != NULL; padlist = g_list_next (padlist)) {
    GstPad *pad = GST_PAD (padlist->data);
    GstMyFilterInputContext *context = gst_pad_get_private_data (pad);

    if (GST_PAD_IS_SRC (pad))
      continue;

    while (GST_PAD_IS_USABLE (pad) &&
           !context->eos && !context->lastbuf) {
      GstData *data = gst_pad_pull (pad);

      if (GST_IS_EVENT (data)) {
        /* We handle events immediately */
        GstEvent *event = GST_EVENT (data);

        switch (GST_EVENT_TYPE (event)) {
          case GST_EVENT_EOS:
            context->eos = TRUE;
            gst_event_unref (event);
            break;
          case GST_EVENT_DISCONTINUOUS:
            g_warning ("HELP! How do I handle this?");
            /* fall-through */
          default:
            gst_pad_event_default (pad, event);
            break;
        }
      } else {
        /* We store the buffer to handle synchronization below */
        context->lastbuf = GST_BUFFER (data);
      }
    }

    /* synchronize streams by always using the earliest buffer */
```

```
    if (context->lastbuf) {
      if (!first_context) {
        first_context = context;
      } else {
        if (GST_BUFFER_TIMESTAMP (context->lastbuf) <
  GST_BUFFER_TIMESTAMP (first_context->lastbuf))
          first_context = context;
      }
    }
  }

  /* If we handle no data at all, we're at the end-of-stream, so
   * we should signal EOS. */
  if (!first_context) {
    gst_pad_push (filter->srcpad, GST_DATA (gst_event_new (GST_EVENT_EOS)));
    gst_element_set_eos (element);
    return;
  }

  /* So we do have data! Let's forward that to our source pad. */
  gst_pad_push (filter->srcpad, GST_DATA (first_context->lastbuf));
  first_context->lastbuf = NULL;
}
```

Note that a loop-function is allowed to return. Better yet, a loop function *has to* return so the scheduler can let other elements run (this is particularly true for the optimal scheduler). Whenever the scheduler feels right, it will call the loop-function of the element again.

The Bytestream Object

A second type of elements that wants to be loop-based, are the so-called bytestream-elements. Until now, we've only dealt with elements that receive of pull full buffers of a random size from other elements. Often, however, it is wanted to have control over the stream at a byte-level, such as in stream parsers or demuxers. It is possible to manually pull buffers and merge them until a certain size; it is easier, however, to use bytestream, which wraps this behaviour.

Bytestream-using elements are ususally stream parsers or demuxers. For now, we will take a parser as an example. Demuxers require some more magic that will be dealt with later in this guide: Chapter 14. The goal of this parser will be to parse a text-file and to push each line of text as a separate buffer over its source pad.

```
static void
gst_my_filter_loopfunc (GstElement *element)
{
  GstMyFilter *filter = GST_MY_FILTER (element);
  gint n, num;
  guint8 *data;

  for (n = 0; ; n++) {
    num = gst_bytestream_peek_bytes (filter->bs, &data, n + 1);
    if (num != n + 1) {
      GstEvent *event = NULL;
      guint remaining;

      gst_bytestream_get_status (filter->bs, &remaining, &event);
      if (event) {
        if (GST_EVENT_TYPE (event) == GST_EVENT_EOS)) {
          /* end-of-file */
          gst_pad_push (filter->srcpad, GST_DATA (event));
```

```
            gst_element_set_eos (element);

            return;
          }
          gst_event_unref (event);
        }

        /* failed to read - throw error and bail out */
        gst_element_error (element, STREAM, READ, (NULL), (NULL));

        return;
      }

      /* check if the last character is a newline */
      if (data[n] == '\n') {
        GstBuffer *buf = gst_buffer_new_and_alloc (n + 1);

        /* read the line of text without newline - then flush the newline */
        gst_bytestream_peek_data (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);
        g_print ("Pushing '%s'\n", GST_BUFFER_DATA (buf));
        gst_pad_push (filter->srcpad, GST_DATA (buf));

        return;
      }
    }
  }

  static void
  gst_my_filter_change_state (GstElement *element)
  {
    GstMyFilter *filter = GST_MY_FILTER (element);

    switch (GST_STATE_TRANSITION (element)) {
      case GST_STATE_READY_TO_PAUSED:
        filter->bs = gst_bytestream_new (filter->sinkpad);
        break;
      case GST_STATE_PAUSED_TO_READY:
        gst_bytestream_destroy (filter->bs);
        break;
      default:
        break;
    }

    if (GST_ELEMENT_CLASS (parent_class)->change_state)
      return GST_ELEMENT_CLASS (parent_class)->change_state (element);

    return GST_STATE_SUCCESS;
  }
```

In the above example, you'll notice how bytestream handles buffering of data for you. The result is that you can handle the same data multiple times. Event handling in bytestream is currently sort of *wacky*, but it works quite well. The one big disadvantage of bytestream is that it *requires* the element to be loop-based. Long-term, we hope to have a chain-based usable version of bytestream, too.

Adding a second output

WRITEME

Modifying the test application

WRITEME

Chapter 13. 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 neccessary 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 ususally lives in `gst-plugins/gst/typefind/gsttypefindfunctions.c`, unless there's a good reason (like library dependencies) to put it elsewhere. The reason for this centralization is to decreate 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 will be discussed in great detail in the chapter called Writing an Autoplugger.

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 13-1. 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.					
audio/x-raw-int	Unstructured and uncompressed raw fixed-integer audio data.	endianness	integer	G_BIG_ENDIAN (1234) or G_LITTLE_ENDIAN (4321)	The order of bytes in a sample. The value G_LITTLE_ENDIAN (4321) means "little-endian" (byte-order is "least significant byte first"). The value G_BIG_ENDIAN (1234) 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 (1234) or G_LITTLE_ENDIAN (4321)	The order of bytes in a sample. The value G_LITTLE_ENDIAN (4321) means "little-endian" (byte-order is "least significant byte first"). The value G_BIG_ENDIAN (1234) 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.
		buffer-frames	integer	greater than 0	The number of frames per buffer. The reason for this property is that the element does not need to reuse buffers or use data spanned over multiple buffers, so this property - when used rightly - will decrease latency. Note that some people think that this property is very ugly, whereas others think it is vital for the use of `GStreamer` in professional audio applications.
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	"quick-time", "wav", "mi-crosoft" 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 (Quick-time) stream.				There are currently no specific properties defined or needed for this type.
audio/x-dv	Audio as provided in a Digital Video 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-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.
audio/mpeg	Audio data compressed using the MPEG audio encoding scehem.	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.

Mime Type	Description	Property	Property Type	Property Values	Property Description
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.
audio/x-wma	Windows Media Audio	wmaversion	integer	1,2 or 3	The version of the WMA codec used to encode the stream.

Table 13-2. 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
		height	integer	greater than 0	The height of the video image
		framerate	double	greater than 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”).
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.

Mime Type	Description	Property	Property Type	Property Values	Property Description
		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 (1234) or G_LITTLE_ENDIAN (4321)	The order of bytes in a sample. The value G_LITTLE_ENDIAN (4321) means "little-endian" (byte-order is "least significant byte first"). The value G_BIG_ENDIAN (1234) 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.
		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.				There are currently no specific properties defined or needed for this type.
video/x-h264	H-264 video.				There are currently no specific properties defined or needed for this type.
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.
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).

Mime Type	Description	Property	Property Type	Property Values	Property Description
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	TRUE or 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 (oldish 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.
		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.

Mime Type	Description	Property	Property Type	Property Values	Property Description
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/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.

Table 13-3. 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.
video/x-matroska	Matroska.				There are currently no specific properties defined or needed for this type.

Mime Type	Description	Property	Property Type	Property Values	Property Description
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.
video/x-pn-realvideo	Digital Video.	systemstream	boolean	TRUE	Indicates that this is a container system stream rather than an elementary video stream.
audio/x-wav	WAV.				There are currently no specific properties defined or needed for this type.

Table 13-4. Table of Subtitle Types

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

Table 13-5. Table of Other Types

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

Notes

1. http://www.isi.edu/in-notes/iana/assignments/media-types/media-types

Chapter 14. 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 chapted 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_element_set_private_data (pad, context);

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

  gst_element_add_pad (element, pad);

  return pad;
}
```

The `_loop ()` function is the same as the one given previously in Multi-Input Elements.

