`GStreamer` Application Development Manual (0.10.23)

Wim Taymans

Steve Baker

Andy Wingo

Ronald S. Bultje

Stefan Kost

`GStreamer` Application Development Manual (0.10.23)
by Wim Taymans, Steve Baker, Andy Wingo, Ronald S. Bultje, and Stefan Kost

Table of Contents

Foreword

`GStreamer` is an extremely powerful and versatile framework for creating streaming media applications. Many of the virtues of the `GStreamer` framework come from its modularity: `GStreamer` can seamlessly incorporate new plugin modules. But because modularity and power often come at a cost of greater complexity, writing new applications is not always easy.

This guide is intended to help you understand the `GStreamer` framework (version 0.10.23) so you can develop applications based on it. The first chapters will focus on development of a simple audio player, with much effort going into helping you understand `GStreamer` concepts. Later chapters will go into more advanced topics related to media playback, but also at other forms of media processing (capture, editing, etc.).

Introduction

Who should read this manual?

This book is about `GStreamer` from an application developer's point of view; it describes how to write a `GStreamer` application using the `GStreamer` libraries and tools. For an explanation about writing plugins, we suggest the Plugin Writers Guide[1].

Also check out the other documentation available on the `GStreamer` web site[2].

Preliminary reading

In order to understand this manual, you need to have a basic understanding of the *C language*.

Since `GStreamer` adheres to the GObject programming model, this guide also assumes that you understand the basics of GObject[3] and glib[4] programming. Especially,

- GObject instantiation
- GObject properties (set/get)
- GObject casting
- GObject referecing/dereferencing
- glib memory management
- glib signals and callbacks
- glib main loop

Structure of this manual

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

Part I in *`GStreamer` Application Development Manual (0.10.23)* gives you an overview of `GStreamer`, it's design principles and foundations.

Part II in *`GStreamer` Application Development Manual (0.10.23)* covers the basics of `GStreamer` application programming. At the end of this part, you should be able to build your own audio player using `GStreamer`

In Part III in *`GStreamer` Application Development Manual (0.10.23)*, we will move on to advanced subjects which make `GStreamer` stand out of its competitors. We will discuss application-pipeline interaction using dynamic parameters and interfaces, we will discuss threading and threaded pipelines, scheduling and clocks (and synchronization). Most of those topics are not just there to introduce you to their API, but primarily to give a deeper insight in solving application programming problems with `GStreamer` and understanding their concepts.

Next, in Part IV in *`GStreamer` Application Development Manual (0.10.23)*, we will go into higher-level programming APIs for `GStreamer`. You don't exactly need to know all the details from the previous parts to understand this, but you will need to understand basic `GStreamer` concepts nevertheless. We will, amongst others, discuss XML, playbin and autopluggers.

Finally in Part V in *`GStreamer` Application Development Manual (0.10.23)*, you will find some random information on integrating with GNOME, KDE, OS X or Windows, some debugging help and general tips to improve and simplify `GStreamer` programming.

Notes

1. http://gstreamer.freedesktop.org/data/doc/gstreamer/head/pwg/html/index.html
2. http://gstreamer.freedesktop.org/documentation/
3. http://library.gnome.org/devel/gobject/stable/
4. http://library.gnome.org/devel/glib/stable/

Chapter 1. What is `GStreamer`?

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

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

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

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

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

Specifically, `GStreamer` provides

- an API for multimedia applications
- a plugin architecture
- a pipeline architecture
- a mechanism for media type handling/negotiation
- over 150 plug-ins
- a set of tools

`GStreamer` plug-ins could be classified into

- protocols handling
- sources: for audio and video (involves protocol plugins)
- formats: parsers, formaters, muxers, demuxers, metadata, subtitles
- codecs: coders and decoders
- filters: converters, mixers, effects, ...

- sinks: for audio and video (involves protocol plugins)

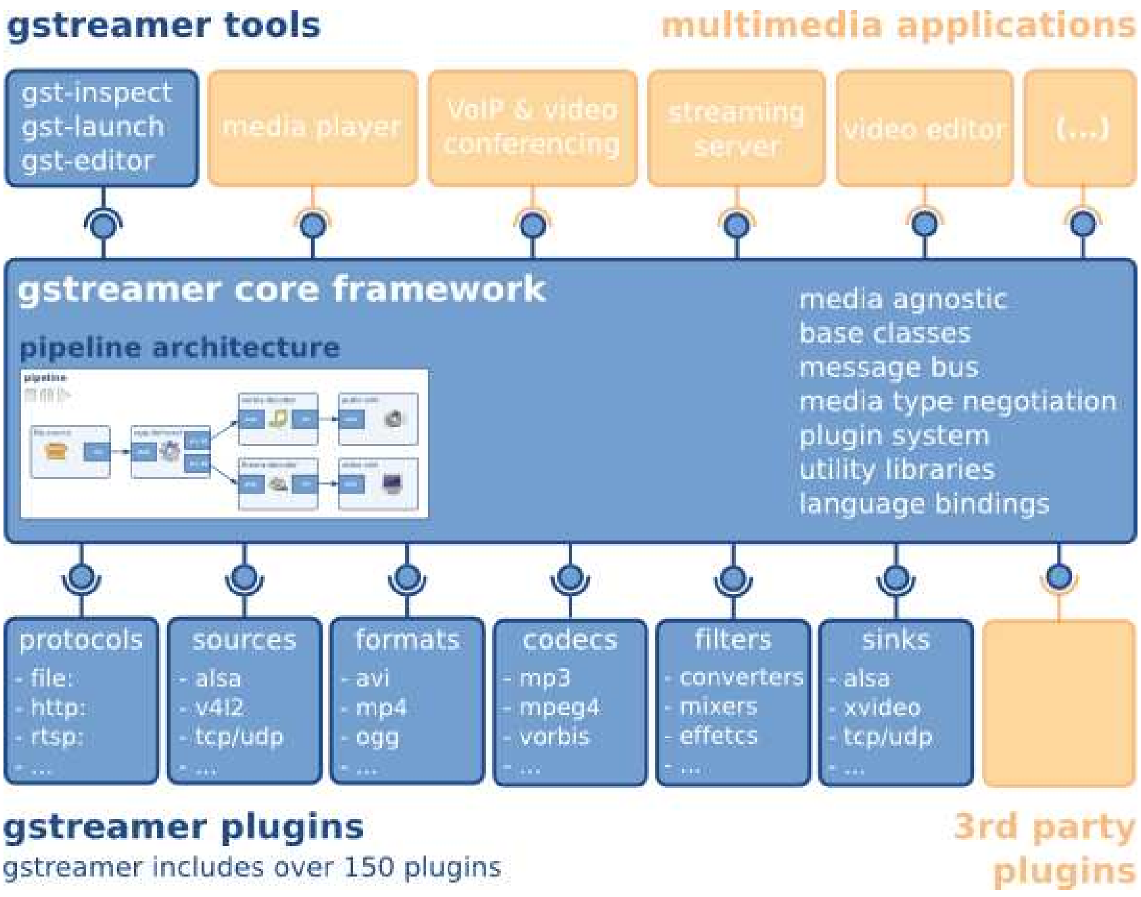

Figure 1-1. Gstreamer overview

`GStreamer` is packaged into

- gstreamer: the core package
- gst-plugins-base: an essential exemplary set of elements
- gst-plugins-good: a set of good-quality plug-ins under LGPL
- gst-plugins-ugly: a set of good-quality plug-ins that might pose distribution problems
- gst-plugins-bad: a set of plug-ins that need more quality
- gst-python: the python bindings

- a few others packages

Chapter 2. Design principles

Clean and powerful

`GStreamer` provides a clean interface to:

- The application programmer who wants to build a media pipeline. The programmer can use an extensive set of powerful tools to create media pipelines without writing a single line of code. Performing complex media manipulations becomes very easy.
- The plugin programmer. Plugin programmers are provided a clean and simple API to create self-contained plugins. An extensive debugging and tracing mechanism has been integrated. GStreamer also comes with an extensive set of real-life plugins that serve as examples too.

Object oriented

`GStreamer` adheres to GObject, the GLib 2.0 object model. A programmer familiar with GLib 2.0 or GTK+ will be comfortable with `GStreamer`.

`GStreamer` uses the mechanism of signals and object properties.

All objects can be queried at runtime for their various properties and capabilities.

`GStreamer` intends to be similar in programming methodology to GTK+. This applies to the object model, ownership of objects, reference counting, etc.

Extensible

All `GStreamer` Objects can be extended using the GObject inheritance methods.

All plugins are loaded dynamically and can be extended and upgraded independently.

Allow binary-only plugins

Plugins are shared libraries that are loaded at runtime. Since all the properties of the plugin can be set using the GObject properties, there is no need (and in fact no way) to have any header files installed for the plugins.

Special care has been taken to make plugins completely self-contained. All relevant aspects of plugins can be queried at run-time.

High performance

High performance is obtained by:

- using GLib's `GSlice` allocator
- extremely light-weight links between plugins. Data can travel the pipeline with minimal overhead. Data passing between plugins only involves a pointer dereference in a typical pipeline.
- providing a mechanism to directly work on the target memory. A plugin can for example directly write to the X server's shared memory space. Buffers can also point to arbitrary memory, such as a sound card's internal hardware buffer.
- refcounting and copy on write minimize usage of memcpy. Sub-buffers efficiently split buffers into manageable pieces.
- dedicated streaming threads, with scheduling handled by the kernel.
- allowing hardware acceleration by using specialized plugins.
- using a plugin registry with the specifications of the plugins so that the plugin loading can be delayed until the plugin is actually used.

Clean core/plugins separation

The core of `GStreamer` is essentially media-agnostic. It only knows about bytes and blocks, and only contains basic elements. The core of `GStreamer` is functional enough to even implement low-level system tools, like cp.

All of the media handling functionality is provided by plugins external to the core. These tell the core how to handle specific types of media.

Provide a framework for codec experimentation

`GStreamer` also wants to be an easy framework where codec developers can experiment with different algorithms, speeding up the development of open and free multimedia codecs like Theora and Vorbis[1].

Notes

1. http://www.xiph.org/ogg/index.html

Chapter 3. Foundations

This chapter of the guide introduces the basic concepts of `GStreamer`. Understanding these concepts will be important in reading any of the rest of this guide, all of them assume understanding of these basic concepts.

Elements

An *element* is the most important class of objects in `GStreamer`. You will usually create a chain of elements linked together and let data flow through this chain of elements. An element has one specific function, which can be the reading of data from a file, decoding of this data or outputting this data to your sound card (or anything else). By chaining together several such elements, you create a *pipeline* that can do a specific task, for example media playback or capture. `GStreamer` ships with a large collection of elements by default, making the development of a large variety of media applications possible. If needed, you can also write new elements. That topic is explained in great deal in the *GStreamer Plugin Writer's Guide*.

Pads

Pads are element's input and output, where you can connect other elements. They used to negotiate links and data flow between elements in `GStreamer`. A pad can be viewed as a "plug" 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. Data types are negotiated between pads using a process called *caps negotiation*. Data types are described as a `GstCaps`.

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. Data usually means buffers (described by the `GstBuffer` [1] object) and events (described by the `GstEvent`[2] object).

Bins and pipelines

A *bin* is a container for a collection of elements. A *pipeline* is a special subtype of a bin that allows execution of all of its contained child elements. Since bins are subclasses of elements themselves, you can mostly control a bin as if it were an element, thereby abstracting away a lot of complexity for your application. You can, for example change state on all elements in a bin by changing the state of that bin itself. Bins also forward bus messages from their contained children (such as error messages, tag messages or EOS messages).

A *pipeline* is a top-level bin. As you set it to PAUSED or PLAYING state, data flow will start and media processing will take place. Once started, pipelines will run in a separate thread until you stop them or the end of the data stream is reached.

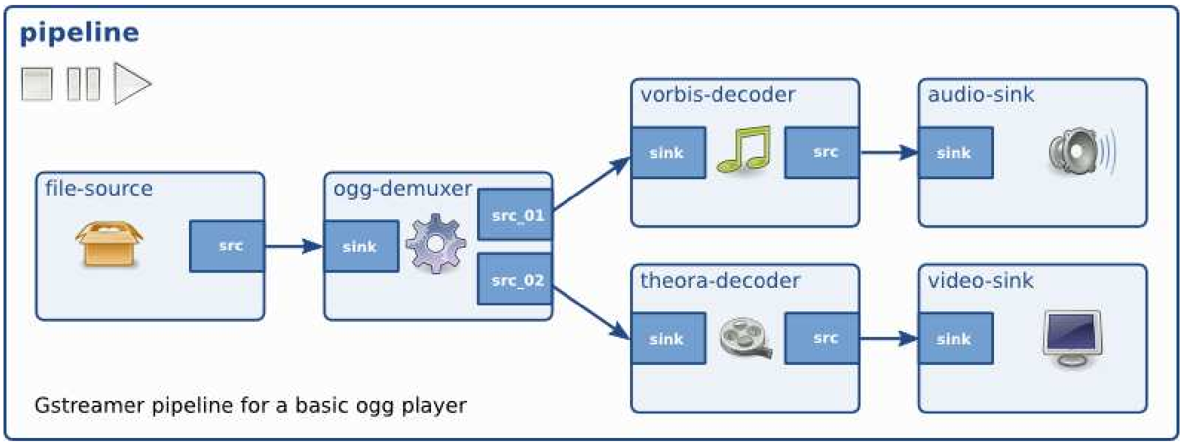

Figure 3-1. `GStreamer` pipeline for a simple ogg player

Notes

1. http://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html//gstreamer-GstBuffer.html
2. http://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html//gstreamer-GstEvent.html

Chapter 4. Initializing `GStreamer`

When writing a `GStreamer` application, you can simply include `gst/gst.h` to get access to the library functions. Besides that, you will also need to intialize the `GStreamer` library.

Simple initialization

Before the `GStreamer` libraries can be used, `gst_init` has to be called from the main application. This call will perform the necessary initialization of the library as well as parse the `GStreamer`-specific command line options.

A typical program [1] would have code to initialize `GStreamer` that looks like this:

Example 4-1. Initializing GStreamer

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

int
main (int   argc,
      char *argv[])
{
  const gchar *nano_str;
  guint major, minor, micro, nano;

  gst_init (&argc, &argv);

  gst_version (&major, &minor, &micro, &nano);

  if (nano == 1)
    nano_str = "(CVS)";
  else if (nano == 2)
    nano_str = "(Prerelease)";
  else
    nano_str = "";

  printf ("This program is linked against GStreamer %d.%d.%d %s\n",
          major, minor, micro, nano_str);

  return 0;
}
```

Use the GST_VERSION_MAJOR, GST_VERSION_MINOR and GST_VERSION_MICRO macros to get the `GStreamer` version you are building against, or use the function `gst_version` to get the version your application is linked against. `GStreamer` currently uses a scheme where versions with the same major and minor versions are API-/ and ABI-compatible.

It is also possible to call the `gst_init` function with two NULL arguments, in which case no command line options will be parsed by `GStreamer`.

The GOption interface

You can also use a GOption table to initialize your own parameters as shown in the next example:

Example 4-2. Initialisation using the GOption interface

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

int
main (int   argc,
      char *argv[])
{
  gboolean silent = FALSE;
  gchar *savefile = NULL;
  GOptionContext *ctx;
  GError *err = NULL;
  GOptionEntry entries[] = {
    { "silent", 's', 0, G_OPTION_ARG_NONE, &silent,
      "do not output status information", NULL },
    { "output", 'o', 0, G_OPTION_ARG_STRING, &savefile,
      "save xml representation of pipeline to FILE and exit", "FILE" },
    { NULL }
  };

  /* we must initialise the threading system before using any
   * other GLib funtion, such as g_option_context_new() */
  if (!g_thread_supported ())
    g_thread_init (NULL);

  ctx = g_option_context_new ("- Your application");
  g_option_context_add_main_entries (ctx, entries, NULL);
  g_option_context_add_group (ctx, gst_init_get_option_group ());
  if (!g_option_context_parse (ctx, &argc, &argv, &err)) {
    g_print ("Failed to initialize: %s\n", err->message);
    g_error_free (err);
    return 1;
  }

  printf ("Run me with --help to see the Application options appended.\n");

  return 0;
}
```

As shown in this fragment, you can use a GOption[2] table to define your application-specific command line options, and pass this table to the GLib initialization function along with the option group returned from the function `gst_init_get_option_group`. Your application options will be parsed in addition to the standard `GStreamer` options.

Notes

1. The code for this example is automatically extracted from the documentation and built under `examples/manual` in the GStreamer tarball.
2. http://developer.gnome.org/doc/API/2.0/glib/glib-Commandline-option-parser.html

Chapter 5. Elements

The most important object in `GStreamer` for the application programmer is the `GstElement`[1] object. An element is the basic building block for a media pipeline. All the different high-level components you will use are derived from `GstElement`. Every decoder, encoder, demuxer, video or audio output is in fact a `GstElement`

What are elements?

For the application programmer, elements are best visualized as black boxes. On the one end, you might put something in, the element does something with it and something else comes out at the other side. For a decoder element, for example, you'd put in encoded data, and the element would output decoded data. In the next chapter (see Pads and capabilities), you will learn more about data input and output in elements, and how you can set that up in your application.

Source elements

Source elements generate data for use by a pipeline, for example reading from disk or from a sound card. Figure 5-1 shows how we will visualise a source element. We always draw a source pad to the right of the element.

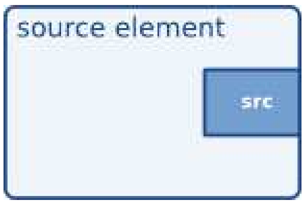

Figure 5-1. Visualisation of a source element

Source elements do not accept data, they only generate data. You can see this in the figure because it only has a source pad (on the right). A source pad can only generate data.

Filters, convertors, demuxers, muxers and codecs

Filters and filter-like elements have both input and outputs pads. They operate on data that they receive on their input (sink) pads, and will provide data on their output (source) pads. Examples of such elements are a volume element (filter), a video scaler (convertor), an Ogg demuxer or a Vorbis decoder.

Filter-like elements can have any number of source or sink pads. A video demuxer, for example, would have one sink pad and several (1-N) source pads, one for each elementary stream contained in the container format. Decoders, on the other hand, will only have one source and sink pads.

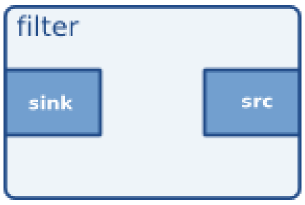

Figure 5-2. Visualisation of a filter element

Figure 5-2 shows how we will visualise a filter-like element. This specific element has one source and one sink element. Sink pads, receiving input data, are depicted at the left of the element; source pads are still on the right.

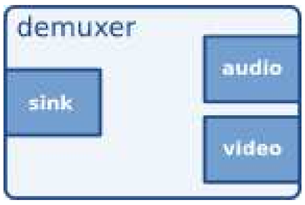

Figure 5-3. Visualisation of a filter element with more than one output pad

Figure 5-3 shows another filter-like element, this one having more than one output (source) pad. An example of one such element could, for example, be an Ogg demuxer for an Ogg stream containing both audio and video. One source pad will contain the elementary video stream, another will contain the elementary audio stream. Demuxers will generally fire signals when a new pad is created. The application programmer can then handle the new elementary stream in the signal handler.

Sink elements

Sink elements are end points in a media pipeline. They accept data but do not produce anything. Disk writing, soundcard playback, and video output would all be implemented by sink elements. Figure 5-4 shows a sink element.

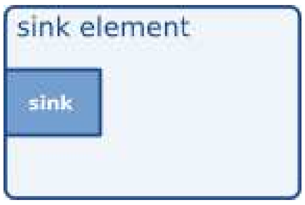

Figure 5-4. Visualisation of a sink element

Creating a `GstElement`

The simplest way to create an element is to use `gst_element_factory_make ()`[2]. This function takes a factory name and an element name for the newly created element. The name of the element is something you can use later on to look up the element in a bin, for example. The name will also be used in debug output. You can pass NULL as the name argument to get a unique, default name.

When you don't need the element anymore, you need to unref it using `gst_object_unref ()`[3]. This decreases the reference count for the element by 1. An element has a refcount of 1 when it gets created. An element gets destroyed completely when the refcount is decreased to 0.

The following example [4] shows how to create an element named *source* from the element factory named *fakesrc*. It checks if the creation succeeded. After checking, it unrefs the element.

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

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

  /* init GStreamer */
  gst_init (&argc, &argv);

  /* create element */
  element = gst_element_factory_make ("fakesrc", "source");
  if (!element) {
    g_print ("Failed to create element of type 'fakesrc'\n");
    return -1;
  }

  gst_object_unref (GST_OBJECT (element));

  return 0;
}
```

`gst_element_factory_make` is actually a shorthand for a combination of two functions. A `GstElement`[5] object is created from a factory. To create the element, you have to get access to a `GstElementFactory`[6] object using a unique factory name. This is done with `gst_element_factory_find ()`[7].

The following code fragment is used to get a factory that can be used to create the *fakesrc* element, a fake data source. The function `gst_element_factory_create ()`[8] will use the element factory to create an element with the given name.

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

int
main (int   argc,
      char *argv[])
{
  GstElementFactory *factory;
  GstElement * element;

  /* init GStreamer */
  gst_init (&argc, &argv);

  /* create element, method #2 */
  factory = gst_element_factory_find ("fakesrc");
  if (!factory) {
    g_print ("Failed to find factory of type ’fakesrc’\n");
    return -1;
  }
  element = gst_element_factory_create (factory, "source");
  if (!element) {
    g_print ("Failed to create element, even though its factory exists!\n");
    return -1;
  }

  gst_object_unref (GST_OBJECT (element));

  return 0;
}
```

Using an element as a `GObject`

A `GstElement`[9] can have several properties which are implemented using standard `GObject` properties. The usual `GObject` methods to query, set and get property values and `GParamSpecs` are therefore supported.

Every `GstElement` inherits at least one property from its parent `GstObject`: the "name" property. This is the name you provide to the functions `gst_element_factory_make ()` or `gst_element_factory_create ()`. You can get and set this property using the functions `gst_object_set_name` and `gst_object_get_name` or use the `GObject` property mechanism as shown below.

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

int
main (int   argc,
      char *argv[])
{
  GstElement *element;
  gchar *name;

  /* init GStreamer */
  gst_init (&argc, &argv);

  /* create element */
  element = gst_element_factory_make ("fakesrc", "source");

  /* get name */
  g_object_get (G_OBJECT (element), "name", &name, NULL);
  g_print ("The name of the element is ’%s’.\n", name);
  g_free (name);

  gst_object_unref (GST_OBJECT (element));

  return 0;
}
```

Most plugins provide additional properties to provide more information about their configuration or to configure the element. **gst-inspect** is a useful tool to query the properties of a particular element, it will also use property introspection to give a short explanation about the function of the property and about the parameter types and ranges it supports. See the appendix for details about **gst-inspect**.

For more information about `GObject` properties we recommend you read the GObject manual[10] and an introduction to The Glib Object system[11].

A `GstElement`[12] also provides various `GObject` signals that can be used as a flexible callback mechanism. Here, too, you can use **gst-inspect** to see which signals a specific element supports. Together, signals and properties are the most basic way in which elements and applications interact.

More about element factories

In the previous section, we briefly introduced the `GstElementFactory`[13] object already as a way to create instances of an element. Element factories, however, are much more than just that. Element factories are the basic types retrieved from the `GStreamer` registry, they describe all plugins and elements that `GStreamer` can create. This means that element factories are useful for automated element instancing, such as what autopluggers do, and for creating lists of available elements, such as what pipeline editing applications (e.g. `GStreamer` Editor[14]) do.

Getting information about an element using a factory

Tools like **gst-inspect** will provide some generic information about an element, such as the person that wrote the plugin, a descriptive name (and a shortname), a rank and a category. The category can be used to get the type of the element that can be created using this element factory. Examples of categories include `Codec/Decoder/Video` (video decoder), `Codec/Encoder/Video` (video encoder), `Source/Video` (a video generator), `Sink/Video` (a video output), and all these exist for audio as well, of course. Then, there's also `Codec/Demuxer` and `Codec/Muxer` and a whole lot more. **gst-inspect** will give a list of all factories, and **gst-inspect <factory-name>** will list all of the above information, and a lot more.

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

int
main (int   argc,
      char *argv[])
{
  GstElementFactory *factory;
  /* init GStreamer */
  gst_init (&argc, &argv);

  /* get factory */
  factory = gst_element_factory_find ("fakesrc");
  if (!factory) {
    g_print ("You don’t have the ’fakesrc’ element installed!\n");
    return -1;
  }

  /* display information */
  g_print ("The ’%s’ element is a member of the category %s.\n"
           "Description: %s\n",
           gst_plugin_feature_get_name (GST_PLUGIN_FEATURE (factory)),
           gst_element_factory_get_klass (factory),
           gst_element_factory_get_description (factory));

  return 0;
}
```

You can use `gst_registry_pool_feature_list (GST_TYPE_ELEMENT_FACTORY)` to get a list of all the element factories that `GStreamer` knows about.

Finding out what pads an element can contain

Perhaps the most powerful feature of element factories is that they contain a full description of the pads that the element can generate, and the capabilities of those pads (in layman words: what types of media can stream over those pads), without actually having to load those plugins into memory. This can be used to provide a codec selection list for encoders, or it can be used for autoplugging purposes for media players. All current `GStreamer`-based media players and autopluggers work this way. We’ll look closer at these features as we learn about `GstPad` and `GstCaps` in the next chapter: Pads and capabilities

Linking elements

By linking a source element with zero or more filter-like elements and finally a sink element, you set up a media pipeline. Data will flow through the elements. This is the basic concept of media handling in `GStreamer`.

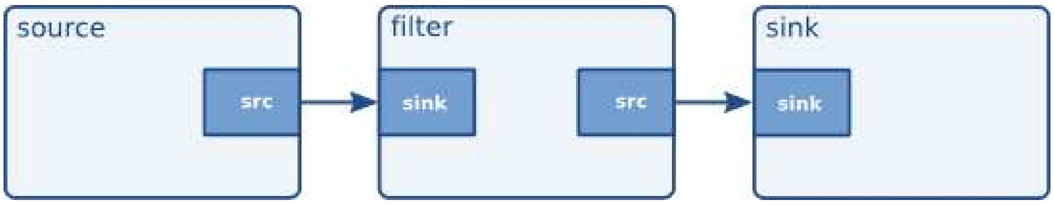

Figure 5-5. Visualisation of three linked elements

By linking these three elements, we have created a very simple chain of elements. The effect of this will be that the output of the source element (“element1”) will be used as input for the filter-like element (“element2”). The filter-like element will do something with the data and send the result to the final sink element (“element3”).

Imagine the above graph as a simple Ogg/Vorbis audio decoder. The source is a disk source which reads the file from disc. The second element is a Ogg/Vorbis audio decoder. The sink element is your soundcard, playing back the decoded audio data. We will use this simple graph to construct an Ogg/Vorbis player later in this manual.

In code, the above graph is written like this:

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

int
main (int   argc,
      char *argv[])
{
  GstElement *pipeline;
  GstElement *source, *filter, *sink;

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

  /* create pipeline */
  pipeline = gst_pipeline_new ("my-pipeline");

  /* create elements */
  source = gst_element_factory_make ("fakesrc", "source");
  filter = gst_element_factory_make ("identity", "filter");
  sink = gst_element_factory_make ("fakesink", "sink");

  /* must add elements to pipeline before linking them */
  gst_bin_add_many (GST_BIN (pipeline), source, filter, sink, NULL);

  /* link */
  if (!gst_element_link_many (source, filter, sink, NULL)) {
    g_warning ("Failed to link elements!");
  }

[..]

}
```

For more specific behaviour, there are also the functions `gst_element_link ()` and `gst_element_link_pads ()`. You can also obtain references to individual pads and link those using various `gst_pad_link_* ()` functions. See the API references for more details.

Important: you must add elements to a bin or pipeline *before* linking them, since adding an element to a bin will disconnect any already existing links. Also, you cannot directly link elements that are not in the same bin or pipeline; if you want to link elements or pads at different hierarchy levels, you will need to use ghost pads (more about ghost pads later).

Element States

After being created, an element will not actually perform any actions yet. You need to change elements state to make it do something. `GStreamer` knows four element states, each with a very specific meaning. Those four states are:

- `GST_STATE_NULL`: this is the default state. This state will deallocate all resources held by the element.
- `GST_STATE_READY`: in the ready state, an element has allocated all of its global resources, that is, resources that can be kept within streams. You can think about opening devices, allocating buffers and so on. However, the stream is not opened in this state, so the stream positions is automatically zero. If a stream was previously opened, it should be closed in this state, and position, properties and such should be reset.
- `GST_STATE_PAUSED`: in this state, an element has opened the stream, but is not actively processing it. An element is allowed to modify a stream's position, read and process data and such to prepare for playback as soon as state is changed to PLAYING, but it is *not* allowed to play the data which would make the clock run. In summary, PAUSED is the same as PLAYING but without a running clock.

  Elements going into the PAUSED state should prepare themselves for moving over to the PLAYING state as soon as possible. Video or audio outputs would, for example, wait for data to arrive and queue it so they can play it right after the state change. Also, video sinks can already play the first frame (since this does not affect the clock yet). Autopluggers could use this same state transition to already plug together a pipeline. Most other elements, such as codecs or filters, do not need to explicitely do anything in this state, however.
- `GST_STATE_PLAYING`: in the PLAYING state, an element does exactly the same as in the PAUSED state, except that the clock now runs.

You can change the state of an element using the function `gst_element_set_state ()`. If you set an element to another state, `GStreamer` will internally traverse all intermediate states. So if you set an element from NULL to PLAYING, `GStreamer` will internally set the element to READY and PAUSED in between.

When moved to `GST_STATE_PLAYING`, pipelines will process data automatically. They do not need to be iterated in any form. Internally, `GStreamer` will start threads that take this task on to them. `GStreamer` will also take care of switching messages from the pipeline's thread into the application's own thread, by using a `GstBus`[15]. See Chapter 7 for details.

Notes

1. ../../gstreamer/html/GstElement.html
2. http://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html/GstElementFacto
   element-factory-make
3. http://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html/GstObject.html#g
   object-unref
4. The code for this example is automatically extracted from the documentation and built under `examples/manual` in the GStreamer tarball.
5. http://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html/GstElement.html
6. http://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html/GstElementFacto
7. http://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html/GstElementFacto
   element-factory-find
8. http://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html/GstElementFacto
   element-factory-create
9. http://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html/GstElement.html
10. http://developer.gnome.org/doc/API/2.0/gobject/index.html
11. http://developer.gnome.org/doc/API/2.0/gobject/pr01.html
12. http://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html/gstreamer/html/
13. http://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html/GstElement.html
14. http://gstreamer.freedesktop.org/modules/gst-editor.html
15. http://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html/GstBus.html

Chapter 6. Bins

A bin is a container element. You can add elements to a bin. Since a bin is an element itself, a bin can be handled in the same way as any other element. Therefore, the whole previous chapter (Elements) applies to bins as well.

What are bins

Bins allow you to combine a group of linked elements into one logical element. You do not deal with the individual elements anymore but with just one element, the bin. We will see that this is extremely powerful when you are going to construct complex pipelines since it allows you to break up the pipeline in smaller chunks.

The bin will also manage the elements contained in it. It will figure out how the data will flow in the bin and generate an optimal plan for that data flow. Plan generation is one of the most complicated procedures in `GStreamer`. You will learn more about this process, called scheduling, in the Section called *Scheduling in `GStreamer`* in Chapter 16.

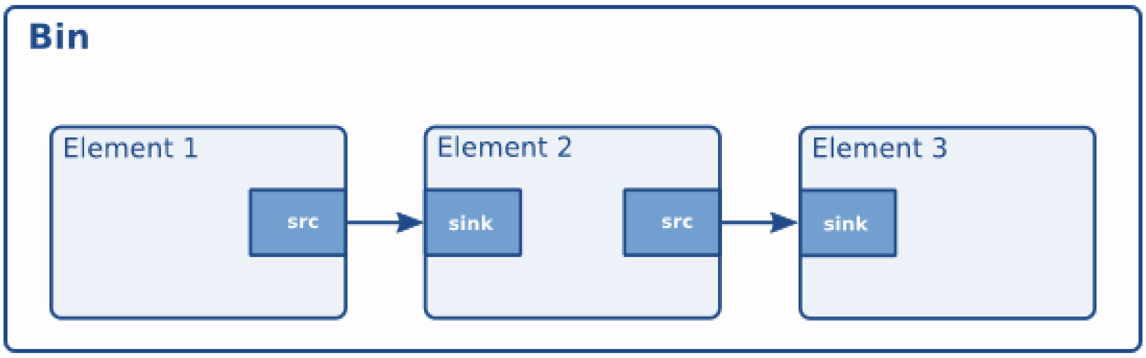

Figure 6-1. Visualisation of a bin with some elements in it

There is one specialized type of bin available to the `GStreamer` programmer:

- A pipeline: a generic container that allows scheduling of the containing elements. The toplevel bin has to be a pipeline, every application thus needs at least one of these. Pipelines will automatically run themselves in a background thread when started.

Creating a bin

Bins are created in the same way that other elements are created, i.e. using an element factory. There are also convenience functions available (`gst_bin_new ()` and `gst_pipeline_new ()`). To add elements to a bin or remove elements from a bin, you can use `gst_bin_add ()` and `gst_bin_remove ()`. Note that the bin that you add an element to will take ownership of that element. If you destroy the bin, the element will be dereferenced with it. If you remove an element from a bin, it will be dereferenced automatically.

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

int
main (int   argc,
      char *argv[])
{
  GstElement *bin, *pipeline, *source, *sink;

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

  /* create */
  pipeline = gst_pipeline_new ("my_pipeline");
  bin = gst_bin_new ("my_bin");
  source = gst_element_factory_make ("fakesrc", "source");
  sink = gst_element_factory_make ("fakesink", "sink");

  /* First add the elements to the bin */
  gst_bin_add_many (GST_BIN (bin), source, sink, NULL);
  /* add the bin to the pipeline */
  gst_bin_add (GST_BIN (pipeline), bin);

  /* link the elements */
  gst_element_link (source, sink);

[..]

}
```

There are various functions to lookup elements in a bin. You can also get a list of all elements that a bin contains using the function `gst_bin_get_list ()`. See the API references of `GstBin`[1] for details.