Chapter 15. Clocking

WRITEME

Chapter 16. 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 expose these parameters as Dynamic Parameters which can be manipulated by any Dynamic Parameters aware application.

Throughout this section, the term *dparams* will be used as an abbreviation for "Dynamic Parameters".

Comparing Dynamic Parameters with GObject Properties

Your first exposure to dparams may be to convert an existing element from using object properties to using dparams. The following table gives an overview of the difference between these approaches. The significance of these differences should become apparent later on.

	Object Properties	**Dynamic Parameters**
Parameter definition	Class level at compile time	Any level at run time
Getting and setting	Implemented by element subclass as functions	Handled entirely by dparams subsystem
Extra objects required	None - all functionality is derived from base GObject	Element needs to create and store a `GstDParamManager` at object creation
Frequency and resolution of updates	Object properties will only be updated between calls to _get, _chain or _loop	dparams can be updated at any rate independant of calls to _get, _chain or _loop up to sample-level accuracy

Getting Started

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

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

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

```
static gboolean
plugin_init (GModule *module, GstPlugin *plugin)
{
  ...

  /* load dparam support library */
  if (!gst_library_load ("gstcontrol"))
  {
    gst_info ("example: could not load support library: 'gstcontrol'\n");
    return FALSE;
  }

  ...
}
```

You need to store an instance of `GstDParamManager` in your element's struct:

```
struct _GstExample {
  GstElement element;
  ...

  GstDParamManager *dpman;

  ...
};
```

The `GstDParamManager` can be initialised in your element's init function:

```
static void
gst_example_init (GstExample *example)
{
  ...

  example->dpman = gst_dpman_new ("example_dpman", GST_ELEMENT(example));

  ...
}
```

Defining Parameter Specificiations

You can define the dparams you need anywhere within your element but will usually need to do so in only a couple of places:

- In the element `init` function, just after the call to `gst_dpman_new`
- Whenever a new pad is created so that parameters can affect data going into or out of a specific pad. An example of this would be a mixer element where a seperate volume parameter is needed on every pad.

There are three different ways the dparams subsystem can pass parameters into your element. Which one you use will depend on how that parameter is used within your element. Each of these methods has its own function to define a required dparam:

- `gst_dpman_add_required_dparam_direct`
- `gst_dpman_add_required_dparam_callback`
- `gst_dpman_add_required_dparam_array`

These functions will return TRUE if the required dparam was added successfully.

The following function will be used as an example.

```
gboolean
gst_dpman_add_required_dparam_direct (GstDParamManager *dpman,
                                      GParamSpec *param_spec,
                                      gboolean is_log,
                                      gboolean is_rate,
                                      gpointer update_data)
```

The common parameters to these functions are:

- `GstDParamManager *dpman` the element's dparam manager
- `GParamSpec *param_spec` the param spec which defines the required dparam
- `gboolean is_log` whether this dparam value should be interpreted on a log scale (such as a frequency or a decibel value)
- `gboolean is_rate` whether this dparam value is a proportion of the sample rate. For example with a sample rate of 44100, 0.5 would be 22050 Hz and 0.25 would be 11025 Hz.

Direct Method

This method is the simplest and has the lowest overhead for parameters which change less frequently than the sample rate. First you need somewhere to store the parameter - this will usually be in your element's stuct.

```
struct _GstExample {
  GstElement element;
  ...

  GstDParamManager *dpman;
  gfloat volume;
  ...
};
```

Then to define the required dparam just call `gst_dpman_add_required_dparam_direct` and pass in the location of the parameter to change. In this case the location is `&(example->volume)`.

```
gst_dpman_add_required_dparam_direct (
  example->dpman,
  g_param_spec_float("volume","Volume","Volume of the audio",
                     0.0, 1.0, 0.8, G_PARAM_READWRITE),
  FALSE,
  FALSE,
  &(example->volume)
);
```

You can now use `example->volume` anywhere in your element knowing that it will always contain the correct value to use.

Callback Method

This should be used if the you have other values to calculate whenever a parameter changes. If you used the direct method you wouldn't know if a parameter had changed so you would have to recalculate the other values every time you needed them. By using the callback method, other values only have to be recalculated when the dparam value actually changes.

The following code illustrates an instance where you might want to use the callback method. If you had a volume dparam which was represented by a gfloat number, your element may only deal with integer arithmatic. The callback could be used to calculate the integer scaler when the volume changes. First you will need somewhere to store these values.

```
struct _GstExample {
  GstElement element;
  ...
```

```
  GstDParamManager *dpman;
  gfloat volume_f;
  gint   volume_i;
  ...
};
```

When the required dparam is defined, the callback function `gst_example_update_volume` and some user data (which in this case is our element instance) is passed in to the call to `gst_dpman_add_required_dparam_callback`.

```
gst_dpman_add_required_dparam_callback (
  example->dpman,
  g_param_spec_float("volume","Volume","Volume of the audio",
                     0.0, 1.0, 0.8, G_PARAM_READWRITE),
  FALSE,
  FALSE,
  gst_example_update_volume,
  example
);
```

The callback function needs to conform to this signiture

```
typedef void (*GstDPMUpdateFunction) (GValue *value, gpointer data);
```

In our example the callback function looks like this

```
static void
gst_example_update_volume(GValue *value, gpointer data)
{
  GstExample *example = (GstExample*)data;
  g_return_if_fail(GST_IS_EXAMPLE(example));

  example->volume_f = g_value_get_float(value);
  example->volume_i = example->volume_f * 8192;
}
```

Now `example->volume_i` can be used elsewhere and it will always contain the correct value.

Array Method

This method is quite different from the other two. It could be thought of as a specialised method which should only be used if you need the advantages that it provides. Instead of giving the element a single value it provides an array of values where each item in the array corresponds to a sample of audio in your buffer. There are a couple of reasons why this might be useful.

- Certain optimisations may be possible since you can iterate over your dparams array and your buffer data together.
- Some dparams may be able to interpolate changing values at the sample rate. This would allow the array to contain very smoothly changing values which may be required for the stability and quality of some DSP algorithms.

The array method is currently the least mature of the three methods and is not yet ready to be used in elements, but plugin writers should be aware of its existance for the future.