Custom bins

The application programmer can create custom bins packed with elements to perform a specific task. This allows you, for example, to write an Ogg/Vorbis decoder with just the following lines of code:

```
int
main (int   argc,
      char *argv[])
{
  GstElement *player;

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

  /* create player */
  player = gst_element_factory_make ("oggvorbisplayer", "player");
  /* set the source audio file */
  g_object_set (player, "location", "helloworld.ogg", NULL);

  /* start playback */
  gst_element_set_state (GST_ELEMENT (player), GST_STATE_PLAYING);
[..]
}
```

Custom bins can be created with a plugin or an XML description. You will find more information about creating custom bin in the Plugin Writers Guide[2].

Examples of such custom bins are the playbin and decodebin elements from gst-plugins-base[3].

Notes

1. http://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html/GstBin.html
2. http://gstreamer.freedesktop.org/data/doc/gstreamer/head/pwg/html/index.html
3. http://gstreamer.freedesktop.org/data/doc/gstreamer/head/gst-plugins-base-plugins/html/index.html

Chapter 7. Bus

A bus is a simple system that takes care of forwarding messages from the pipeline threads to an application in its own thread context. The advantage of a bus is that an application does not need to be thread-aware in order to use `GStreamer`, even though `GStreamer` itself is heavily threaded.

Every pipeline contains a bus by default, so applications do not need to create a bus or anything. The only thing applications should do is set a message handler on a bus, which is similar to a signal handler to an object. When the mainloop is running, the bus will periodically be checked for new messages, and the callback will be called when any message is available.

How to use a bus

There are two different ways to use a bus:

- Run a GLib/Gtk+ main loop (or iterate the default GLib main context yourself regularly) and attach some kind of watch to the bus. This way the GLib main loop will check the bus for new messages and notify you whenever there are messages.

  Typically you would use `gst_bus_add_watch ()` or `gst_bus_add_signal_watch ()` in this case.

  To use a bus, attach a message handler to the bus of a pipeline using `gst_bus_add_watch ()`. This handler will be called whenever the pipeline emits a message to the bus. In this handler, check the signal type (see next section) and do something accordingly. The return value of the handler should be TRUE to remove the message from the bus.
- Check for messages on the bus yourself. This can be done using `gst_bus_peek ()` and/or `gst_bus_poll ()`.

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

static GMainLoop *loop;

static gboolean
my_bus_callback (GstBus     *bus,
   GstMessage *message,
   gpointer    data)
{
  g_print ("Got %s message\n", GST_MESSAGE_TYPE_NAME (message));

  switch (GST_MESSAGE_TYPE (message)) {
    case GST_MESSAGE_ERROR: {
      GError *err;
      gchar *debug;

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

      g_main_loop_quit (loop);
      break;
    }
    case GST_MESSAGE_EOS:
      /* end-of-stream */
      g_main_loop_quit (loop);
      break;
    default:
      /* unhandled message */
      break;
  }

  /* we want to be notified again the next time there is a message
   * on the bus, so returning TRUE (FALSE means we want to stop watching
   * for messages on the bus and our callback should not be called again)
   */
  return TRUE;
}

gint
main (gint   argc,
      gchar *argv[])
{
  GstElement *pipeline;
  GstBus *bus;

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

  /* create pipeline, add handler */
  pipeline = gst_pipeline_new ("my_pipeline");

  /* adds a watch for new message on our pipeline’s message bus to
   * the default GLib main context, which is the main context that our
   * GLib main loop is attached to below
   */
  bus = gst_pipeline_get_bus (GST_PIPELINE (pipeline));
  gst_bus_add_watch (bus, my_bus_callback, NULL);
  gst_object_unref (bus);

[..]

  /* create a mainloop that runs/iterates the default GLib main context
   * (context NULL), in other words: makes the context check if anything
   * it watches for has happened. When a message has been posted on the
   * bus, the default main context will automatically call our
   * my_bus_callback() function to notify us of that message.
   * The main loop will be run until someone calls g_main_loop_quit()
   */
  loop = g_main_loop_new (NULL, FALSE);
  g_main_loop_run (loop);

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

  return 0;
}
```

It is important to know that the handler will be called in the thread context of the mainloop. This means that the interaction between the pipeline and application over the bus is *asynchronous*, and thus not suited for some real-time purposes, such as cross-fading between audio tracks, doing (theoretically) gapless playback or video effects. All such things should be done in the pipeline context, which is easiest by writing a `GStreamer` plug-in. It is very useful for its primary purpose, though: passing messages from pipeline to application. The advantage of this approach is that all the threading that `GStreamer` does internally is hidden from the application and the application developer does not have to worry about thread issues at all.

Note that if you're using the default GLib mainloop integration, you can, instead of attaching a watch, connect to the "message" signal on the bus. This way you don't have to `switch()` on all possible message types; just connect to the interesting signals in form of "message::<type>", where <type> is a specific message type (see the next section for an explanation of message types).

The above snippet could then also be written as:

```
GstBus *bus;

[..]

bus = gst_pipeline_get_bus (GST_PIPELINE (pipeline);
gst_bus_add_signal_watch (bus);
g_signal_connect (bus, "message::error", G_CALLBACK (cb_message_error), NULL);
g_signal_connect (bus, "message::eos", G_CALLBACK (cb_message_eos), NULL);

[..]
```

If you aren't using GLib mainloop, the asynchronous message signals won't be available by default. You can however install a custom sync handler that wakes up the custom mainloop and that uses `gst_bus_async_signal_func ()` to emit the signals. (see also documentation[1] for details)

Message types

`GStreamer` has a few pre-defined message types that can be passed over the bus. The messages are extensible, however. Plug-ins can define additional messages, and applications can decide to either have specific code for those or ignore them. All applications are strongly recommended to at least handle error messages by providing visual feedback to the user.

All messages have a message source, type and timestamp. The message source can be used to see which element emitted the message. For some messages, for example, only the ones emitted by the top-level pipeline will be interesting to most applications (e.g. for state-change notifications). Below is a list of all messages and a short explanation of what they do and how to parse message-specific content.

- Error, warning and information notifications: those are used by elements if a message should be shown to the user about the state of the pipeline. Error messages are fatal and terminate the data-passing. The error should be repaired to resume pipeline activity. Warnings are not fatal, but imply a problem nevertheless. Information messages are for non-problem notifications. All those messages contain a `GError` with the main error type and message, and optionally a debug string. Both can be extracted using `gst_message_parse_error ()`, `_parse_warning ()` and `_parse_info ()`. Both error and debug string should be free'ed after use.
- End-of-stream notification: this is emitted when the stream has ended. The state of the pipeline will not change, but further media handling will stall. Applications can use this to skip to the next song in their playlist. After end-of-stream, it is also possible to seek back in the stream. Playback will then continue automatically. This message has no specific arguments.
- Tags: emitted when metadata was found in the stream. This can be emitted multiple times for a pipeline (e.g. once for descriptive metadata such as artist name or song title, and another one for stream-information, such as samplerate and bitrate). Applications should cache metadata internally. `gst_message_parse_tag ()` should be used to parse the taglist, which should be `gst_tag_list_free ()`'ed when no longer needed.
- State-changes: emitted after a successful state change. `gst_message_parse_state_changed ()` can be used to parse the old and new state of this transition.
- Buffering: emitted during caching of network-streams. One can manually extract the progress (in percent) from the message by extracting the "buffer-percent" property from the structure returned by `gst_message_get_structure ()`.
- Element messages: these are special messages that are unique to certain elements and usually represent additional features. The element's documentation should mention in detail which element messages a particular element may send. As an example, the 'qtdemux' QuickTime demuxer element may send a 'redirect' element message on certain occasions if the stream contains a redirect instruction.
- Application-specific messages: any information on those can be extracted by getting the message structure (see above) and reading its fields. Usually these messages can safely be ignored.

  Application messages are primarily meant for internal use in applications in case the application needs to marshal information from some thread into the main thread. This is particularly useful when the application is making use of element signals (as those signals will be emitted in the context of the streaming thread).

Notes

1. http://gstreamer.freedesktop.org/data/doc/gstreamer/stable/gstreamer/html/GstBus.html

Chapter 8. Pads and capabilities

As we have seen in Elements, the pads are the element's interface to the outside world. Data streams from one element's source pad to another element's sink pad. The specific type of media that the element can handle will be exposed by the pad's capabilities. We will talk more on capabilities later in this chapter (see the Section called *Capabilities of a pad*).

Pads

A pad type is defined by two properties: its direction and its availability. As we've mentioned before, `GStreamer` defines two pad directions: source pads and sink pads. This terminology is defined from the view of within the element: elements receive data on their sink pads and generate data on their source pads. Schematically, sink pads are drawn on the left side of an element, whereas source pads are drawn on the right side of an element. In such graphs, data flows from left to right. [1]

Pad directions are very simple compared to pad availability. A pad can have any of three availabilities: always, sometimes and on request. The meaning of those three types is exactly as it says: always pads always exist, sometimes pad exist only in certain cases (and can disappear randomly), and on-request pads appear only if explicitely requested by applications.

Dynamic (or sometimes) pads

Some elements might not have all of their pads when the element is created. This can happen, for example, with an Ogg demuxer element. The element will read the Ogg stream and create dynamic pads for each contained elementary stream (vorbis, theora) when it detects such a stream in the Ogg stream. Likewise, it will delete the pad when the stream ends. This principle is very useful for demuxer elements, for example.

Running `gst-inspect oggdemux` will show that the element has only one pad: a sink pad called 'sink'. The other pads are "dormant". You can see this in the pad template because there is an "Exists: Sometimes" property. Depending on the type of Ogg file you play, the pads will be created. We will see that this is very important when you are going to create dynamic pipelines. You can attach a signal handler to an element to inform you when the element has created a new pad from one of its "sometimes" pad templates. The following piece of code is an example of how to do this:

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

static void
cb_new_pad (GstElement *element,
     GstPad     *pad,
     gpointer    data)
{
  gchar *name;

  name = gst_pad_get_name (pad);
  g_print ("A new pad %s was created\n", name);
  g_free (name);
  /* here, you would setup a new pad link for the newly created pad */
[..]

}

int
main (int   argc,
      char *argv[])
{
  GstElement *pipeline, *source, *demux;
  GMainLoop *loop;

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

  /* create elements */
  pipeline = gst_pipeline_new ("my_pipeline");
  source = gst_element_factory_make ("filesrc", "source");
  g_object_set (source, "location", argv[1], NULL);
  demux = gst_element_factory_make ("oggdemux", "demuxer");

  /* you would normally check that the elements were created properly */

  /* put together a pipeline */
  gst_bin_add_many (GST_BIN (pipeline), source, demux, NULL);
  gst_element_link_pads (source, "src", demux, "sink");

  /* listen for newly created pads */
  g_signal_connect (demux, "pad-added", G_CALLBACK (cb_new_pad), NULL);

  /* start the pipeline */
  gst_element_set_state (GST_ELEMENT (pipeline), GST_STATE_PLAYING);
  loop = g_main_loop_new (NULL, FALSE);
  g_main_loop_run (loop);

[..]

}
```

Request pads

An element can also have request pads. These pads are not created automatically but are only created on demand. This is very useful for multiplexers, aggregators and tee elements. Aggregators are elements that merge the content of several input streams together into one output stream. Tee elements are the reverse: they are elements that have one input stream and copy this stream to each of their output pads, which are created on request. Whenever an application needs another copy of the stream, it can simply request a new output pad from the tee element.

The following piece of code shows how you can request a new output pad from a “tee” element:

```
static void
some_function (GstElement *tee)
{
  GstPad * pad;
  gchar *name;

  pad = gst_element_get_request_pad (tee, "src%d");
  name = gst_pad_get_name (pad);
  g_print ("A new pad %s was created\n", name);
  g_free (name);

  /* here, you would link the pad */
[..]

  /* and, after doing that, free our reference */
  gst_object_unref (GST_OBJECT (pad));
}
```

The `gst_element_get_request_pad ()` method can be used to get a pad from the element based on the name of the pad template. It is also possible to request a pad that is compatible with another pad template. This is very useful if you want to link an element to a multiplexer element and you need to request a pad that is compatible. The method `gst_element_get_compatible_pad ()` can be used to request a compatible pad, as shown in the next example. It will request a compatible pad from an Ogg multiplexer from any input.

```
static void
link_to_multiplexer (GstPad     *tolink_pad,
       GstElement *mux)
{
  GstPad *pad;
  gchar *srcname, *sinkname;

  srcname = gst_pad_get_name (tolink_pad);
  pad = gst_element_get_compatible_pad (mux, tolink_pad);
  gst_pad_link (tolinkpad, pad);
  sinkname = gst_pad_get_name (pad);
  gst_object_unref (GST_OBJECT (pad));

  g_print ("A new pad %s was created and linked to %s\n", srcname, sinkname);
  g_free (sinkname);
  g_free (srcname);
}
```

Capabilities of a pad

Since the pads play a very important role in how the element is viewed by the outside world, a mechanism is implemented to describe the data that can flow or currently flows through the pad by using capabilities. Here, we will briefly describe what capabilities are and how to use them, enough to get an understanding of the concept. For an in-depth look into capabilities and a list of all capabilities defined in `GStreamer`, see the Plugin Writers Guide[2].

Capabilities are attached to pad templates and to pads. For pad templates, it will describe the types of media that may stream over a pad created from this template. For pads, it can either be a list of possible caps (usually a copy of the pad template's capabilities), in which case the pad is not yet negotiated, or it is the type of media that currently streams over this pad, in which case the pad has been negotiated already.

Dissecting capabilities

A pads capabilities are described in a `GstCaps` object. Internally, a `GstCaps`[3] will contain one or more `GstStructure`[4] that will describe one media type. A negotiated pad will have capabilities set that contain exactly *one* structure. Also, this structure will contain only *fixed* values. These constraints are not true for unnegotiated pads or pad templates.

As an example, below is a dump of the capabilities of the "vorbisdec" element, which you will get by running **gst-inspect vorbisdec**. You will see two pads: a source and a sink pad. Both of these pads are always available, and both have capabilities attached to them. The sink pad will accept vorbis-encoded audio data, with the mime-type "audio/x-vorbis". The source pad will be used to send raw (decoded) audio samples to the next element, with a raw audio mime-type (in this case, "audio/x-raw-int") The source pad will also contain properties for the audio samplerate and the amount of channels, plus some more that you don't need to worry about for now.

```
Pad Templates:
  SRC template: 'src'
    Availability: Always
    Capabilities:
      audio/x-raw-float
                   rate: [ 8000, 50000 ]
               channels: [ 1, 2 ]
             endianness: 1234
                  width: 32
          buffer-frames: 0

  SINK template: 'sink'
    Availability: Always
    Capabilities:
      audio/x-vorbis
```

Properties and values

Properties are used to describe extra information for capabilities. A property consists of a key (a string) and a value. There are different possible value types that can be used:

- Basic types, this can be pretty much any `GType` registered with Glib. Those properties indicate a specific, non-dynamic value for this property. Examples include:

  - An integer value (`G_TYPE_INT`): the property has this exact value.
  - A boolean value (`G_TYPE_BOOLEAN`): the property is either TRUE or FALSE.
  - A float value (`G_TYPE_FLOAT`): the property has this exact floating point value.
  - A string value (`G_TYPE_STRING`): the property contains a UTF-8 string.
  - A fraction value (`GST_TYPE_FRACTION`): contains a fraction expressed by an integer numerator and denominator.

- Range types are `GTypes` registered by `GStreamer` to indicate a range of possible values. They are used for indicating allowed audio samplerate values or supported video sizes. The two types defined in `GStreamer` are:
  - An integer range value (`GST_TYPE_INT_RANGE`): the property denotes a range of possible integers, with a lower and an upper boundary. The "vorbisdec" element, for example, has a rate property that can be between 8000 and 50000.
  - A float range value (`GST_TYPE_FLOAT_RANGE`): the property denotes a range of possible floating point values, with a lower and an upper boundary.
  - A fraction range value (`GST_TYPE_FRACTION_RANGE`): the property denotes a range of possible fraction values, with a lower and an upper boundary.

- A list value (`GST_TYPE_LIST`): the property can take any value from a list of basic values given in this list.

  Example: caps that express that either a sample rate of 44100 Hz and a sample rate of 48000 Hz is supported would use a list of integer values, with one value being 44100 and one value being 48000.
- An array value (`GST_TYPE_ARRAY`): the property is an array of values. Each value in the array is a full value on its own, too. All values in the array should be of the same elementary type. This means that an array can contain any combination of integers, lists of integers, integer ranges together, and the same for floats or strings, but it can not contain both floats and ints at the same time.

  Example: for audio where there are more than two channels involved the channel layout needs to be specified (for one and two channel audio the channel layout is implicit unless stated otherwise in the caps). So the channel layout would be an array of integer enum values where each enum value represents a loudspeaker position. Unlike a `GST_TYPE_LIST`, the values in an array will be interpreted as a whole.

What capabilities are used for

Capabilities (short: caps) describe the type of data that is streamed between two pads, or that one pad (template) supports. This makes them very useful for various purposes:

- Autoplugging: automatically finding elements to link to a pad based on its capabilities. All autopluggers use this method.
- Compatibility detection: when two pads are linked, `GStreamer` can verify if the two pads are talking about the same media type. The process of linking two pads and checking if they are compatible is called “caps negotiation”.
- Metadata: by reading the capabilities from a pad, applications can provide information about the type of media that is being streamed over the pad, which is information about the stream that is currently being played back.
- Filtering: an application can use capabilities to limit the possible media types that can stream between two pads to a specific subset of their supported stream types. An application can, for example, use “filtered caps” to set a specific (fixed or non-fixed) video size that should stream between two pads. You will see an example of filtered caps later in this manual, in the Section called *Manually adding or removing data from/to a pipeline* in Chapter 18. You can do caps filtering by inserting a capsfilter element into your pipeline and setting its “caps” property. Caps filters are often placed after converter elements like audioconvert, audioresample, ffmpegcolorspace or videoscale to force those converters to convert data to a specific output format at a certain point in a stream.

Using capabilities for metadata

A pad can have a set (i.e. one or more) of capabilities attached to it. Capabilities (`GstCaps`) are represented as an array of one or more `GstStructure`s, and each `GstStructure` is an array of fields where each field consists of a field name string (e.g. "width") and a typed value (e.g. `G_TYPE_INT` or `GST_TYPE_INT_RANGE`).

Note that there is a distinct difference between the *possible* capabilities of a pad (ie. usually what you find as caps of pad templates as they are shown in gst-inspect), the *allowed* caps of a pad (can be the same as the pad’s template caps or a subset of them, depending on the possible caps of the peer pad) and lastly *negotiated* caps (these describe the exact format of a stream or buffer and contain exactly one structure and have no variable bits like ranges or lists, ie. they are fixed caps).

You can get values of properties in a set of capabilities by querying individual properties of one structure. You can get a structure from a caps using `gst_caps_get_structure ()` and the number of structures in a `GstCaps` using `gst_caps_get_size ()`.

Caps are called *simple caps* when they contain only one structure, and *fixed caps* when they contain only one structure and have no variable field types (like ranges or lists of possible values). Two other special types of caps are *ANY caps* and *empty caps*.

Here is an example of how to extract the width and height from a set of fixed video caps:

```
static void
read_video_props (GstCaps *caps)
{
  gint width, height;
  const GstStructure *str;

  g_return_if_fail (gst_caps_is_fixed (caps));
  str = gst_caps_get_structure (caps, 0);
  if (!gst_structure_get_int (str, "width", &width) ||
      !gst_structure_get_int (str, "height", &height)) {
    g_print ("No width/height available\n");
    return;
  }

  g_print ("The video size of this set of capabilities is %dx%d\n",
    width, height);
}
```

Creating capabilities for filtering

While capabilities are mainly used inside a plugin to describe the media type of the pads, the application programmer often also has to have basic understanding of capabilities in order to interface with the plugins, especially when using filtered caps. When you're using filtered caps or fixation, you're limiting the allowed types of media that can stream between two pads to a subset of their supported media types. You do this using a `capsfilter` element in your pipeline. In order to do this, you also need to create your own `GstCaps`. The easiest way to do this is by using the convenience function `gst_caps_new_simple ()`:

```
static gboolean
link_elements_with_filter (GstElement *element1, GstElement *element2)
{
  gboolean link_ok;
  GstCaps *caps;

  caps = gst_caps_new_simple ("video/x-raw-yuv",
         "format", GST_TYPE_FOURCC, GST_MAKE_FOURCC ('I', '4', '2', '0'),
      "width", G_TYPE_INT, 384,
      "height", G_TYPE_INT, 288,
      "framerate", GST_TYPE_FRACTION, 25, 1,
      NULL);

  link_ok = gst_element_link_filtered (element1, element2, caps);
  gst_caps_unref (caps);

  if (!link_ok) {
    g_warning ("Failed to link element1 and element2!");
  }

  return link_ok;
}
```

This will force the data flow between those two elements to a certain video format, width, height and framerate (or the linking will fail if that cannot be achieved in the context of the elments involved). Keep in mind that when you use `gst_element_link_filtered ()` it will automatically create a `capsfilter` element for you and insert it into your bin or pipeline between the two elements you want to connect (this is important if you ever want to disconnect those elements because then you will have to disconnect both elements from the capsfilter instead).

In some cases, you will want to create a more elaborate set of capabilities to filter a link between two pads. Then, this function is too simplistic and you'll want to use the method `gst_caps_new_full ()`:

```
static gboolean
link_elements_with_filter (GstElement *element1, GstElement *element2)
{
  gboolean link_ok;
  GstCaps *caps;

  caps = gst_caps_new_full (
      gst_structure_new ("video/x-raw-yuv",
    "width", G_TYPE_INT, 384,
    "height", G_TYPE_INT, 288,
    "framerate", GST_TYPE_FRACTION, 25, 1,
    NULL),
      gst_structure_new ("video/x-raw-rgb",
    "width", G_TYPE_INT, 384,
    "height", G_TYPE_INT, 288,
    "framerate", GST_TYPE_FRACTION, 25, 1,
    NULL),
      NULL);

  link_ok = gst_element_link_filtered (element1, element2, caps);
  gst_caps_unref (caps);

  if (!link_ok) {
    g_warning ("Failed to link element1 and element2!");
  }

  return link_ok;
}
```

See the API references for the full API of `GstStructure` and `GstCaps`.

Ghost pads

You can see from Figure 8-1 how a bin has no pads of its own. This is where "ghost pads" come into play.

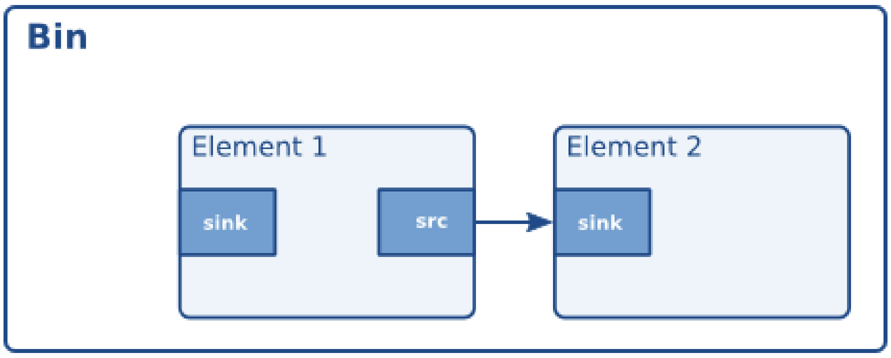

Figure 8-1. Visualisation of a `GstBin`[5] element without ghost pads

A ghost pad is a pad from some element in the bin that can be accessed directly from the bin as well. Compare it to a symbolic link in UNIX filesystems. Using ghost pads on bins, the bin also has a pad and can transparently be used as an element in other parts of your code.

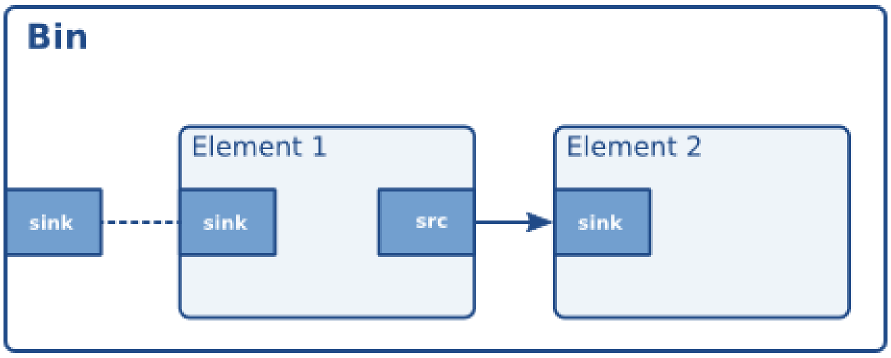

Figure 8-2. Visualisation of a `GstBin`[6] element with a ghost pad

Figure 8-2 is a representation of a ghost pad. The sink pad of element one is now also a pad of the bin. Because ghost pads look and work like any other pads, they can be added to any type of elements, not just to a `GstBin`, just like ordinary pads.

A ghostpad is created using the function `gst_ghost_pad_new ()`:

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

int
main (int   argc,
      char *argv[])
{
  GstElement *bin, *sink;
  GstPad *pad;

  /* init */
  gst_init (&argc, &argv);
  /* create element, add to bin */
  sink = gst_element_factory_make ("fakesink", "sink");
  bin = gst_bin_new ("mybin");
  gst_bin_add (GST_BIN (bin), sink);

  /* add ghostpad */
  pad = gst_element_get_static_pad (sink, "sink");
  gst_element_add_pad (bin, gst_ghost_pad_new ("sink", pad));
  gst_object_unref (GST_OBJECT (pad));

[..]

}
```

In the above example, the bin now also has a pad: the pad called “sink” of the given element. The bin can, from here on, be used as a substitute for the sink element. You could, for example, link another element to the bin.

Notes

1. In reality, there is no objection to data flowing from a source pad to the sink pad of an element upstream (to the left of this element in drawings). Data will, however, always flow from a source pad of one element to the sink pad of another.
2. http://gstreamer.freedesktop.org/data/doc/gstreamer/head/pwg/html/index.html
3. ../../gstreamer/html/gstreamer-GstCaps.html
4. ../../gstreamer/html/gstreamer-GstStructure.html
5. ../../gstreamer/html/GstBin.html
6. ../../gstreamer/html/GstBin.html

Chapter 9. Buffers and Events

The data flowing through a pipeline consists of a combination of buffers and events. Buffers contain the actual media data. Events contain control information, such as seeking information and end-of-stream notifiers. All this will flow through the pipeline automatically when it's running. This chapter is mostly meant to explain the concept to you; you don't need to do anything for this.

Buffers

Buffers contain the data that will flow through the pipeline you have created. A source element will typically create a new buffer and pass it through a pad to the next element in the chain. When using the GStreamer infrastructure to create a media pipeline you will not have to deal with buffers yourself; the elements will do that for you.

A buffer consists, amongst others, of:

- A pointer to a piece of memory.
- The size of the memory.
- A timestamp for the buffer.
- A refcount that indicates how many elements are using this buffer. This refcount will be used to destroy the buffer when no element has a reference to it.
- Buffer flags.

The simple case is that a buffer is created, memory allocated, data put in it, and passed to the next element. That element reads the data, does something (like creating a new buffer and decoding into it), and unreferences the buffer. This causes the data to be free'ed and the buffer to be destroyed. A typical video or audio decoder works like this.

There are more complex scenarios, though. Elements can modify buffers in-place, i.e. without allocating a new one. Elements can also write to hardware memory (such as from video-capture sources) or memory allocated from the X-server (using XShm). Buffers can be read-only, and so on.

Events

Events are control particles that are sent both up- and downstream in a pipeline along with buffers. Downstream events notify fellow elements of stream states. Possible events include seeking, flushes, end-of-stream notifications and so on. Upstream events are used both in application-element interaction as well as element-element interaction to request changes in stream state, such as seeks. For applications, only upstream events are important. Downstream events are just explained to get a more complete picture of the data concept.

Since most applications seek in time units, our example below does so too:

```
static void
```

```
seek_to_time (GstElement *element,
       guint64     time_ns)
{
  GstEvent *event;

  event = gst_event_new_seek (1.0, GST_FORMAT_TIME,
         GST_SEEK_FLAG_NONE,
         GST_SEEK_METHOD_SET, time_ns,
         GST_SEEK_TYPE_NONE, G_GUINT64_CONSTANT (0));
  gst_element_send_event (element, event);
}
```

The function `gst_element_seek ()` is a shortcut for this. This is mostly just to show how it all works.

Chapter 10. Your first application

This chapter will summarize everything you've learned in the previous chapters. It describes all aspects of a simple `GStreamer` application, including initializing libraries, creating elements, packing elements together in a pipeline and playing this pipeline. By doing all this, you will be able to build a simple Ogg/Vorbis audio player.

Hello world

We're going to create a simple first application, a simple Ogg/Vorbis command-line audio player. For this, we will use only standard `GStreamer` components. The player will read a file specified on the command-line. Let's get started!

We've learned, in Chapter 4, that the first thing to do in your application is to initialize `GStreamer` by calling `gst_init ()`. Also, make sure that the application includes `gst/gst.h` so all function names and objects are properly defined. Use `#include <gst/gst.h>` to do that.

Next, you'll want to create the different elements using `gst_element_factory_make ()`. For an Ogg/Vorbis audio player, we'll need a source element that reads files from a disk. `GStreamer` includes this element under the name "filesrc". Next, we'll need something to parse the file and decode it into raw audio. `GStreamer` has two elements for this: the first parses Ogg streams into elementary streams (video, audio) and is called "oggdemux". The second is a Vorbis audio decoder, it's conveniently called "vorbisdec". Since "oggdemux" creates dynamic pads for each elementary stream, you'll need to set a "pad-added" event handler on the "oggdemux" element, like you've learned in the Section called *Dynamic (or sometimes) pads* in Chapter 8, to link the Ogg demuxer and the Vorbis decoder elements together. At last, we'll also need an audio output element, we will use "autoaudiosink", which automatically detects your audio device.

The last thing left to do is to add all elements into a container element, a `GstPipeline`, and iterate this pipeline until we've played the whole song. We've previously learned how to add elements to a container bin in Chapter 6, and we've learned about element states in the Section called *Element States* in Chapter 5. We will also attach a message handler to the pipeline bus so we can retrieve errors and detect the end-of-stream.

Let's now add all the code together to get our very first audio player:

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


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

  switch (GST_MESSAGE_TYPE (msg)) {
    case GST_MESSAGE_EOS:
      g_print ("End of stream\n");
      g_main_loop_quit (loop);
      break;

    case GST_MESSAGE_ERROR: {
      gchar  *debug;
      GError *error;

      gst_message_parse_error (msg, &error, &debug);
      g_free (debug);

      g_printerr ("Error: %s\n", error->message);
      g_error_free (error);

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

  return TRUE;
}


static void
on_pad_added (GstElement *element,
              GstPad     *pad,
              gpointer    data)
{
  GstPad *sinkpad;
  GstElement *decoder = (GstElement *) data;

  /* We can now link this pad with the vorbis-decoder sink pad */
  g_print ("Dynamic pad created, linking demuxer/decoder\n");

  sinkpad = gst_element_get_static_pad (decoder, "sink");

  gst_pad_link (pad, sinkpad);

  gst_object_unref (sinkpad);
}



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

  GstElement *pipeline, *source, *demuxer, *decoder, *conv, *sink;
  GstBus *bus;
```

```
/* Initialisation */
gst_init (&argc, &argv);

loop = g_main_loop_new (NULL, FALSE);


/* Check input arguments */
if (argc != 2) {
  g_printerr ("Usage: %s <Ogg/Vorbis filename>\n", argv[0]);
  return -1;
}


/* Create gstreamer elements */
pipeline = gst_pipeline_new ("audio-player");
source   = gst_element_factory_make ("filesrc",       "file-source");
demuxer  = gst_element_factory_make ("oggdemux",      "ogg-demuxer");
decoder  = gst_element_factory_make ("vorbisdec",     "vorbis-decoder");
conv     = gst_element_factory_make ("audioconvert",  "converter");
sink     = gst_element_factory_make ("autoaudiosink", "audio-output");

if (!pipeline || !source || !demuxer || !decoder || !conv || !sink) {
  g_printerr ("One element could not be created. Exiting.\n");
  return -1;
}

/* Set up the pipeline */

/* we set the input filename to the source element */
g_object_set (G_OBJECT (source), "location", argv[1], NULL);

/* we add a message handler */
bus = gst_pipeline_get_bus (GST_PIPELINE (pipeline));
gst_bus_add_watch (bus, bus_call, loop);
gst_object_unref (bus);

/* we add all elements into the pipeline */
/* file-source | ogg-demuxer | vorbis-decoder | converter | alsa-output */
gst_bin_add_many (GST_BIN (pipeline),
                  source, demuxer, decoder, conv, sink, NULL);

/* we link the elements together */
/* file-source -> ogg-demuxer ~> vorbis-decoder -> converter -> alsa-output */
gst_element_link (source, demuxer);
gst_element_link_many (decoder, conv, sink, NULL);
g_signal_connect (demuxer, "pad-added", G_CALLBACK (on_pad_added), decoder);

/* note that the demuxer will be linked to the decoder dynamically.
   The reason is that Ogg may contain various streams (for example
   audio and video). The source pad(s) will be created at run time,
   by the demuxer when it detects the amount and nature of streams.
   Therefore we connect a callback function which will be executed
   when the "pad-added" is emitted.*/


/* Set the pipeline to "playing" state*/
```

```
  g_print ("Now playing: %s\n", argv[1]);
  gst_element_set_state (pipeline, GST_STATE_PLAYING);


  /* Iterate */
  g_print ("Running...\n");
  g_main_loop_run (loop);


  /* Out of the main loop, clean up nicely */
  g_print ("Returned, stopping playback\n");
  gst_element_set_state (pipeline, GST_STATE_NULL);

  g_print ("Deleting pipeline\n");
  gst_object_unref (GST_OBJECT (pipeline));

  return 0;
}
```