The Data Processing Loop

This is the most critical aspect of the dparams subsystem as it relates to elements. In a traditional audio processing loop, a `for` loop will usually iterate over each sample in the buffer, processing one sample at a time until the buffer is finished. A simplified loop with no error checking might look something like this.

```
static void
example_chain (GstPad *pad, GstBuffer *buf)
{
  ...
  gfloat *float_data;
  int j;
  GstExample *example = GST_EXAMPLE(GST_OBJECT_PARENT (pad));
  int num_samples = GST_BUFFER_SIZE(buf)/sizeof(gfloat);
  float_data = (gfloat *)GST_BUFFER_DATA(buf);
  ...
  for (j = 0; j < num_samples; j++) {
    float_data[j] *= example->volume;
  }
  ...
}
```

To make this dparams aware, a couple of changes are needed.

```
static void
example_chain (GstPad *pad, GstBuffer *buf)
{
  ...
  int j = 0;
  GstExample *example = GST_EXAMPLE(GST_OBJECT_PARENT (pad));
  int num_samples = GST_BUFFER_SIZE(buf)/sizeof(gfloat);
  gfloat *float_data = (gfloat *)GST_BUFFER_DATA(buf);
  int frame_countdown = GST_DPMAN_PREPROCESS(example->dpman, num_samples, GST_BUFFER_TI
  ...
  while (GST_DPMAN_PROCESS_COUNTDOWN(example->dpman, frame_countdown, j)) {
    float_data[j++] *= example->volume;
  }
  ...
}
```

The biggest changes here are 2 new macros, `GST_DPMAN_PREPROCESS` and `GST_DPMAN_PROCESS_COUNTDOWN`. You will also notice that the for loop has become a while loop. `GST_DPMAN_PROCESS_COUNTDOWN` is called as the condition for the while loop so that any required dparams can be updated in the middle of a buffer if required. This is because one of the required behaviours of dparams is that they can be *sample accurate*. This means that parameters change at the exact timestamp that they are supposed to - not after the buffer has finished being processed.

It may be alarming to see a macro as the condition for a while loop, but it is actually very efficient. The macro expands to the following.

```
#define GST_DPMAN_PROCESS_COUNTDOWN(dpman, frame_countdown, frame_count) \
    (frame_countdown-- || \
    (frame_countdown = GST_DPMAN_PROCESS(dpman, frame_count)))
```

So as long as `frame_countdown` is greater than 0, `GST_DPMAN_PROCESS` will not be called at all. Also in many cases, `GST_DPMAN_PROCESS` will do nothing and simply return 0, meaning that there is no more data in the buffer to process.

The macro `GST_DPMAN_PREPROCESS` will do the following:

- Update any dparams which are due to be updated.
- Calculate how many samples should be processed before the next required update
- Return the number of samples until next update, or the number of samples in the buffer - whichever is less.

In fact `GST_DPMAN_PROCESS` may do the same things as `GST_DPMAN_PREPROCESS` depending on the mode that the dparam manager is running in (see below).

DParam Manager Modes

A brief explanation of dparam manager modes might be useful here even though it doesn't generally affect the way your element is written. There are different ways media applications will be used which require that an element's parameters be updated in differently. These include:

- *Timelined* - all parameter changes are known in advance before the pipeline is run.
- *Realtime low-latency* - Nothing is known ahead of time about when a parameter might change. Changes need to be propagated to the element as soon as possible.

When a dparam-aware application gets the dparam manager for an element, the first thing it will do is set the dparam manager mode. Current modes are `"synchronous"` and `"asynchronous"`.

If you are in a realtime low-latency situation then the `"synchronous"` mode is appropriate. During `GST_DPMAN_PREPROCESS` this mode will poll all dparams for required updates and propagate them. `GST_DPMAN_PROCESS` will do nothing in this mode. To then achieve the desired latency, the size of the buffers needs to be reduced so that the dparams will be polled for updates at the desired frequency.

In a timelined situation, the `"asynchronous"` mode will be required. This mode hasn't actually been implemented yet but will be described anyway. The `GST_DPMAN_PREPROCESS` call will precalculate when and how often each dparam needs to update for the duration of the current buffer. From then on `GST_DPMAN_PROCESS` will propagate the calculated updates each time it is called until end of the buffer. If the application is rendering to disk in non-realtime, the render could be sped up by increasing the buffer size. In the `"asynchronous"` mode this could be done without affecting the sample accuracy of the parameter updates

DParam Manager Modes

All of the explanation so far has presumed that the buffer contains audio data with many samples. Video should be regarded differently since a video buffer often contains only 1 frame. In this case some of the complexity of dparams isn't required but the other benefits still make it useful for video parameters. If a buffer only contains one frame of video, only a single call to `GST_DPMAN_PREPROCESS` should be required. For more than one frame per buffer, treat it the same as the audio case.

Chapter 17. MIDI

WRITEME

Chapter 18. Interfaces

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

The first disadvantage has to do with 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 stands for a bitrate property is the same as an UI widget that stands for the size of a video, as long as both are of the same `GParamSpec` type. Another problem, related to the one about parameter important, is that things like parameter grouping, function grouping or anything to make parameters coherent, is not really possible.

The second argument against 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 run-time. 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`.

One important note: interfaces do *not* replace properties. Rather, interfaces should be built *next to* properties. There are two important reasons for this. Firstly, properties can be saved in XML files. Secondly, properties can be specified on the commandline (`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`. 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`, 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 */
}
```

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` interface to be implemented by the element. The example below will feature both, so it serves as an example for the `GstImplementsInterface`, too. In the `GstImplementsInterface`, 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 *iface
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` 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 *iface
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 a enumeraiton list are static. Crrently, 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.

Profile Interface

WRITEME

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

Chapter 19. Tagging (Metadata and Streaminfo)

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.

A tag reading element is called `TagGetter` in `GStreamer`. A tag writer is called `TagSetter`. An element supporting both can be used in a tag editor for quick tag changing.

Reading Tags from Streams

The basic object for tags is a `GstTagList`. 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_values ()`. Note that an element probably reads metadata as strings, but values might not necessarily be strings. Be sure to use `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 ()`. The tags should also be part of the datastream, so they should be pushed over all source pads. The function `gst_event_new_tag ()` creates an event from a taglist. This can be pushed over source pads using `gst_pad_push ()`. Simple elements with only one source pad can combine all these steps all-in-one by using the function `gst_element_found_tags_for_pad ()`.

The following example program will parse a file and parse the data as metadata/tags rathen 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_loopfunc (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_loopfunc (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_list (tagsetter)) {
    gst_tag_list_insert (taglist,
    gst_tag_setter_get_list (tagsetter),
    gst_tag_setter_get_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.

Chapter 20. Events: Seeking, Navigation and More

There are many different event types but only 2 ways they can travel accross the pipeline: downstream or upstream. It is very important to understand how both of those 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 dataflow. Depending if your element is loop or chain based you will receive events in your loop/chain function as a GstData with `gst_pad_pull` or directly in the function call arguments. So when receiving dataflow from the sink pad you have to check first if this data chunk is an event. If that's the case you check what kind of event it is to react on relevant ones and then forward others dowstream using `gst_pad_event_default`. Here is an example for both loop and chain based elements.