We now have created a complete pipeline. We can visualise the pipeline as follows:

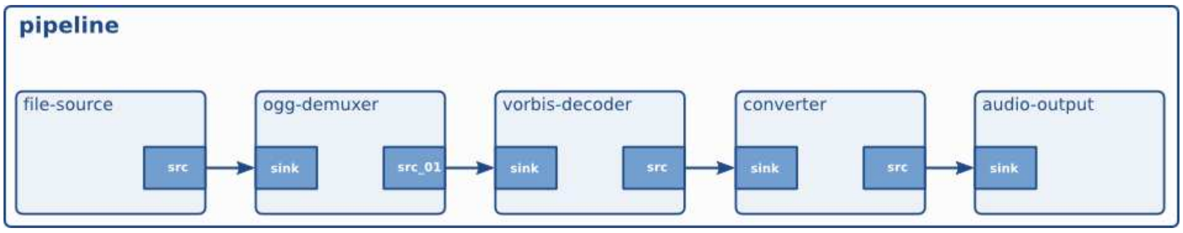

Figure 10-1. The "hello world" pipeline

Compiling and Running helloworld.c

To compile the helloworld example, use: **gcc -Wall $(pkg-config --cflags --libs gstreamer-0.10) helloworld.c -o helloworld**. `GStreamer` makes use of **pkg-config** to get compiler and linker flags needed to compile this application. If you're running a non-standard installation, make sure the `PKG_CONFIG_PATH` environment variable is set to the correct location (`$libdir/pkgconfig`). application against the uninstalled location.

You can run this example application with **./helloworld file.ogg**. Substitute `file.ogg` with your favourite Ogg/Vorbis file.

Conclusion

This concludes our first example. As you see, setting up a pipeline is very low-level but powerful. You will see later in this manual how you can create a more powerful media player with even less effort using higher-level interfaces. We will discuss all that in Part IV in `GStreamer` *Application Development Manual (0.10.23)*. We will first, however, go more in-depth into more advanced `GStreamer` internals.

It should be clear from the example that we can very easily replace the “filesrc” element with some other element that reads data from a network, or some other data source element that is better integrated with your desktop environment. Also, you can use other decoders and parsers/demuxers to support other media types. You can use another audio sink if you’re not running Linux, but Mac OS X, Windows or FreeBSD, or you can instead use a filesink to write audio files to disk instead of playing them back. By using an audio card source, you can even do audio capture instead of playback. All this shows the reusability of `GStreamer` elements, which is its greatest advantage.

Chapter 11. Position tracking and seeking

So far, we've looked at how to create a pipeline to do media processing and how to make it run. Most application developers will be interested in providing feedback to the user on media progress. Media players, for example, will want to show a slider showing the progress in the song, and usually also a label indicating stream length. Transcoding applications will want to show a progress bar on how much percent of the task is done. `GStreamer` has built-in support for doing all this using a concept known as *querying*. Since seeking is very similar, it will be discussed here as well. Seeking is done using the concept of *events*.

Querying: getting the position or length of a stream

Querying is defined as requesting a specific stream-property related to progress tracking. This includes getting the length of a stream (if available) or getting the current position. Those stream properties can be retrieved in various formats such as time, audio samples, video frames or bytes. The function most commonly used for this is `gst_element_query ()`, although some convenience wrappers are provided as well (such as `gst_element_query_position ()` and `gst_element_query_duration ()`). You can generally query the pipeline directly, and it'll figure out the internal details for you, like which element to query.

Internally, queries will be sent to the sinks, and "dispatched" backwards until one element can handle it; that result will be sent back to the function caller. Usually, that is the demuxer, although with live sources (from a webcam), it is the source itself.

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




static gboolean
cb_print_position (GstElement *pipeline)
{
  GstFormat fmt = GST_FORMAT_TIME;
  gint64 pos, len;

  if (gst_element_query_position (pipeline, &fmt, &pos)
    && gst_element_query_duration (pipeline, &fmt, &len)) {
    g_print ("Time: %" GST_TIME_FORMAT " / %" GST_TIME_FORMAT "\r",
      GST_TIME_ARGS (pos), GST_TIME_ARGS (len));
  }

  /* call me again */
  return TRUE;
}

gint
main (gint   argc,
      gchar *argv[])
{
  GstElement *pipeline;
[..]

  /* run pipeline */
  g_timeout_add (200, (GSourceFunc) cb_print_position, pipeline);
  g_main_loop_run (loop);

[..]

}
```

Events: seeking (and more)

Events work in a very similar way as queries. Dispatching, for example, works exactly the same for events (and also has the same limitations), and they can similarly be sent to the toplevel pipeline and it will figure out everything for you. Although there are more ways in which applications and elements can interact using events, we will only focus on seeking here. This is done using the seek-event. A seek-event contains a playback rate, a seek offset format (which is the unit of the offsets to follow, e.g. time, audio samples, video frames or bytes), optionally a set of seeking-related flags (e.g. whether internal buffers should be flushed), a seek method (which indicates relative to what the offset was given), and seek offsets. The first offset (cur) is the new position to seek to, while the second offset (stop) is optional and specifies a position where streaming is supposed to stop. Usually it is fine to just specify GST_SEEK_TYPE_NONE and -1 as end_method and end offset. The behaviour of a seek is also wrapped in the `gst_element_seek ()`.

```
static void
seek_to_time (GstElement *pipeline,
       gint64      time_nanoseconds)
{
  if (!gst_element_seek (pipeline, 1.0, GST_FORMAT_TIME, GST_SEEK_FLAG_FLUSH,
                         GST_SEEK_TYPE_SET, time_nanoseconds,
                         GST_SEEK_TYPE_NONE, GST_CLOCK_TIME_NONE)) {
    g_print ("Seek failed!\n");
  }
}
```

Seeks with the GST_SEEK_FLAG_FLUSH should be done when the pipeline is in PAUSED or PLAYING state. The pipeline will automatically go to preroll state until the new data after the seek will cause the pipeline to preroll again. After the pipeline is prerolled, it will go back to the state (PAUSED or PLAYING) it was in when the seek was executed. You can wait (blocking) for the seek to complete with `gst_element_get_state()` or by waiting for the ASYNC_DONE message to appear on the bus.

Seeks without the GST_SEEK_FLAG_FLUSH should only be done when the pipeline is in the PLAYING state. Executing a non-flushing seek in the PAUSED state might deadlock because the pipeline streaming threads might be blocked in the sinks.

It is important to realise that seeks will not happen instantly in the sense that they are finished when the function `gst_element_seek ()` returns. Depending on the specific elements involved, the actual seeking might be done later in another thread (the streaming thread), and it might take a short time until buffers from the new seek position will reach downstream elements such as sinks (if the seek was non-flushing then it might take a bit longer).

It is possible to do multiple seeks in short time-intervals, such as a direct response to slider movement. After a seek, internally, the pipeline will be paused (if it was playing), the position will be re-set internally, the demuxers and decoders will decode from the new position onwards and this will continue until all sinks have data again. If it was playing originally, it will be set to playing again, too. Since the new position is immediately available in a video output, you will see the new frame, even if your pipeline is not in the playing state.

Chapter 12. Metadata

`GStreamer` makes a clear distinction between two types of metadata, and has support for both types. The first is stream tags, which describe the content of a stream in a non-technical way. Examples include the author of a song, the title of that very same song or the album it is a part of. The other type of metadata is stream-info, which is a somewhat technical description of the properties of a stream. This can include video size, audio samplerate, codecs used and so on. Tags are handled using the `GStreamer` tagging system. Stream-info can be retrieved from a `GstPad`.

Metadata reading

Stream information can most easily be read by reading them from a `GstPad`. This has already been discussed before in the Section called *Using capabilities for metadata* in Chapter 8. Therefore, we will skip it here. Note that this requires access to all pads of which you want stream information.

Tag reading is done through a bus in `GStreamer`, which has been discussed previously in Chapter 7. You can listen for `GST_MESSAGE_TAG` messages and handle them as you wish.

Note, however, that the `GST_MESSAGE_TAG` message may be fired multiple times in the pipeline. It is the application's responsibility to put all those tags together and display them to the user in a nice, coherent way. Usually, using `gst_tag_list_merge ()` is a good enough way of doing this; make sure to empty the cache when loading a new song, or after every few minutes when listening to internet radio. Also, make sure you use `GST_TAG_MERGE_PREPEND` as merging mode, so that a new title (which came in later) has a preference over the old one for display.

Tag writing

Tag writing is done using the `GstTagSetter` interface. All that's required is a tag-set-supporting element in your pipeline. In order to see if any of the elements in your pipeline supports tag writing, you can use the function `gst_bin_iterate_all_by_interface (pipeline, GST_TYPE_TAG_SETTER)`.
On the resulting element, usually an encoder or muxer, you can use `gst_tag_setter_merge ()` (with a taglist) or `gst_tag_setter_add ()` (with individual tags) to set tags on it.

A nice extra feature in `GStreamer` tag support is that tags are preserved in pipelines. This means that if you transcode one file containing tags into another media type, and that new media type supports tags too, then the tags will be handled as part of the data stream and be merged into the newly written media file, too.

Chapter 13. Interfaces

In the Section called *Using an element as a `GObject`* in Chapter 5, you have learned how to use `GObject` properties as a simple way to do interaction between applications and elements. This method suffices for the simple'n'straight settings, but fails for anything more complicated than a getter and setter. For the more complicated use cases, `GStreamer` uses interfaces based on the Glib `GInterface` type.

Most of the interfaces handled here will not contain any example code. See the API references for details. Here, we will just describe the scope and purpose of each interface.

The URI interface

In all examples so far, we have only supported local files through the "filesrc" element. `GStreamer`, obviously, supports many more location sources. However, we don't want applications to need to know any particular element implementation details, such as element names for particular network source types and so on. Therefore, there is a URI interface, which can be used to get the source element that supports a particular URI type. There is no strict rule for URI naming, but in general we follow naming conventions that others use, too. For example, assuming you have the correct plugins installed, `GStreamer` supports "file:///<path>/<file>", "http://<host>/<path>/<file>", "mms://<host>/<path>/<file>", and so on.

In order to get the source or sink element supporting a particular URI, use `gst_element_make_from_uri ()`, with the URI type being either `GST_URI_SRC` for a source element, or `GST_URI_SINK` for a sink element.

The Mixer interface

The mixer interface provides a uniform way to control the volume on a hardware (or software) mixer. The interface is primarily intended to be implemented by elements for audio inputs and outputs that talk directly to the hardware (e.g. OSS or ALSA plugins).

Using this interface, it is possible to control a list of tracks (such as Line-in, Microphone, etc.) from a mixer element. They can be muted, their volume can be changed and, for input tracks, their record flag can be set as well.

Example plugins implementing this interface include the OSS elements (osssrc, ossink, ossmixer) and the ALSA plugins (alsasrc, alsasink and alsamixer).

The Tuner interface

The tuner interface is a uniform way to control inputs and outputs on a multi-input selection device. This is primarily used for input selection on elements for TV- and capture-cards.

Using this interface, it is possible to select one track from a list of tracks supported by that tuner-element. The tuner will than select that track for media-processing internally. This can, for example, be used to switch inputs on a TV-card (e.g. from Composite to S-video).

This interface is currently only implemented by the Video4linux and Video4linux2 elements.

The Color Balance interface

The colorbalance interface is a way to control video-related properties on an element, such as brightness, contrast and so on. It's sole reason for existance is that, as far as its authors know, there's no way to dynamically register properties using `GObject`.

The colorbalance interface is implemented by several plugins, including xvimagesink and the Video4linux and Video4linux2 elements.

The Property Probe interface

The property probe is a way to autodetect allowed values for a `GObject` property. It's primary use is to autodetect devices in several elements. For example, the OSS elements use this interface to detect all OSS devices on a system. Applications can then "probe" this property and get a list of detected devices.

> **Note:** Given the overlap between HAL and the practical implementations of this interface, this might in time be deprecated in favour of HAL.

This interface is currently implemented by many elements, including the ALSA, OSS, XVImageSink, Video4linux and Video4linux2 elements.

The X Overlay interface

The X Overlay interface was created to solve the problem of embedding video streams in an application window. The application provides an X-window to the element implementing this interface to draw on, and the element will then use this X-window to draw on rather than creating a new toplevel window. This is useful to embed video in video players.

This interface is implemented by, amongst others, the Video4linux and Video4linux2 elements and by ximagesink, xvimagesink and sdlvideosink.

Chapter 14. Clocks in GStreamer

To maintain sync in pipeline playback (which is the only case where this really matters), `GStreamer` uses *clocks*. Clocks are exposed by some elements, whereas other elements are merely clock slaves. The primary task of a clock is to represent the time progress according to the element exposing the clock, based on its own playback rate. If no clock provider is available in a pipeline, the system clock is used instead.

`GStreamer` derives several times from the clock and the playback state. It is important to note, that a *clock-time* is monotonically rising, but the value itself is not meaningful. Subtracting the *base-time* yields the *running-time*. It is the same as the *stream-time* if one plays from start to end at original rate. The *stream-time* indicates the position in the media.

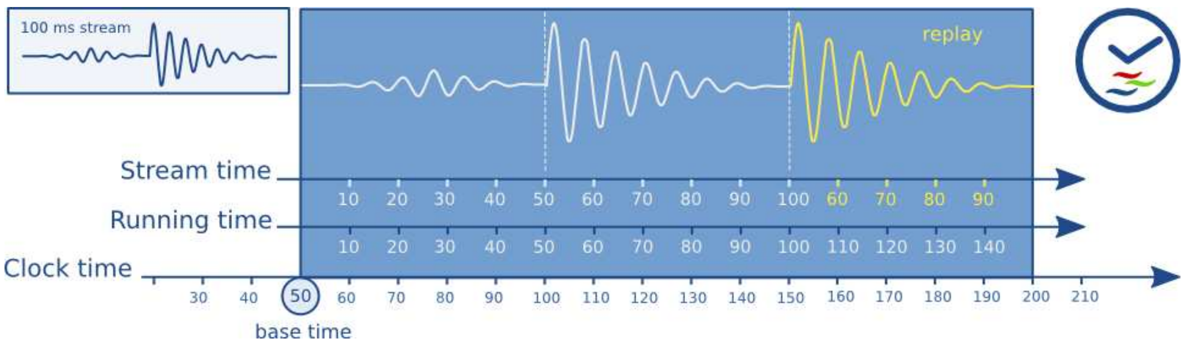

Figure 14-1. `GStreamer` clock and various times

Clock providers

Clock providers exist because they play back media at some rate, and this rate is not necessarily the same as the system clock rate. For example, a soundcard may playback at 44,1 kHz, but that doesn't mean that after *exactly* 1 second *according to the system clock*, the soundcard has played back 44.100 samples. This is only true by approximation. Therefore, generally, pipelines with an audio output use the audiosink as clock provider. This ensures that one second of video will be played back at the same rate as that the soundcard plays back 1 second of audio.

Whenever some part of the pipeline requires to know the current clock time, it will be requested from the clock through `gst_clock_get_time ()`. The clock-time does not need to start at 0. The pipeline, which contains the global clock that all elements in the pipeline will use, in addition has a "base time", which is the clock time at the the point where media time is starting from zero. This timestamp is subctracted from the clock time, and that value is returned by `_get_time ()`.

The clock provider is responsible for making sure that the clock time always represents the current media time as closely as possible; it has to take care of things such as playback latencies, buffering in audio-kernel modules, and so on, since all those could affect a/v sync and thus decrease the user experience.

Clock slaves

Clock slaves get assigned a clock by their containing pipeline. Their task is to make sure that media playback follows the time progress as represented by this clock as closely as possible. For most elements, that will simply mean to wait until a certain time is reached before playing back their current sample; this can be done with the function `gst_clock_id_wait ()`. Some elements may need to support dropping samples too, however.

For more information on how to write elements that conform to this required behaviour, see the Plugin Writer's Guide.

Chapter 15. Dynamic Controllable Parameters

Getting Started

The controller subsystem offers a lightweight way to adjust gobject properties over stream-time. It works by using time-stamped value pairs that are queued for element-properties. At run-time the elements continously pull values changes for the current stream-time.

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

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

Your application should link to the shared library `gstreamer-controller`.

The `gstreamer-controller` library needs to be initialized when your application is run. This can be done after the the GStreamer library has been initialized.

```
  ...
  gst_init (&argc, &argv);
  gst_controller_init (&argc, &argv);
  ...
```

Setting up parameter control

The first step is to select the parameters that should be controlled. This returns a controller object that is needed to further adjust the behaviour.

```
  controller = gst_object_control_properties(object, "prop1", "prop2",...);
```

Next we can select an interpolation mode. This mode controls how inbetween values are determined. The controller subsystem can e.g. fill gaps by smoothing parameter changes. Each controllable GObject property can be interpolated differently.

```
  gst_controller_set_interpolation_mode(controller,"prop1",mode);
```

Finally one needs to set control points. These are time-stamped GValues. The values become active when the timestamp is reached. They still stay in the list. If e.g. the pipeline runs a loop (using a segmented seek), the control-curve gets repeated as well.

```
  gst_controller_set (controller, "prop1" ,0 * GST_SECOND, value1);
  gst_controller_set (controller, "prop1" ,1 * GST_SECOND, value2);
```

The controller subsystem has a builtin live-mode. Even though a parameter has timestamped control-values assigned one can change the GObject property through `g_object_set()`. This is highly useful when binding the GObject properties to GUI widgets. When the user adjusts the value with the widget, one can set the GOBject property and this remains active until the next timestamped value overrides. This also works with smoothed parameters.

Chapter 16. Threads

`GStreamer` is inherently multi-threaded, and is fully thread-safe. Most threading internals are hidden from the application, which should make application development easier. However, in some cases, applications may want to have influence on some parts of those. `GStreamer` allows applications to force the use of multiple threads over some parts of a pipeline.

When would you want to force a thread?

There are several reasons to force the use of threads. However, for performance reasons, you never want to use one thread for every element out there, since that will create some overhead. Let’s now list some situations where threads can be particularly useful:

- Data buffering, for example when dealing with network streams or when recording data from a live stream such as a video or audio card. Short hickups elsewhere in the pipeline will not cause data loss.

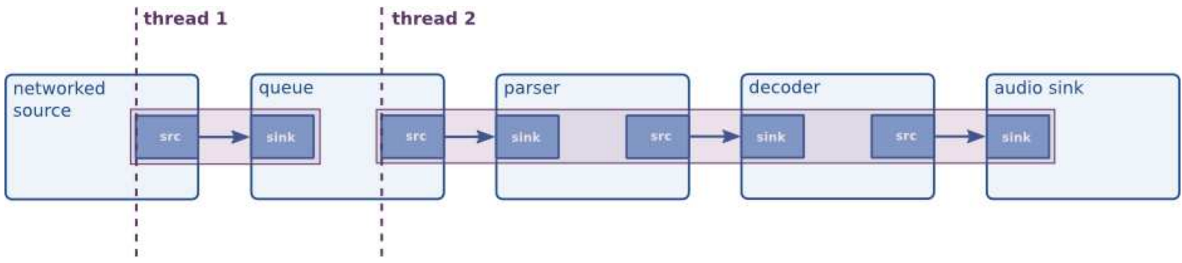

Figure 16-1. Data buffering, from a networked source

- Synchronizing output devices, e.g. when playing a stream containing both video and audio data. By using threads for both outputs, they will run independently and their synchronization will be better.

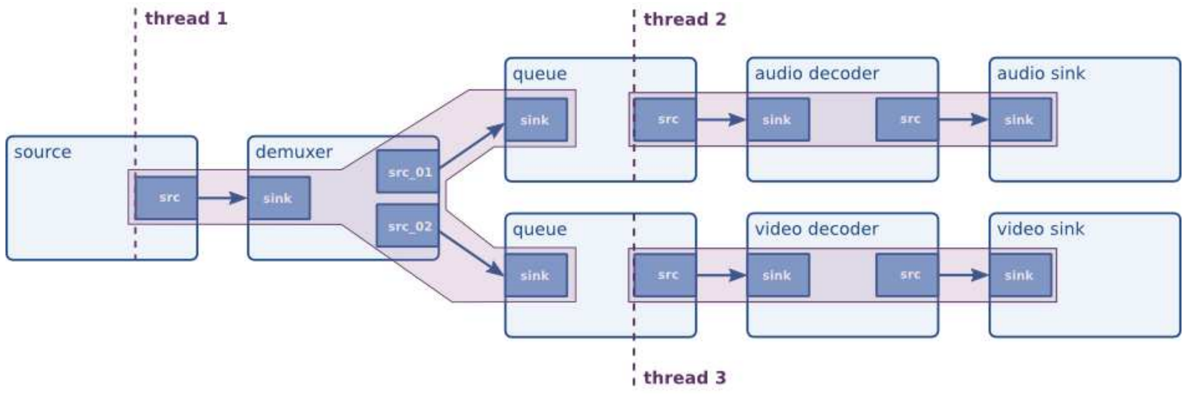

Figure 16-2. Synchronizing audio and video sinks

Above, we've mentioned the "queue" element several times now. A queue is the thread boundary element through which you can force the use of threads. It does so by using a classic provider/receiver model as learned in threading classes at universities all around the world. By doing this, it acts both as a means to make data throughput between threads threadsafe, and it can also act as a buffer. Queues have several `GObject` properties to be configured for specific uses. For example, you can set lower and upper tresholds for the element. If there's less data than the lower treshold (default: disabled), it will block output. If there's more data than the upper treshold, it will block input or (if configured to do so) drop data.

To use a queues (and therefore force the use of two distinct threads in the pipeline), one can simply create a "queue" element and put this in as part of the pipeline. `GStreamer` will take care of all threading details internally.

Scheduling in `GStreamer`

Scheduling of pipelines in `GStreamer` is done by using a thread for each "group", where a group is a set of elements separated by "queue" elements. Within such a group, scheduling is either push-based or pull-based, depending on which mode is supported by the particular element. If elements support random access to data, such as file sources, then elements downstream in the pipeline become the entry point of this group (i.e. the element controlling the scheduling of other elements). The entry point pulls data from upstream and pushes data downstream, thereby calling data handling functions on either type of element.

In practice, most elements in `GStreamer`, such as decoders, encoders, etc. only support push-based scheduling, which means that in practice, `GStreamer` uses a push-based scheduling model.

Chapter 17. Autoplugging

In Chapter 10, you've learned to build a simple media player for Ogg/Vorbis files. By using alternative elements, you are able to build media players for other media types, such as Ogg/Speex, MP3 or even video formats. However, you would rather want to build an application that can automatically detect the media type of a stream and automatically generate the best possible pipeline by looking at all available elements in a system. This process is called autoplugging, and `GStreamer` contains high-quality autopluggers. If you're looking for an autoplugger, don't read any further and go to Chapter 19. This chapter will explain the *concept* of autoplugging and typefinding. It will explain what systems `GStreamer` includes to dynamically detect the type of a media stream, and how to generate a pipeline of decoder elements to playback this media. The same principles can also be used for transcoding. Because of the full dynamicity of this concept, `GStreamer` can be automatically extended to support new media types without needing any adaptations to its autopluggers.

We will first introduce the concept of MIME types as a dynamic and extendible way of identifying media streams. After that, we will introduce the concept of typefinding to find the type of a media stream. Lastly, we will explain how autoplugging and the `GStreamer` registry can be used to setup a pipeline that will convert media from one mimetype to another, for example for media decoding.

MIME-types as a way to identity streams

We have previously introduced the concept of capabilities as a way for elements (or, rather, pads) to agree on a media type when streaming data from one element to the next (see the Section called *Capabilities of a pad* in Chapter 8). We have explained that a capability is a combination of a mimetype and a set of properties. For most container formats (those are the files that you will find on your hard disk; Ogg, for example, is a container format), no properties are needed to describe the stream. Only a MIME-type is needed. A full list of MIME-types and accompanying properties can be found in the Plugin Writer's Guide[1].

An element must associate a MIME-type to its source and sink pads when it is loaded into the system. `GStreamer` knows about the different elements and what type of data they expect and emit through the `GStreamer` registry. This allows for very dynamic and extensible element creation as we will see.

In Chapter 10, we've learned to build a music player for Ogg/Vorbis files. Let's look at the MIME-types associated with each pad in this pipeline. Figure 17-1 shows what MIME-type belongs to each pad in this pipeline.

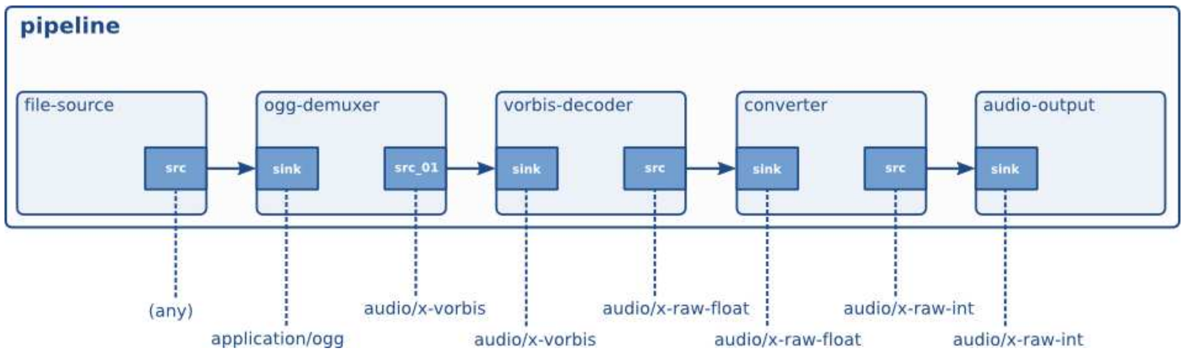

Figure 17-1. The Hello world pipeline with MIME types

Now that we have an idea how `GStreamer` identifies known media streams, we can look at methods `GStreamer` uses to setup pipelines for media handling and for media type detection.

Media stream type detection

Usually, when loading a media stream, the type of the stream is not known. This means that before we can choose a pipeline to decode the stream, we first need to detect the stream type. `GStreamer` uses the concept of typefinding for this. Typefinding is a normal part of a pipeline, it will read data for as long as the type of a stream is unknown. During this period, it will provide data to all plugins that implement a typefinder. when one of the typefinders recognizes the stream, the typefind element will emit a signal and act as a passthrough module from that point on. If no type was found, it will emit an error and further media processing will stop.

Once the typefind element has found a type, the application can use this to plug together a pipeline to decode the media stream. This will be discussed in the next section.

Plugins in `GStreamer` can, as mentioned before, implement typefinder functionality. A plugin implementing this functionality will submit a mimetype, optionally a set of file extensions commonly used for this media type, and a typefind function. Once this typefind function inside the plugin is called, the plugin will see if the data in this media stream matches a specific pattern that marks the media type identified by that mimetype. If it does, it will notify the typefind element of this fact, telling which mediatype was recognized and how certain we are that this stream is indeed that mediatype. Once this run has been completed for all plugins implementing a typefind functionality, the typefind element will tell the application what kind of media stream it thinks to have recognized.

The following code should explain how to use the typefind element. It will print the detected media type, or tell that the media type was not found. The next section will introduce more useful behaviours, such as plugging together a decoding pipeline.

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

[.. my_bus_callback goes here ..]
```

```
static gboolean
idle_exit_loop (gpointer data)
{
  g_main_loop_quit ((GMainLoop *) data);

  /* once */
  return FALSE;
}

static void
cb_typefound (GstElement *typefind,
       guint       probability,
       GstCaps    *caps,
       gpointer    data)
{
  GMainLoop *loop = data;
  gchar *type;

  type = gst_caps_to_string (caps);
  g_print ("Media type %s found, probability %d%%\n", type, probability);
  g_free (type);

  /* since we connect to a signal in the pipeline thread context, we need
   * to set an idle handler to exit the main loop in the mainloop context.
   * Normally, your app should not need to worry about such things. */
  g_idle_add (idle_exit_loop, loop);
}

gint
main (gint   argc,
      gchar *argv[])
{
  GMainLoop *loop;
  GstElement *pipeline, *filesrc, *typefind, *fakesink;
  GstBus *bus;

  /* init GStreamer */
  gst_init (&argc, &argv);
  loop = g_main_loop_new (NULL, FALSE);

  /* check args */
  if (argc != 2) {
    g_print ("Usage: %s <filename>\n", argv[0]);
    return -1;
  }

  /* create a new pipeline to hold the elements */
  pipeline = gst_pipeline_new ("pipe");

  bus = gst_pipeline_get_bus (GST_PIPELINE (pipeline));
  gst_bus_add_watch (bus, my_bus_callback, NULL);
  gst_object_unref (bus);

  /* create file source and typefind element */
  filesrc = gst_element_factory_make ("filesrc", "source");
  g_object_set (G_OBJECT (filesrc), "location", argv[1], NULL);
  typefind = gst_element_factory_make ("typefind", "typefinder");
  g_signal_connect (typefind, "have-type", G_CALLBACK (cb_typefound), loop);
  fakesink = gst_element_factory_make ("fakesink", "sink");

  /* setup */
  gst_bin_add_many (GST_BIN (pipeline), filesrc, typefind, fakesink, NULL);
  gst_element_link_many (filesrc, typefind, fakesink, NULL);
  gst_element_set_state (GST_ELEMENT (pipeline), GST_STATE_PLAYING);
  g_main_loop_run (loop);

  /* unset */
  gst_element_set_state (GST_ELEMENT (pipeline), GST_STATE_NULL);
  gst_object_unref (GST_OBJECT (pipeline));

  return 0;
}
```

Once a media type has been detected, you can plug an element (e.g. a demuxer or decoder) to the source pad of the typefind element, and decoding of the media stream will start right after.

Plugging together dynamic pipelines

Warning

The code in this section is broken, outdated and overly complicated. Also, you should use decodebin, playbin or uridecodebin to get decoders plugged automatically.

In this chapter we will see how you can create a dynamic pipeline. A dynamic pipeline is a pipeline that is updated or created while data is flowing through it. We will create a partial pipeline first and add more elements while the pipeline is playing. The basis of this player will be the application that we wrote in the previous section (the Section called *Media stream type detection*) to identify unknown media streams.