```
/* Chain based element */
static void
gst_my_filter_chain (GstPad  *pad,
                     GstData *data)
{
  GstMyFilter *filter = GST_MY_FILTER (gst_pad_get_parent (pad));
  ...
  if (GST_IS_EVENT (data)) {
    GstEvent *event = GST_EVENT (data);

    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);
        /* fall-through to default event handling */
      default:
        gst_pad_event_default (pad, event);
        break;
    }
    return;
  }
  ...
}

/* Loop based element */
static void
gst_my_filter_loop (GstElement *element)
{
  GstMyFilter *filter = GST_MY_FILTER (element);
  GstData *data = NULL;

  data = gst_pad_pull (filter->sinkpad);

  if (GST_IS_EVENT (data)) {
    GstEvent *event = GST_EVENT (data);

    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);
        /* fall-through to default event handling */
      default:
        gst_pad_event_default (filter->sinkpad, event);
        break;
    }
    return;
```

```
  }
  ...
}
```

Upstream events

Upstream events are generated by an element somewhere in the pipeline and sent using the `gst_pad_send_event` function. This function simply realizes the pad and call the default event handler of that pad. The default event handler of pads is `gst_pad_event_default` , 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 you need to handle that event locally. For example a navigation event that you will want to convert before sending it upstream.

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.

Here is an example of correct upstream event handling for a plugin that wants to modify navigation events.

```
static gboolean
gst_my_filter_handle_src_event (GstPad   *pad,
    GstEvent *event)
{
  GstMyFilter *filter = GST_MY_FILTER (gst_pad_get_parent (pad));

  switch (GST_EVENT_TYPE (event)) {
    case GST_EVENT_NAVIGATION:
      GstEvent *new_event = gst_event_new (GST_EVENT_NAVIGATION);;
      /* Create a new event based on received one and then send it */
      ...
      gst_event_unref (event);
      return gst_pad_event_default (pad, new_event);
    default:
      /* Falling back to default event handling for that pad */
      return gst_pad_event_default (pad, event);
  }
}
```

All Events Together

In this chapter follows a list of all defined events that are currently being used, plus how they should be used/interpretted. Events are stored in a `GstEvent` structure, which is simply a big C union with the types for each event in it. For the next development cycle, we intend to switch events over to `GstStructure`, but you don't need to worry about that too much for now.

In this chapter, we will discuss the following events:

- End of Stream (EOS)
- Flush
- Stream Discontinuity
- Seek Request
- Stream Filler
- Interruption
- Navigation
- Tag (metadata)

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 forward the event further downstream and set itself to EOS (`gst_element_set_eos ()`). `gst_pad_event_default ()` takes care of all this, so most elements do not need to support this event. Exceptions are elements that explicitely 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 set the pipeline to PLAYING again. N-to-1 elements have been discussed previously in Multi-Input Elements.

The EOS event (`GST_EVENT_EOS`) has no properties, and that makes it one of the simplest events in `GStreamer`. It is created using `gst_event_new (GST_EVENT_EOS);`.

Some elements support the EOS event upstream, too. This signals the element to go into EOS as soon as possible and signal the EOS event forward downstream. This is useful for elements that have no concept of end-of-stream themselves. Examples are TV card sources, audio card sources, etc. This is not (yet) part of the official specifications of this event, though.

Flush

The flush event is being 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.

The flush event is created with `gst_event_new (GST_EVENT_FLUSH);`. Like the EOS event, it has no properties.

Stream Discontinuity

A discontinuity event is sent downstream to indicate a discontinuity in the data stream. This can happen because the application used the seek event to seek to a different position in the stream, but it can also be because a real-time network source temporarily lost the connection. After the connection is restored, the data stream will continue, but not at the same point where it got lost. Therefore, a discontinuity event is being sent downstream, too.

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 discontinuity 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 discontinuity event (in time units, `GST_FORMAT_TIME`) further downstream.

The discontinuity event is created using the function `gst_event_new_discontinuous ()`. It should set a boolean value which indicates if the discontinuity event is sent because of a new media type (this can happen if - during iteration - a new location was set on a network source or on a file source). then, it should give a list of formats and offsets in that format. The list should be terminated by 0 as format.

```
static void
my_filter_some_function (GstMyFilter *filter)
{
  GstEvent *event;
[..]
  event = gst_event_new_discontinuous (FALSE,
           GST_FORMAT_BYTES, 0,
           GST_FORMAT_TIME,  0,
           0);
  gst_pad_push (filter->srcpad, GST_DATA (event));
[..]
}
```

Elements parsing this event can use macros and functions to access the various properties. `GST_EVENT_DISCONT_NEW_MEDIA (event)` checks the new-media boolean value. `gst_event_discont_get_value (event, format, &value)` gets the offset of the new stream position in the specified format. If that format was not specified when creating the event, the function returns FALSE.

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 "units" [a term indicating frames for video, samples for audio, etc.]). Seeking can be done with respect to the end-of-file, start-of-file or current position, and can happen in both upstream and downstream direction. Elements receiving seek events should, depending on the element type, either forward it (filters, decoders), change the format in which the event is given and forward it (demuxers), handle the event by changing the file pointer in their internal stream resource (file sources) or something else.

Seek events are, like discontinuity events, built up using positions in specified formats (time, bytes, units). They are created using the function `gst_event_new_seek ()`, where the first argument is the seek type (indicating with respect to which position [current, end, start] the seek should be applied, and the format in which the new

position is given (time, bytes, units), and an offset which is the requested position in the specified format.

```
static void
my_filter_some_function (GstMyFilter *filter)
{
  GstEvent *event;
[..]
  /* seek to the start of a resource */
  event = gst_event_new_seek (GST_SEEK_SET | GST_FORMAT_BYTES, 0);
  gst_pad_push (filter->srcpad, GST_DATA (event));
[..]
}
```

Elements parsing this event can use macros and functions to access the properties. The seek type can be retrieved using `GST_EVENT_SEEK_TYPE (event)`. This seek type contains both the indicator of with respect to what position the seek should be applied, and the format in which the seek event is given. To get either one of these properties separately, use `GST_EVENT_SEEK_FORMAT (event)` or `GST_EVENT_SEEK_METHOD (event)`. The requested position is available through `GST_EVENT_SEEK_OFFSET (event)`, and is given in the specified format.

Stream Filler

The filler event is, as the name says, a "filler" of the stream which has no special meaning associated with itself. It is used to provide data to downstream elements and should be interpreted as a way of assuring that the normal data flow will continue further downstream. The event is especially intended for real-time MIDI source elements, which only generate data when something *changes*. MIDI decoders will therefore stall if nothing changes for several seconds, and therefore playback will stop. The filler event is sent downstream to assure the MIDI decoder that nothing changed, so that the normal decoding process will continue and playback will, too. Unless you intend to work with MIDI or other control-language-based data types, you don't need this event. You can mostly simply forward it with `gst_pad_event_default ()`.

The stream filler is created using `gst_event_new (GST_EVENT_FILLER);`. It has no properties.

Interruption

The interrupt event is generated by queue elements and sent downstream if a timeout occurs on the stream. The scheduler will use this event to get back in its own main loop and schedule other elements. This prevents deadlocks or a stream stall if no data is generated over a part of the pipeline for a considerable amount of time. The scheduler will process this event internally, so any normal elements do not need to generate or handle this event at all.

The difference between the filler event and the interrupt event is that the filler event is a real part of a pipeline, so it will reach fellow elements, which can use it to "do nothing else than what I used to do". The interrupt event never reaches fellow elements.

The interrupt event (`gst_event_new (GST_EVENT_INTERRUPT);`) has no properties.

Navigation

WRITEME

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 19. 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 ()`. It requires a filled taglist as argument.

Elements parsing this event can use the function `gst_event_tag_get_list (event)` to acquire the taglist that was parsed.

Chapter 21. Writing a Source

Source elements are the start of a data streaming pipeline. Source elements have no sink pads and have one or more source pads. We will focus on single-sourcepad elements here, but the concepts apply equally well to multi-sourcepad elements. This chapter will explain the essentials of source elements, which features it should implement and which it doesn't have to, and how source elements will interact with other elements in a pipeline.

The get()-function

Source elements have the special option of having a `_get ()`-function rather than a `_loop ()`- or `_chain ()`-function. A `_get ()`-function is called by the scheduler every time the next elements needs data. Apart from corner cases, every source element will want to be `_get ()`-based.