Once the type of the media has been found, we will find elements in the registry that can decode this streamtype. For this, we will get all element factories (which we've seen before in the Section called *Creating a `GstElement`* in Chapter 5) and find the ones with the given MIME-type and capabilities on their sinkpad. Note that we will only use parsers, demuxers and decoders. We will not use factories for any other element types, or we might get into a loop of encoders and decoders. For this, we will want to build a list of "allowed" factories right after initializing `GStreamer`.

```
static GList *factories;

/*
 * This function is called by the registry loader. Its return value
 * (TRUE or FALSE) decides whether the given feature will be included
 * in the list that we're generating further down.
 */
```

```
static gboolean
cb_feature_filter (GstPluginFeature *feature,
     gpointer          data)
{
  const gchar *klass;
  guint rank;

  /* we only care about element factories */
  if (!GST_IS_ELEMENT_FACTORY (feature))
    return FALSE;

  /* only parsers, demuxers and decoders */
  klass = gst_element_factory_get_klass (GST_ELEMENT_FACTORY (feature));
  if (g_strrstr (klass, "Demux") == NULL &&
      g_strrstr (klass, "Decoder") == NULL &&
      g_strrstr (klass, "Parse") == NULL)
    return FALSE;

  /* only select elements with autoplugging rank */
  rank = gst_plugin_feature_get_rank (feature);
  if (rank < GST_RANK_MARGINAL)
    return FALSE;

  return TRUE;
}

/*
 * This function is called to sort features by rank.
 */

static gint
cb_compare_ranks (GstPluginFeature *f1,
    GstPluginFeature *f2)
{
  return gst_plugin_feature_get_rank (f2) - gst_plugin_feature_get_rank (f1);
}

static void
init_factories (void)
{
  /* first filter out the interesting element factories */
  factories = gst_registry_feature_filter (
      gst_registry_get_default (),
      (GstPluginFeatureFilter) cb_feature_filter, FALSE, NULL);

  /* sort them according to their ranks */
  factories = g_list_sort (factories, (GCompareFunc) cb_compare_ranks);
}
```

From this list of element factories, we will select the one that most likely will help us decoding a media stream to a given output type. For each newly created element, we will again try to autoplug new elements to its source pad(s). Also, if the element has dynamic pads (which we've seen before in the Section called *Dynamic (or sometimes) pads* in Chapter 8), we will listen for newly created source pads and handle those, too. The following code replaces the `cb_type_found` from the previous section with a function to initiate autoplugging, which will continue with the above approach.

```
static void try_to_plug (GstPad *pad, const GstCaps *caps);

static GstElement *audiosink;

static void
cb_newpad (GstElement *element,
    GstPad     *pad,
    gpointer    data)
{
  GstCaps *caps;

  caps = gst_pad_get_caps (pad);
  try_to_plug (pad, caps);
  gst_caps_unref (caps);
}

static void
close_link (GstPad      *srcpad,
     GstElement  *sinkelement,
     const gchar *padname,
     const GList *templlist)
{
  GstPad *pad;
  gboolean has_dynamic_pads = FALSE;

  g_print ("Plugging pad %s:%s to newly created %s:%s\n",
    gst_object_get_name (GST_OBJECT (gst_pad_get_parent (srcpad))),
    gst_pad_get_name (srcpad),
    gst_object_get_name (GST_OBJECT (sinkelement)), padname);

  /* add the element to the pipeline and set correct state */
  if (sinkelement != audiosink) {
    gst_bin_add (GST_BIN (pipeline), sinkelement);
    gst_element_set_state (sinkelement, GST_STATE_READY);
  }
  pad = gst_element_get_static_pad (sinkelement, padname);
  gst_pad_link (srcpad, pad);
  if (sinkelement != audiosink) {
    gst_element_set_state (sinkelement, GST_STATE_PAUSED);
  }
  gst_object_unref (GST_OBJECT (pad));

  /* if we have static source pads, link those. If we have dynamic
   * source pads, listen for pad-added signals on the element */
  for ( ; templlist != NULL; templlist = templlist->next) {
    GstStaticPadTemplate *templ = templlist->data;

    /* only sourcepads, no request pads */
    if (templ->direction != GST_PAD_SRC ||
        templ->presence == GST_PAD_REQUEST) {
      continue;
    }
    switch (templ->presence) {
      case GST_PAD_ALWAYS: {
        GstPad *pad = gst_element_get_static_pad (sinkelement, templ->name_template)
        GstCaps *caps = gst_pad_get_caps (pad);

        /* link */
        try_to_plug (pad, caps);
        gst_object_unref (GST_OBJECT (pad));
        gst_caps_unref (caps);
        break;
      }
      case GST_PAD_SOMETIMES:
        has_dynamic_pads = TRUE;
        break;
      default:
        break;
    }
  }

  /* listen for newly created pads if this element supports that */
  if (has_dynamic_pads) {
    g_signal_connect (sinkelement, "pad-added", G_CALLBACK (cb_newpad), NULL);
  }
}

static void
try_to_plug (GstPad        *pad,
      const GstCaps *caps)
{
  GstObject *parent = GST_OBJECT (GST_OBJECT_PARENT (pad));
  const gchar *mime;
  const GList *item;
  GstCaps *res, *audiocaps;

  /* don't plug if we're already plugged - FIXME: memleak for pad */
  if (GST_PAD_IS_LINKED (gst_element_get_static_pad (audiosink, "sink"))) {
    g_print ("Omitting link for pad %s:%s because we're already linked\n",
      GST_OBJECT_NAME (parent), GST_OBJECT_NAME (pad));
    return;
  }

  /* as said above, we only try to plug audio... Omit video */
  mime = gst_structure_get_name (gst_caps_get_structure (caps, 0));
  if (g_strrstr (mime, "video")) {
    g_print ("Omitting link for pad %s:%s because mimetype %s is non-audio\n",
      GST_OBJECT_NAME (parent), GST_OBJECT_NAME (pad), mime);
    return;
  }

  /* can it link to the audiopad? */
  audiocaps = gst_pad_get_caps (gst_element_get_static_pad (audiosink, "sink"));
  res = gst_caps_intersect (caps, audiocaps);
  if (res && !gst_caps_is_empty (res)) {
    g_print ("Found pad to link to audiosink - plugging is now done\n");
    close_link (pad, audiosink, "sink", NULL);
```

```
      gst_caps_unref (audiocaps);
      gst_caps_unref (res);
      return;
    }
    gst_caps_unref (audiocaps);
    gst_caps_unref (res);

    /* try to plug from our list */
    for (item = factories; item != NULL; item = item->next) {
      GstElementFactory *factory = GST_ELEMENT_FACTORY (item->data);
      const GList *pads;

      for (pads = gst_element_factory_get_static_pad_templates (factory);
           pads != NULL; pads = pads->next) {
        GstStaticPadTemplate *templ = pads->data;

        /* find the sink template - need an always pad*/
        if (templ->direction != GST_PAD_SINK ||
            templ->presence != GST_PAD_ALWAYS) {
          continue;
        }

        /* can it link? */
        res = gst_caps_intersect (caps,
            gst_static_caps_get (&templ->static_caps));
        if (res && !gst_caps_is_empty (res)) {
          GstElement *element;
          gchar *name_template = g_strdup (templ->name_template);

          /* close link and return */
          gst_caps_unref (res);
          element = gst_element_factory_create (factory, NULL);
          close_link (pad, element, name_template,
        gst_element_factory_get_static_pad_templates (factory));
          g_free (name_template);
          return;
        }
        gst_caps_unref (res);

        /* we only check one sink template per factory, so move on to the
         * next factory now */
        break;
      }
    }

    /* if we get here, no item was found */
    g_print ("No compatible pad found to decode %s on %s:%s\n",
      mime, GST_OBJECT_NAME (parent), GST_OBJECT_NAME (pad));
  }

  static void
  cb_typefound (GstElement *typefind,
         guint       probability,
         GstCaps    *caps,
         gpointer    data)
  {
```

```
  gchar *s;
  GstPad *pad;

  s = gst_caps_to_string (caps);
  g_print ("Detected media type %s\n", s);
  g_free (s);

  /* actually plug now */
  pad = gst_element_get_static_pad (typefind, "src");
  try_to_plug (pad, caps);
  gst_object_unref (GST_OBJECT (pad));
}
```

By doing all this, we will be able to make a simple autoplugger that can automatically setup a pipeline for any media type. In the example below, we will do this for audio only. However, we can also do this for video to create a player that plays both audio and video.

The example above is a good first try for an autoplugger. Next steps would be to listen for "pad-removed" signals, so we can dynamically change the plugged pipeline if the stream changes (this happens for DVB or Ogg radio). Also, you might want special-case code for input with known content (such as a DVD or an audio-CD), and much, much more. Moreover, you'll want many checks to prevent infinite loops during autoplugging, maybe you'll want to implement shortest-path-finding to make sure the most optimal pipeline is chosen, and so on. Basically, the features that you implement in an autoplugger depend on what you want to use it for. For full-blown implementations, see the "playbin" and "decodebin" elements.

Notes

1. http://gstreamer.freedesktop.org/data/doc/gstreamer/head/pwg/html/section-types-definitions.html

Chapter 18. Pipeline manipulation

This chapter will discuss how you can manipulate your pipeline in several ways from your application on. Parts of this chapter are downright hackish, so be assured that you'll need some programming knowledge before you start reading this.

Topics that will be discussed here include how you can insert data into a pipeline from your application, how to read data from a pipeline, how to manipulate the pipeline's speed, length, starting point and how to listen to a pipeline's data processing.

Data probing

Probing is best envisioned as a pad listener. Technically, a probe is nothing more than a signal callback that can be attached to a pad. Those signals are by default not fired at all (since that may have a negative impact on performance), but can be enabled by attaching a probe using `gst_pad_add_buffer_probe ()`, `gst_pad_add_event_probe ()`, or `gst_pad_add_data_probe ()`. Those functions attach the signal handler and enable the actual signal emission. Similarly, one can use the `gst_pad_remove_buffer_probe ()`, `gst_pad_remove_event_probe ()`, or `gst_pad_remove_data_probe ()` to remove the signal handlers again.

Probes run in pipeline threading context, so callbacks should try to not block and generally not do any weird stuff, since this could have a negative impact on pipeline performance or, in case of bugs, cause deadlocks or crashes. More precisely, one should usually not call any GUI-related functions from within a probe callback, nor try to change the state of the pipeline. An application may post custom messages on the pipeline's bus though to communicate with the main application thread and have it do things like stop the pipeline.

In any case, most common buffer operations that elements can do in `_chain ()` functions, can be done in probe callbacks as well. The example below gives a short impression on how to use them (even if this usage is not entirely correct, but more on that below):

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

static gboolean
cb_have_data (GstPad    *pad,
       GstBuffer *buffer,
       gpointer   u_data)
{
  gint x, y;
  guint16 *data = (guint16 *) GST_BUFFER_DATA (buffer), t;

  /* invert data */
  for (y = 0; y < 288; y++) {
    for (x = 0; x < 384 / 2; x++) {
      t = data[384 - 1 - x];
      data[384 - 1 - x] = data[x];
      data[x] = t;
    }
    data += 384;
  }
  return TRUE;
}

gint
main (gint   argc,
      gchar *argv[])
{
  GMainLoop *loop;
  GstElement *pipeline, *src, *sink, *filter, *csp;
  GstCaps *filtercaps;
  GstPad *pad;

  /* init GStreamer */
  gst_init (&argc, &argv);
  loop = g_main_loop_new (NULL, FALSE);

  /* build */
  pipeline = gst_pipeline_new ("my-pipeline");
  src = gst_element_factory_make ("videotestsrc", "src");
  if (src == NULL)
    g_error ("Could not create 'videotestsrc' element");

  filter = gst_element_factory_make ("capsfilter", "filter");
  g_assert (filter != NULL); /* should always exist */

  csp = gst_element_factory_make ("ffmpegcolorspace", "csp");
  if (csp == NULL)
    g_error ("Could not create 'ffmpegcolorspace' element");

  sink = gst_element_factory_make ("xvimagesink", "sink");
  if (sink == NULL) {
    sink = gst_element_factory_make ("ximagesink", "sink");
    if (sink == NULL)
      g_error ("Could not create neither 'xvimagesink' nor 'ximagesink' element");
  }

  gst_bin_add_many (GST_BIN (pipeline), src, filter, csp, sink, NULL);
  gst_element_link_many (src, filter, csp, sink, NULL);
  filtercaps = gst_caps_new_simple ("video/x-raw-rgb",
      "width", G_TYPE_INT, 384,
      "height", G_TYPE_INT, 288,
      "framerate", GST_TYPE_FRACTION, 25, 1,
      "bpp", G_TYPE_INT, 16,
      "depth", G_TYPE_INT, 16,
      "endianness", G_TYPE_INT, G_BYTE_ORDER,
      NULL);
  g_object_set (G_OBJECT (filter), "caps", filtercaps, NULL);
  gst_caps_unref (filtercaps);

  pad = gst_element_get_pad (src, "src");
  gst_pad_add_buffer_probe (pad, G_CALLBACK (cb_have_data), NULL);
  gst_object_unref (pad);

  /* run */
  gst_element_set_state (pipeline, GST_STATE_PLAYING);
  /* wait until it’s up and running or failed */
  if (gst_element_get_state (pipeline, NULL, NULL, -1) == GST_STATE_CHANGE_FAILURE)
    g_error ("Failed to go into PLAYING state");
  }

  g_print ("Running ...\n");
  g_main_loop_run (loop);

  /* exit */
  gst_element_set_state (pipeline, GST_STATE_NULL);
  gst_object_unref (pipeline);

  return 0;
}
```

Compare that output with the output of “gst-launch-0.10 videotestsrc ! xvimagesink”, just so you know what you’re looking for.

The above example is not really correct though. Strictly speaking, a pad probe callback is only allowed to modify the buffer content if the buffer is writable, and it is only allowed to modify buffer metadata like timestamps, caps, etc. if the buffer metadata is writable. Whether this is the case or not depends a lot on the pipeline and the elements involved. Often enough, this is the case, but sometimes it is not, and if it is not then unexpected modification of the data or metadata can introduce bugs that are very hard to debug and track down. You can check if a buffer and its metadata are writable with `gst_buffer_is_writable ()` and `gst_buffer_is_metadata_writable ()`. Since you can’t pass back a different buffer than the one passed in, there is no point of making a buffer writable in the callback function.

Pad probes are suited best for looking at data as it passes through the pipeline. If you need to modify data, you should write your own GStreamer element. Base classes like GstAudioFilter, GstVideoFilter or GstBaseTransform make this fairly easy.

If you just want to inspect buffers as they pass through the pipeline, you don’t even need to set up pad probes. You could also just insert an identity element into the pipeline and connect to its "handoff" signal. The identity element also provides a few useful debugging tools like the "dump" property or the "last-message" property (the latter is enabled by passing the ’-v’ switch to gst-launch).

Manually adding or removing data from/to a pipeline

Many people have expressed the wish to use their own sources to inject data into a pipeline. Some people have also expressed the wish to grab the output in a pipeline and take care of the actual output inside their application. While either of these methods are stongly discouraged, `GStreamer` offers hacks to do this. *However, there is no support for those methods.* If it doesn’t work, you’re on your own. Also, synchronization, thread-safety and other things that you’ve been able to take for granted so far are no longer guaranteed if you use any of those methods. It’s always better to simply write a plugin and have the pipeline schedule and manage it. See the Plugin

Writer's Guide for more information on this topic. Also see the next section, which will explain how to embed plugins statically in your application.

> **Note:** New API is being developed at the moment to make data insertion and extraction less painful for applications. It can be found as GstAppSrc and GstAppSink in the gst-plugins-bad module. At the time of writing (October 2007), this API is not quite stable and ready yet, even though it may work fine for your purposes.

After all those disclaimers, let's start. There's three possible elements that you can use for the above-mentioned purposes. Those are called "fakesrc" (an imaginary source), "fakesink" (an imaginary sink) and "identity" (an imaginary filter). The same method applies to each of those elements. Here, we will discuss how to use those elements to insert (using fakesrc) or grab (using fakesink or identity) data from a pipeline, and how to set negotiation.

Those who're paying close attention, will notice that the purpose of identity is almost identical to that of probes. Indeed, this is true. Probes allow for the same purpose, and a bunch more, and with less overhead plus dynamic removing/adding of handlers, but apart from those, probes and identity have the same purpose, just in a completely different implementation type.

Inserting or grabbing data

The three before-mentioned elements (fakesrc, fakesink and identity) each have a "handoff" signal that will be called in the `_get ()`- (fakesrc) or `_chain ()`-function (identity, fakesink). In the signal handler, you can set (fakesrc) or get (identity, fakesink) data to/from the provided buffer. Note that in the case of fakesrc, you have to set the size of the provided buffer using the "sizemax" property. For both fakesrc and fakesink, you also have to set the "signal-handoffs" property for this method to work.

Note that your handoff function should *not* block, since this will block pipeline iteration. Also, do not try to use all sort of weird hacks in such functions to accomplish something that looks like synchronization or so; it's not the right way and will lead to issues elsewhere. If you're doing any of this, you're basically misunderstanding the `GStreamer` design.

Forcing a format

Sometimes, when using fakesrc as a source in your pipeline, you'll want to set a specific format, for example a video size and format or an audio bitsize and number of channels. You can do this by forcing a specific `GstCaps` on the pipeline, which is possible by using *filtered caps*. You can set a filtered caps on a link by using the "capsfilter" element in between the two elements, and specifying a `GstCaps` as "caps" property on this element. It will then only allow types matching that specified capability set for negotiation.

Example application

This example application will generate black/white (it switches every second) video to an X-window output by using fakesrc as a source and using filtered caps to force a format. Since the depth of the image depends on your X-server settings, we use a colorspace conversion element to make sure that the output to your X server will have the correct bitdepth. You can also set timestamps on the provided buffers to override the fixed framerate.