```
static GstData * gst_my_source_get (GstPad *pad);

static void
gst_my_source_init (GstMySource *src)
{
[..]
  gst_pad_set_get_function (src->srcpad, gst_my_source_get);
}

static GstData *
gst_my_source_get (GstPad *pad)
{
  GstBuffer *buffer;

  buffer = gst_buffer_new ();
  GST_BUFFER_DATA (buf) = g_strdup ("hello pipeline!");
  GST_BUFFER_SIZE (buf) = strlen (GST_BUFFER_DATA (buf));
  /* terminating '/0' */
  GST_BUFFER_MAZSIZE (buf) = GST_BUFFER_SIZE (buf) + 1;

  return GST_DATA (buffer);
}
```

Events, querying and converting

One of the most important functions of source elements is to implement correct query, convert and event handling functions. Those will continuously describe the current state of the stream. Query functions can be used to get stream properties such as current position and length. This can be used by fellow elements to convert this same value into a different unit, or by appliations to provide information about the length/position of the stream to the user. Conversion functions are used to convert such values from one unit to another. Lastly, events are mostly used to seek to positions inside the stream. Any function is essentially optional, but the element should try to provide as much information as it knows. Note that elements providing an event function should also list their supported events in an `_get_event_mask ()` function. Elements supporting query operations should list the supported operations in a `_get_query_types ()` function. Elements supporting either conversion or query operations should also implement a `_get_formats ()` function.

An example source element could, for example, be an element that continuously generates a wave tone at 44,1 kHz, mono, 16-bit. This element will generate 44100 audio samples per second or 88,2 kB/s. This information can be used to implement such functions:

```
static GstFormat * gst_my_source_format_list (GstPad      *pad);
static GstQueryType * gst_my_source_query_list (GstPad      *pad);

static gboolean  gst_my_source_convert  (GstPad      *pad,
        GstFormat    from_fmt,
        gint64       from_val,
        GstFormat   *to_fmt,
        gint64      *to_val);
static gboolean  gst_my_source_query  (GstPad      *pad,
        GstQueryType type,
        GstFormat   *to_fmt,
        gint64      *to_val);

static void
gst_my_source_init (GstMySource *src)
{
[..]
  gst_pad_set_convert_function (src->srcpad, gst_my_source_convert);
  gst_pad_set_formats_function (src->srcpad, gst_my_source_format_list);
  gst_pad_set_query_function (src->srcpad, gst_my_source_query);
  gst_pad_set_query_type_function (src->srcpad, gst_my_source_query_list);
}

/*
 * This function returns an enumeration of supported GstFormat
 * types in the query() or convert() functions. See gst/gstformat.h
 * for a full list.
 */

static GstFormat *
gst_my_source_format_list (GstPad *pad)
{
  static const GstFormat formats[] = {
    GST_FORMAT_TIME,
    GST_FORMAT_DEFAULT, /* means "audio samples" */
    GST_FORMAT_BYTES,
    0
  };

  return formats;
}

/*
 * This function returns an enumeration of the supported query()
 * operations. Since we generate audio internally, we only provide
 * an indication of how many samples we've played so far. File sources
 * or such elements could also provide GST_QUERY_TOTAL for the total
 * stream length, or other things. See gst/gstquery.h for details.
 */

static GstQueryType *
gst_my_source_query_list (GstPad *pad)
{
  static const GstQueryType query_types[] = {
    GST_QUERY_POSITION,
    0,
  };

  return query_types;
}

/*
 * And below are the logical implementations.
 */

static gboolean
```

```
gst_my_source_convert (GstPad    *pad,
         GstFormat  from_fmt,
         gint64     from_val,
         GstFormat *to_fmt,
         gint64    *to_val)
{
  gboolean res = TRUE;
  GstMySource *src = GST_MY_SOURCE (gst_pad_get_parent (pad));

  switch (from_fmt) {
    case GST_FORMAT_TIME:
      switch (*to_fmt) {
        case GST_FORMAT_TIME:
          /* nothing */
          break;

        case GST_FORMAT_BYTES:
          *to_val = from_val / (GST_SECOND / (44100 * 2));
          break;

        case GST_FORMAT_DEFAULT:
          *to_val = from_val / (GST_SECOND / 44100);
          break;

        default:
          res = FALSE;
          break;
      }
      break;

    case GST_FORMAT_BYTES:
      switch (*to_fmt) {
        case GST_FORMAT_TIME:
          *to_val = from_val * (GST_SECOND / (44100 * 2));
          break;

        case GST_FORMAT_BYTES:
          /* nothing */
          break;

        case GST_FORMAT_DEFAULT:
          *to_val = from_val / 2;
          break;

        default:
          res = FALSE;
          break;
      }
      break;

    case GST_FORMAT_DEFAULT:
      switch (*to_fmt) {
        case GST_FORMAT_TIME:
          *to_val = from_val * (GST_SECOND / 44100);
          break;

        case GST_FORMAT_BYTES:
          *to_val = from_val * 2;
          break;

        case GST_FORMAT_DEFAULT:
          /* nothing */
          break;

        default:
          res = FALSE;
```

```
          break;
      }
      break;

    default:
      res = FALSE;
      break;
  }

  return res;
}

static gboolean
gst_my_source_query (GstPad      *pad,
       GstQueryType type,
       GstFormat   *to_fmt,
       gint64      *to_val)
{
  GstMySource *src = GST_MY_SOURCE (gst_pad_get_parent (pad));
  gboolean res = TRUE;

  switch (type) {
    case GST_QUERY_POSITION:
      res = gst_pad_convert (pad, GST_FORMAT_BYTES, src->total_bytes,
                             to_fmt, to_val);
      break;

    default:
      res = FALSE;
      break;
  }

  return res;
}
```

Be sure to increase src->total_bytes after each call to your `_get ()` function.

Event handling has already been explained previously in the events chapter.

Time, clocking and synchronization

The above example does not provide any timing info, but will suffice for elementary data sources such as a file source or network data source element. Things become slightly more complicated, but still very simple, if we create artificial video or audio data sources, such as a video test image source or an artificial audio source (e.g. `sinesrc` or `silence`). It will become more complicated if we want the element to be a realtime capture source, such as a video4linux source (for reading video frames from a TV card) or an ALSA source (for reading data from soundcards supported by an ALSA-driver). Here, we will need to make the element aware of timing and clocking.

Timestamps can essentially be generated from all the information given above without any difficulty. We could add a very small amount of code to generate perfectly timestamped buffers from our `_get ()`-function:

```
static void
gst_my_source_init (GstMySource *src)
{
[..]
  src->total_bytes = 0;
}

static GstData *
gst_my_source_get (GstPad *pad)
```

```
{
  GstMySource *src = GST_MY_SOURCE (gst_pad_get_parent (pad));
  GstBuffer *buf;
  GstFormat fmt = GST_FORMAT_TIME;
[..]
  GST_BUFFER_DURATION (buf) = GST_BUFFER_SIZE (buf) * (GST_SECOND / (44100 * 2));
  GST_BUFFER_TIMESTAMP (buf) = src->total_bytes * (GST_SECOND / (44100 * 2));
  src->total_bytes += GST_BUFFER_SIZE (buf);

  return GST_DATA (buf);
}

static GstStateReturn
gst_my_source_change_state (GstElement *element)
{
  GstMySource *src = GST_MY_SOURCE (element);

  switch (GST_STATE_PENDING (element)) {
    case GT_STATE_PAUSED_TO_READY:
      src->total_bytes = 0;
      break;

    default:
      break;
  }

  if (GST_ELEMENT_CLASS (parent_class)->change_state)
    return GST_ELEMENT_CLASS (parent_class)->change_state (element);

  return GST_STATE_SUCCESS;
}
```

That wasn't too hard. Now, let's assume real-time elements. Those can either have hardware-timing, in which case we can rely on backends to provide sync for us (in which case you probably want to provide a clock), or we will have to emulate that internally (e.g. to acquire sync in artificial data elements such as `sinesrc`). Let's first look at the second option (software sync). The first option (hardware sync + providing a clock) does not require any special code with respect to timing, and the clocking section already explained how to provide a clock.

```
enum {
  ARG_0,
[..]
  ARG_SYNC,
[..]
};

static void
gst_my_source_class_init (GstMySourceClass *klass)
{
  GObjectClass *object_class = G_OBJECT_CLASS (klass);
[..]
  g_object_class_install_property (object_class, ARG_SYNC,
      g_param_spec_boolean ("sync", "Sync", "Synchronize to clock",
                            FALSE, G_PARAM_READWRITE));
[..]
}

static void
gst_my_source_init (GstMySource *src)
{
[..]
  src->sync = FALSE;
}
```

```
static void
gst_my_source_get (GstPad *pad)
{
  GstMySource *src = GST_MY_SOURCE (gst_pad_get_parent (pad));
  GstBuffer *buf;
[..]
  if (src->sync) {
    /* wait on clock */
    gst_element_wait (GST_ELEMENT (src), GST_BUFFER_TIMESTAMP (buf));
  }

  return GST_DATA (buf);
}

static void
gst_my_source_get_property (GObject    *object,
       guint       prop_id,
       GParamSpec *pspec,
       GValue     *value)
{
  GstMySource *src = GST_MY_SOURCE (gst_pad_get_parent (pad));

  switch (prop_id) {
[..]
    case ARG_SYNC:
      g_value_set_boolean (value, src->sync);
      break;
[..]
  }
}

static void
gst_my_source_get_property (GObject      *object,
       guint         prop_id,
       GParamSpec   *pspec,
       const GValue *value)
{
  GstMySource *src = GST_MY_SOURCE (gst_pad_get_parent (pad));

  switch (prop_id) {
[..]
    case ARG_SYNC:
      src->sync = g_value_get_boolean (value);
      break;
[..]
  }
}
```

Most of this is GObject wrapping code. The actual code to do software-sync (in the `_get ()`-function) is relatively small.

Using special memory

In some cases, it might be useful to use specially allocated memory (e.g. `mmap ()`'ed DMA'able memory) in your buffers, and those will require special handling when they are being dereferenced. For this, `GStreamer` uses the concept of buffer-free functions. Those are special functions pointers that an element can set on buffers that it created itself. The given function will be called when the buffer has been dereferenced, so that the element can clean up or re-use memory internally rather than using the default implementation (which simply calls `g_free ()` on the data pointer).

```
static void
```

```
gst_my_source_buffer_free (GstBuffer *buf)
{
  GstMySource *src = GST_MY_SOURCE (GST_BUFFER_PRIVATE (buf));

  /* do useful things here, like re-queueing the buffer which
   * makes it available for DMA again. The default handler will
   * not free this buffer because of the GST_BUFFER_DONTFREE
   * flag. */
}

static void
gst_my_source_get (GstPad *pad)
{
  GstMySource *src = GST_MY_SOURCE (gst_pad_get_parent (pad));
  GstBuffer *buf;
[..]
  buf = gst_buffer_new ();
  GST_BUFFER_FREE_DATA_FUNC (buf) = gst_my_source_buffer_free;
  GST_BUFFER_PRIVATE (buf) = src;
  GST_BUFFER_FLAG_SET (buf, GST_BUFFER_READONLY | GST_BUFFER_DONTFREE);
[..]

  return GST_DATA (buf);
}
```

Note that this concept should *not* be used to decrease the number of calls made to functions such as `g_malloc ()` inside your element. We have better ways of doing that elsewhere (`GStreamer` core, Glib, Glibc, Linux kernel, etc.).

Chapter 22. Writing a Sink

Sinks are output elements that, opposite to sources, have no source pads and one or more (usually one) sink pad. They can be sound card outputs, disk writers, etc. This chapter will discuss the basic implementation of sink elements.

Data processing, events, synchronization and clocks

Except for corner cases, sink elements will be `_chain ()`-based elements. The concept of such elements has been discussed before in detail, so that will be skipped here. What is very important in sink elements, specifically in real-time audio and video sources (such as `osssink` or `ximagesink`), is event handling in the `_chain ()`-function, because most elements rely on EOS-handling of the sink element, and because A/V synchronization can only be perfect if the element takes this into account.

How to achieve synchronization between streams depends on whether you're a clock-providing or a clock-receiving element. If you're the clock provider, you can do with time whatever you want. Correct handling would mean that you check whether the end of the previous buffer (if any) and the start of the current buffer are the same. If so, there's no gap between the two and you can continue playing right away. If there is a gap, then you'll need to wait for your clock to reach that time. How to do that depends on the element type. In the case of audio output elements, you would output silence for a while. In the case of video, you would show background color. In case of subtitles, show no subtitles at all.

In the case that the provided clock and the received clock are not the same (or in the case where your element provides no clock, which is the same), you simply wait for the clock to reach the timestamp of the current buffer and then you handle the data in it.

A simple data handling function would look like this:

```
static void
gst_my_sink_chain (GstPad  *pad,
     GstData *data)
{
  GstMySink *sink = GST_MY_SINK (gst_pad_get_parent (pad));
  GstBuffer *buf;
  GstClockTime time;

  /* only needed if the element is GST_EVENT_AWARE */
  if (GST_IS_EVENT (data)) {
    GstEvent *event = GST_EVENT (data);

    switch (GST_EVENT_TYPE (event)) {
      case GST_EVENT_EOS:
        [ if your element provides a clock, disable (inactivate) it here ]
        /* pass-through */

      default:
        /* the default handler handles discontinuities, even if your
         * element provides a clock! */
        gst_pad_event_default (pad, event);
        break;
    }

    return;
  }

  buf = GST_BUFFER (data);
  if (GST_BUFFER_TIME_IS_VALID (buf))
    time = GST_BUFFER_TIMESTAMP (buf);
  else
```

```
    time = sink->expected_next_time;

  /* Synchronization - the property is only useful in case the
   * element has the option of not syncing. So it is not useful
   * for hardware-sync (clock-providing) elements. */
  if (sink->sync) {
    /* This check is only needed if you provide a clock. Else,
     * you can always execute the 'else' clause. */
    if (sink->provided_clock == sink->received_clock) {
      /* GST_SECOND / 10 is 0,1 sec, it's an arbitrary value. The
       * casts are needed because else it'll be unsigned and we
       * won't detect negative values. */
      if (llabs ((gint64) sink->expected_next_time - (gint64) time) >
            (GST_SECOND / 10)) {
        /* so are we ahead or behind? */
        if (time > sink->expected_time) {
          /* we need to wait a while... In case of audio, output
           * silence. In case of video, output background color.
           * In case of subtitles, display nothing. */
          [..]
        } else {
          /* Drop data. */
          [..]
        }
      }
    } else {
      /* You could do more sophisticated things here, but we'll
       * keep it simple for the purpose of the example. */
      gst_element_wait (GST_ELEMENT (sink), time);
    }
  }

  /* And now handle the data. */
[..]
}
```

Special memory

Like source elements, sink elements can sometimes provide externally allocated (such as X-provided or DMA'able) memory to elements earlier in the pipeline, and thereby prevent the need for `memcpy ()` for incoming data. We do this by providing a pad-allocate-buffer function.

```
static GstBuffer * gst_my_sink_buffer_allocate (GstPad *pad,
        guint64 offset,
        guint   size);

static void
gst_my_sink_init (GstMySink *sink)
{
[..]
  gst_pad_set_bufferalloc_function (sink->sinkpad,
                                    gst_my_sink_buffer_allocate);
}

static void
gst_my_sink_buffer_free (GstBuffer *buf)
{
  GstMySink *sink = GST_MY_SINK (GST_BUFFER_PRIVATE (buf));

  /* Do whatever is needed here. */
[..]
}
```

```
static GstBuffer *
gst_my_sink_buffer_allocate (GstPad *pad,
        guint64 offset,
        guint   size)
{
  GstBuffer *buf = gst_buffer_new ();

  /* So here it's up to you to wrap your private buffers and
   * return that. */
  GST_BUFFER_FREE_DATA_FUNC (buf) = gst_my_sink_buffer_free;
  GST_BUFFER_PRIVATE (buf) = sink;
  GST_BUFFER_FLAG_SET (buf, GST_BUFFER_DONTFREE);
[..]

  return buf;
}
```

Chapter 23. Writing a 1-to-N Element, Demuxer or Parser

1-to-N elements don't have much special needs or requirements that haven't been discussed already. The most important thing to take care of in 1-to-N elements (things like `tee`-elements or so) is to use proper buffer refcounting and caps negotiation. If those two are taken care of (see the `tee` element if you need example code), there's little that can go wrong.

Demuxers are the 1-to-N elements that need very special care, though. 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.

Demuxer Caps Negotiation

Demuxers will usually contain several elementary streams, and each of those streams' properties will be defined in a stream header at the start of the file (or, rather, stream) that you're parsing. Since those are fixed and there is no possibility to negotiate stream properties with elements earlier in the pipeline, you should always use explicit caps on demuxer source pads. This prevents a whole lot of caps negotiation or re-negotiation errors.

Data processing and downstream events

Data parsing, pulling this into subbuffers and sending that to the source pads of the elementary streams is the one single most important task of demuxers and parsers. Usually, an element will have a `_loop ()` function using the `bytestream` object to read data. Try to have a single point of data reading from the bytestream object. In this single point, do *proper* event handling (in case there is any) and *proper* error handling in case that's needed. Make your element as fault-tolerant as possible, but do not go further than possible.

Parsing versus interpreting

One particular convention that `GStreamer` demuxers follow is that of separation of parsing and interpreting. The reason for this is maintainability, clarity and code reuse. An easy example of this is something like RIFF, which has a chunk header of 4 bytes, then a length indicator of 4 bytes and then the actual data. We write special functions to read one chunk, to peek a chunk ID and all those; that's the *parsing* part of the demuxer. Then, somewhere else, we like to write the main data processing function, which calls this parse function, reads one chunk and then does with the data whatever it needs to do.

Some example code for RIFF-reading to illustrate the above two points:

```
static gboolean
gst_my_demuxer_peek (GstMyDemuxer *demux,
       guint32      *id,
       guint32      *size)
{
  guint8 *data;

  while (gst_bytestream_peek_bytes (demux->bs, &data, 4) != 4) {
    guint32 remaining;
    GstEvent *event;

    gst_bytestream_get_status (demux->bs, &remaining, &event);
```

```
    if (event) {
      GstEventType type = GST_EVENT_TYPE (event);

      /* or maybe custom event handling, up to you - we lose reference! */
      gst_pad_event_default (demux->sinkpad, event);

      if (type == GST_EVENT_EOS)
        return FALSE;
    } else {
      GST_ELEMENT_ERROR (demux, STREAM, READ, (NULL), (NULL));
      return FALSE;
    }
  }

  *id = GUINT32_FROM_LE (((guint32 *) data)[0]);
  *size = GUINT32_FROM_LE (((guint32 *) data)[0]);

  return TRUE;
}

static void
gst_my_demuxer_loop (GstElement *element)
{
  GstMyDemuxer *demux = GST_MY_DEMUXER (element);
  guint32 id, size;

  if (!gst_my_demuxer_peek (demux, &id, &size))
    return;

  switch (id) {
    [.. normal chunk handling ..]
  }
}
```

Reason for this is that event handling is now centralized in one place and the `_loop ()` function is a lot cleaner and more readable. Those are common code practices, but since the mistake of *not* using such common code practices has been made too often, we explicitely mention this here.

Simple seeking and indexes

Sources will generally receive a seek event in the exact supported format by the element. Demuxers, however, can not seek in themselves directly, but need to convert from one unit (e.g. time) to the other (e.g. bytes) and send a new event to its sink pad. Given this, the `_convert ()`-function (or, more general: unit conversion) is the most important function in a demuxer. Some demuxers (AVI, Matroska) and parsers will keep an index of all chunks in a stream, firstly to improve seeking precision and secondly so they won't lose sync. Some other demuxers will seek the stream directly without index (e.g. MPEG, Ogg) - usually based on something like a cumulative bitrate - and then find the closest next chunk from their new position. The best choice depends on the format.

Note that it is recommended for demuxers to implement event, conversion and query handling functions (using time units or so), in addition to the ones (usually in byte units) provided by the pipeline source element.

Chapter 24. Writing a N-to-1 Element or Demuxer

N-to-1 elements have been previously mentioned and discussed in both Chapter 14 and in Chapter 12. The main noteworthy thing about N-to-1 elements is that they should *always*, without any single exception, be `_loop ()`-based. Apart from that, there is not much general that you need to know. We will discuss one special type of N-to-1 elements here, these being muxers. The first two of these sections apply to N-to-1 elements in general, though.

The Data Loop Function

As previously mentioned in Chapter 12, N-to-1 elements generally try to have one buffer from each sink pad and then handle the one with the earliest timestamp. There's some exceptions to this rule, we will come to those later. This only works if all streams actually continuously provide input. There might be cases where this is not true, for example subtitles (there might be no subtitle for a while), overlay images and so forth. For this purpose, there is a `_select ()` function in `GStreamer`. It checks whether input is available on a (list of) pad(s). In this way, you can skip over the pads that are 'non- continuous'.

```
/* Pad selection is currently broken, FIXME some day */
```

Events in the Loop Function

N-to-1 elements using a cache will sometimes receive events, and it is often unclear how to handle those. For example, how do you seek to a frame in an *output* file (and what's the point of it anyway)? So, do discontinuity or seek events make sense, and should you use them?

Discontinuities and flushes

Don't do anything. They specify a discontinuity in the output, and you should continue to playback as you would otherwise. You generally do not need to put a discontinuity in the output stream in muxers; you would have to manually start adapting timestamps of output frames (if appliccable) to match the previous timescale, though. Note that the output data stream should be continuous. For other types of N-to-1-elements, it is generally fine to forward the discontinuity once it has been received from all pads. This depends on the specific element.

Seeks

Depends on the element. Muxers would generally not implement this, because the concept of seeking in an *output* stream at frame level is not very useful. Seeking at byte level can be useful, but that is more generally done *by* muxers *on* sink elements.

End-of-Stream

Speaks for itself.

Negotiation

Most container formats will have a fair amount of issues with changing content on an elementary stream. Therefore, you should not allow caps to be changed once you've

started using data from them. The easiest way to achieve this is by using explicit caps, which have been explained before. However, we're going to use them in a slightly different way then what you're used to, having the core do all the work for us.

The idea is that, as long as the stream/file headers have not been written yet and no data has been processed yet, a stream is allowed to renegotiate. After that point, the caps should be fixed, because we can only use a stream once. Caps may then only change within an allowed range (think MPEG, where changes in FPS are allowed), or sometimes not at all (such as AVI audio). In order to do that, we will, after data retrieval and header writing, set an explicit caps on each sink pad, that is the minimal caps describing the properties of the format that may not change. As an example, for MPEG audio inside an MPEG system stream, this would mean a wide caps of audio/mpeg with mpegversion=1 and layer=[1,2]. For the same audio type in MPEG, though, the samplerate, bitrate, layer and number of channels would become static, too. Since the (request) pads will be removed when the stream ends, the static caps will cease to exist too, then. While the explicit caps exist, the `_link ()`- function will not be called, since the core will do all necessary checks for us. Note that the property of using explicit caps should be added along with the actual explicit caps, not any earlier.

Below here follows the simple example of an AVI muxer's audio caps negotiation. The `_link ()`-function is fairly normal, but the `-Loop ()`-function does some of the tricks mentioned above. There is no `_getcaps ()`- function since the pad template contains all that information already (not shown).

```
static GstPadLinkReturn
gst_avi_mux_audio_link (GstPad        *pad,
   const GstCaps *caps)
{
  GstAviMux *mux = GST_AVI_MUX (gst_pad_get_parent (pad));
  GstStructure *str = gst_caps_get_structure (caps, 0);
  const gchar *mime = gst_structure_get_name (str);

  if (!strcmp (str, "audio/mpeg")) {
    /* get version, make sure it's 1, get layer, make sure it's 1-3,
     * then create the 2-byte audio tag (0x0055) and fill an audio
     * stream structure (strh/strf). */
    [..]
    return GST_PAD_LINK_OK;
  } else if !strcmp (str, "audio/x-raw-int")) {
    /* See above, but now with the raw audio tag (0x0001). */
    [..]
    return GST_PAD_LINK_OK;
  } else [..]
[..]
}

static void
gst_avi_mux_loop (GstElement *element)
{
  GstAviMux *mux = GST_AVI_MUX (element);
[..]
  /* As we get here, we should have written the header if we hadn't done
   * that before yet, and we're supposed to have an internal buffer from
   * each pad, also from the audio one. So here, we check again whether
   * this is the first run and if so, we set static caps. */
  if (mux->first_cycle) {
    const GList *padlist = gst_element_get_pad_list (element);
    GList *item;

    for (item = padlist; item != NULL; item = item->next) {
      GstPad *pad = item->data;
      GstCaps *caps;

      if (!GST_PAD_IS_SINK (pad))
```

```
        continue;

      /* set static caps here */
      if (!strncmp (gst_pad_get_name (pad), "audio_", 6)) {
        /* the strf is the struct you filled in the _link () function. */
        switch (strf->format) {
          case 0x0055: /* mp3 */
            caps = gst_caps_new_simple ("audio/mpeg",
   "mpegversion", G_TYPE_INT, 1,
   "layer",       G_TYPE_INT, 3,
   "bitrate",     G_TYPE_INT, strf->av_bps,
   "rate",        G_TYPE_INT, strf->rate,
   "channels",    G_TYPE_INT, strf->channels,
   NULL);
            break;
          case 0x0001: /* pcm */
            caps = gst_caps_new_simple ("audio/x-raw-int",
     [..]);
            break;
          [..]
        }
      } else if (!strncmp (gst_pad_get_name (pad), "video_", 6)) {
        [..]
      } else {
        g_warning ("oi!");
        continue;
      }

      /* set static caps */
      gst_pad_use_explicit_caps (pad);
      gst_pad_set_explicit_caps (pad, caps);
    }
  }
[..]
  /* Next runs will never be the first again. */
  mux->first_cycle = FALSE;
}
```

Note that there are other ways to achieve that, which might be useful for more complex cases. This will do for the simple cases, though. This method is provided to simplify negotiation and renegotiation in muxers, it is not a complete solution, nor is it a pretty one.

Markup vs. data processing

As we noted on demuxers before, we love common programming paradigms such as clean, lean and mean code. To achieve that in muxers, it's generally a good idea to separate the actual data stream markup from the data processing. To illustrate, here's how AVI muxers should write out RIFF tag chunks:

```
static void
gst_avi_mux_write_chunk (GstAviMux *mux,
    guint32    id,
    GstBuffer *data)
{
  GstBuffer *hdr;

  hdr = gst_buffer_new_and_alloc (8);
  ((guint32 *) GST_BUFFER_DATA (buf))[0] = GUINT32_TO_LE (id);
  ((guint32 *) GST_BUFFER_DATA (buf))[1] = GUINT32_TO_LE (GST_BUFFER_SIZE (data));

  gst_pad_push (mux->srcpad, hdr);
  gst_pad_push (mux->srcpad, data);
```

```
}

static void
gst_avi_mux_loop (GstElement *element)
{
  GstAviMux *mux = GST_AVI_MUX (element);
  GstBuffer *buf;
[..]
  buf = gst_pad_pull (mux->sinkpad[0]);
[..]
  gst_avi_mux_write_chunk (GST_MAKE_FOURCC ('0','0','d','b'), buf);
}
```

In general, try to program clean code, that should cover pretty much everything.

Chapter 25. Writing a N-to-N element

FIXME: write.

Chapter 26. Writing an Autoplugger

FIXME: write.

Chapter 27. Writing a Manager

FIXME: write.

Chapter 28. Things to check when writing an element

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.

Chapter 29. Things to check when writing a filter

Chapter 30. Things to check when writing a source or sink