```
#include <string.h> /* for memset () */
#include <gst/gst.h>

static void
cb_handoff (GstElement *fakesrc,
     GstBuffer  *buffer,
     GstPad     *pad,
     gpointer    user_data)
{
  static gboolean white = FALSE;

  /* this makes the image black/white */
  memset (GST_BUFFER_DATA (buffer), white ? 0xff : 0x0,
   GST_BUFFER_SIZE (buffer));
  white = !white;
}

gint
main (gint   argc,
      gchar *argv[])
{
  GstElement *pipeline, *fakesrc, *flt, *conv, *videosink;
  GMainLoop *loop;

  /* init GStreamer */
  gst_init (&argc, &argv);
  loop = g_main_loop_new (NULL, FALSE);

  /* setup pipeline */
  pipeline = gst_pipeline_new ("pipeline");
  fakesrc = gst_element_factory_make ("fakesrc", "source");
  flt = gst_element_factory_make ("capsfilter", "flt");
  conv = gst_element_factory_make ("ffmpegcolorspace", "conv");
  videosink = gst_element_factory_make ("xvimagesink", "videosink");

  /* setup */
  g_object_set (G_OBJECT (flt), "caps",
    gst_caps_new_simple ("video/x-raw-rgb",
         "width", G_TYPE_INT, 384,
         "height", G_TYPE_INT, 288,
         "framerate", GST_TYPE_FRACTION, 1, 1,
         "bpp", G_TYPE_INT, 16,
         "depth", G_TYPE_INT, 16,
         "endianness", G_TYPE_INT, G_BYTE_ORDER,
         NULL), NULL);
  gst_bin_add_many (GST_BIN (pipeline), fakesrc, flt, conv, videosink, NULL);
  gst_element_link_many (fakesrc, flt, conv, videosink, NULL);
  /* setup fake source */
  g_object_set (G_OBJECT (fakesrc),
  "signal-handoffs", TRUE,
  "sizemax", 384 * 288 * 2,
  "sizetype", 2, NULL);
  g_signal_connect (fakesrc, "handoff", G_CALLBACK (cb_handoff), NULL);

  /* play */
  gst_element_set_state (pipeline, GST_STATE_PLAYING);
  g_main_loop_run (loop);

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

  return 0;
}
```

Embedding static elements in your application

The Plugin Writer's Guide[1] describes in great detail how to write elements for the `GStreamer` framework. In this section, we will solely discuss how to embed such elements statically in your application. This can be useful for application-specific elements that have no use elsewhere in `GStreamer`.

Dynamically loaded plugins contain a structure that's defined using `GST_PLUGIN_DEFINE ()`. This structure is loaded when the plugin is loaded by the `GStreamer` core. The structure contains an initialization function (usually called `plugin_init`) that will be called right after that. It's purpose is to register the elements provided by the plugin with the `GStreamer` framework. If you want to embed elements directly in your application, the only thing you need to do is to replace `GST_PLUGIN_DEFINE ()` with `GST_PLUGIN_DEFINE_STATIC ()`. This will cause the elements to be registered when your application loads, and the elements will from then on be available like any other element, without them having to be dynamically loadable libraries. In the example below, you would be able to call `gst_element_factory_make ("my-element-name", "some-name")` to create an instance of the element.

```
/*
 * Here, you would write the actual plugin code.
 */

[..]

static gboolean
register_elements (GstPlugin *plugin)
{
  return gst_element_register (plugin, "my-element-name",
          GST_RANK_NONE, MY_PLUGIN_TYPE);
}
```

```
GST_PLUGIN_DEFINE_STATIC (
  GST_VERSION_MAJOR,
  GST_VERSION_MINOR,
  "my-private-plugins",
  "Private elements of my application",
  register_elements,
  VERSION,
  "LGPL",
  "my-application",
  "http://www.my-application.net/"
)
```

Notes

1. http://gstreamer.freedesktop.org/data/doc/gstreamer/head/pwg/html/index.html

Chapter 19. Components

GStreamer includes several higher-level components to simplify an application developer's life. All of the components discussed here (for now) are targetted at media playback. The idea of each of these components is to integrate as closely as possible with a GStreamer pipeline, but to hide the complexity of media type detection and several other rather complex topics that have been discussed in Part III in *GStreamer Application Development Manual (0.10.23)*.

We currently recommend people to use either playbin (see the Section called *Playbin*) or decodebin (see the Section called *Decodebin*), depending on their needs. Playbin is the recommended solution for everything related to simple playback of media that should just work. Decodebin is a more flexible autoplugger that could be used to add more advanced features, such as playlist support, crossfading of audio tracks and so on. Its programming interface is more low-level than that of playbin, though.

Playbin

Playbin is an element that can be created using the standard GStreamer API (e.g. gst_element_factory_make ()). The factory is conveniently called "playbin". By being a GstPipeline (and thus a GstElement), playbin automatically supports all of the features of this class, including error handling, tag support, state handling, getting stream positions, seeking, and so on.

Setting up a playbin pipeline is as simple as creating an instance of the playbin element, setting a file location using the "uri" property on playbin, and then setting the element to the GST_STATE_PLAYING state (the location has to be a valid URI, so "<protocol>://<location>", e.g. file:///tmp/my.ogg or http://www.example.org/stream.ogg). Internally, playbin will set up a pipeline to playback the media location.

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

[.. my_bus_callback goes here ..]

gint
main (gint   argc,
      gchar *argv[])
{
  GMainLoop *loop;
  GstElement *play;
  GstBus *bus;

  /* init GStreamer */
  gst_init (&argc, &argv);
  loop = g_main_loop_new (NULL, FALSE);

  /* make sure we have a URI */
  if (argc != 2) {
    g_print ("Usage: %s <URI>\n", argv[0]);
    return -1;
  }
  /* set up */
  play = gst_element_factory_make ("playbin", "play");
  g_object_set (G_OBJECT (play), "uri", argv[1], NULL);

  bus = gst_pipeline_get_bus (GST_PIPELINE (play));
  gst_bus_add_watch (bus, my_bus_callback, loop);
  gst_object_unref (bus);

  gst_element_set_state (play, GST_STATE_PLAYING);

  /* now run */
  g_main_loop_run (loop);

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

  return 0;
}
```

Playbin has several features that have been discussed previously:

- Settable video and audio output (using the “video-sink” and “audio-sink” properties).
- Mostly controllable and trackable as a `GstElement`, including error handling, eos handling, tag handling, state handling (through the `GstBus`), media position handling and seeking.
- Buffers network-sources, with buffer fullness notifications being passed through the `GstBus`.
- Supports visualizations for audio-only media.
- Supports subtitles, both in the media as well as from separate files. For separate subtitle files, use the “suburi” property.
- Supports stream selection and disabling. If your media has multiple audio or subtitle tracks, you can dynamically choose which one to play back, or decide to turn it off alltogther (which is especially useful to turn off subtitles). For each of those, use the “current-text” and other related properties.

For convenience, it is possible to test “playbin” on the commandline, using the command “gst-launch-0.10 playbin uri=file:///path/to/file”.

Decodebin

Decodebin is the actual autoplugger backend of playbin, which was discussed in the previous section. Decodebin will, in short, accept input from a source that is linked to its sinkpad and will try to detect the media type contained in the stream, and set up decoder routines for each of those. It will automatically select decoders. For each decoded stream, it will emit the “new-decoded-pad” signal, to let the client know about the newly found decoded stream. For unknown streams (which might be the whole stream), it will emit the "unknown-type" signal. The application is then responsible for reporting the error to the user.

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

[.. my_bus_callback goes here ..]

GstElement *pipeline, *audio;

static void
cb_newpad (GstElement *decodebin,
    GstPad     *pad,
    gboolean    last,
    gpointer    data)
{
  GstCaps *caps;
  GstStructure *str;
  GstPad *audiopad;

  /* only link once */
  audiopad = gst_element_get_static_pad (audio, "sink");
  if (GST_PAD_IS_LINKED (audiopad)) {
    g_object_unref (audiopad);
    return;
  }

  /* check media type */
  caps = gst_pad_get_caps (pad);
  str = gst_caps_get_structure (caps, 0);
  if (!g_strrstr (gst_structure_get_name (str), "audio")) {
    gst_caps_unref (caps);
    gst_object_unref (audiopad);
    return;
  }
  gst_caps_unref (caps);

  /* link'n'play */
  gst_pad_link (pad, audiopad);
}

gint
main (gint   argc,
      gchar *argv[])
{
  GMainLoop *loop;
  GstElement *src, *dec, *conv, *sink;
  GstPad *audiopad;
  GstBus *bus;

  /* init GStreamer */
  gst_init (&argc, &argv);
  loop = g_main_loop_new (NULL, FALSE);

  /* make sure we have input */
  if (argc != 2) {
    g_print ("Usage: %s <filename>\n", argv[0]);
    return -1;
  }

  /* setup */
  pipeline = gst_pipeline_new ("pipeline");

  bus = gst_pipeline_get_bus (GST_PIPELINE (pipeline));
  gst_bus_add_watch (bus, my_bus_callback, loop);
  gst_object_unref (bus);

  src = gst_element_factory_make ("filesrc", "source");
  g_object_set (G_OBJECT (src), "location", argv[1], NULL);
  dec = gst_element_factory_make ("decodebin", "decoder");
  g_signal_connect (dec, "new-decoded-pad", G_CALLBACK (cb_newpad), NULL);
  gst_bin_add_many (GST_BIN (pipeline), src, dec, NULL);
  gst_element_link (src, dec);

  /* create audio output */
  audio = gst_bin_new ("audiobin");
  conv = gst_element_factory_make ("audioconvert", "aconv");
  audiopad = gst_element_get_static_pad (conv, "sink");
  sink = gst_element_factory_make ("alsasink", "sink");
  gst_bin_add_many (GST_BIN (audio), conv, sink, NULL);
  gst_element_link (conv, sink);
  gst_element_add_pad (audio,
      gst_ghost_pad_new ("sink", audiopad));
  gst_object_unref (audiopad);
  gst_bin_add (GST_BIN (pipeline), audio);

  /* run */
  gst_element_set_state (pipeline, GST_STATE_PLAYING);
  g_main_loop_run (loop);

  /* cleanup */
  gst_element_set_state (pipeline, GST_STATE_NULL);
  gst_object_unref (GST_OBJECT (pipeline));

  return 0;
}
```

Decodebin, similar to playbin, supports the following features:

- Can decode an unlimited number of contained streams to decoded output pads.
- Is handled as a `GstElement` in all ways, including tag or error forwarding and state handling.

Although decodebin is a good autoplugger, there's a whole lot of things that it does not do and is not intended to do:

- Taking care of input streams with a known media type (e.g. a DVD, an audio-CD or such).
- Selection of streams (e.g. which audio track to play in case of multi-language media streams).

- Overlaying subtitles over a decoded video stream.

Decodebin can be easily tested on the commandline, e.g. by using the command **gst-launch-0.8 filesrc location=file.ogg ! decodebin ! audioconvert ! alsasink**.

Chapter 20. XML in `GStreamer`

`GStreamer` can use XML to store and load its pipeline definitions.

We will show you how you can save a pipeline to XML and how you can reload that XML file again for later use.

Turning GstElements into XML

We create a simple pipeline and write it to stdout with gst_xml_write_file (). The following code constructs an MP3 player pipeline and then writes out the XML both to stdout and to a file. Use this program with one argument: the MP3 file on disk.

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

gboolean playing;

int
main (int argc, char *argv[])
{
  GstElement *filesrc, *osssink, *decode;
  GstElement *pipeline;

  gst_init (&argc,&argv);

  if (argc != 2) {
    g_print ("usage: %s <mp3 filename>\n", argv[0]);
    exit (-1);
  }

  /* create a new pipeline to hold the elements */
  pipeline = gst_element_factory_make ("pipeline", "pipeline");
  g_assert (pipeline != NULL);

  /* create a disk reader */
  filesrc = gst_element_factory_make ("filesrc", "disk_source");
  g_assert (filesrc != NULL);
  g_object_set (G_OBJECT (filesrc), "location", argv[1], NULL);

  /* and an audio sink */
  osssink = gst_element_factory_make ("osssink", "play_audio");
  g_assert (osssink != NULL);

  decode = gst_element_factory_make ("mad", "decode");
  g_assert (decode != NULL);

  /* add objects to the main pipeline */
  gst_bin_add_many (GST_BIN (pipeline), filesrc, decode, osssink, NULL);

  gst_element_link_many (filesrc, decode, osssink, NULL);

  /* write the pipeline to stdout */
  gst_xml_write_file (GST_ELEMENT (pipeline), stdout);
  /* write the bin to a file */
  gst_xml_write_file (GST_ELEMENT (pipeline), fopen ("xmlTest.gst", "w"));

  exit (0);
}
```

The most important line is:

```
  gst_xml_write_file (GST_ELEMENT (pipeline), stdout);
```

gst_xml_write_file () will turn the given element into an xmlDocPtr that is then formatted and saved to a file. To save to disk, pass the result of a fopen(2) as the second argument.

The complete element hierarchy will be saved along with the inter element pad links and the element parameters. Future `GStreamer` versions will also allow you to store the signals in the XML file.

Loading a GstElement from an XML file

Before an XML file can be loaded, you must create a GstXML object. A saved XML file can then be loaded with the gst_xml_parse_file (xml, filename, rootelement) method. The root element can optionally left NULL. The following code example loads the previously created XML file and runs it.

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

int
main(int argc, char *argv[])
{
  GstXML *xml;
  GstElement *pipeline;
  gboolean ret;

  gst_init (&argc, &argv);

  xml = gst_xml_new ();

  ret = gst_xml_parse_file(xml, "xmlTest.gst", NULL);
  g_assert (ret == TRUE);

  pipeline = gst_xml_get_element (xml, "pipeline");
  g_assert (pipeline != NULL);

  gst_element_set_state (pipeline, GST_STATE_PLAYING);

  g_sleep (4);

  gst_element_set_state (pipeline, GST_STATE_NULL);
  exit (0);
}
```

gst_xml_get_element (xml, "name") can be used to get a specific element from the XML file.

gst_xml_get_topelements (xml) can be used to get a list of all toplevel elements in the XML file.

In addition to loading a file, you can also load from a xmlDocPtr and an in-memory buffer using gst_xml_parse_doc and gst_xml_parse_memory respectively. Both of these methods return a gboolean indicating success or failure of the requested action.

Adding custom XML tags into the core XML data

It is possible to add custom XML tags to the core XML created with gst_xml_write. This feature can be used by an application to add more information to the save plugins. The editor will for example insert the position of the elements on the screen using the custom XML tags.

It is strongly suggested to save and load the custom XML tags using a namespace. This will solve the problem of having your XML tags interfere with the core XML tags.

To insert a hook into the element saving procedure you can link a signal to the GstElement using the following piece of code:

```
xmlNsPtr ns;

  ...
  ns = xmlNewNs (NULL, "http://gstreamer.net/gst-test/1.0/", "test");
    ...
  pipeline = gst_element_factory_make ("pipeline", "pipeline");
  g_signal_connect (G_OBJECT (pipeline), "object_saved",
         G_CALLBACK (object_saved), g_strdup ("decoder pipeline"));
    ...
```

When the thread is saved, the object_save method will be called. Our example will insert a comment tag:

```
static void
object_saved (GstObject *object, xmlNodePtr parent, gpointer data)
{
  xmlNodePtr child;

  child = xmlNewChild (parent, ns, "comment", NULL);
  xmlNewChild (child, ns, "text", (gchar *)data);
}
```

Adding the custom tag code to the above example you will get an XML file with the custom tags in it. Here's an excerpt:

```
        ...
      <gst:element>
        <gst:name>pipeline</gst:name>
        <gst:type>pipeline</gst:type>
        <gst:version>0.1.0</gst:version>
 ...
      </gst:children>
      <test:comment>
        <test:text>decoder pipeline</test:text>
      </test:comment>
    </gst:element>
        ...
```

To retrieve the custom XML again, you need to attach a signal to the GstXML object used to load the XML data. You can then parse your custom XML from the XML tree whenever an object is loaded.

We can extend our previous example with the following piece of code.

```
  xml = gst_xml_new ();

  g_signal_connect (G_OBJECT (xml), "object_loaded",
         G_CALLBACK (xml_loaded), xml);

  ret = gst_xml_parse_file (xml, "xmlTest.gst", NULL);
  g_assert (ret == TRUE);
```

Whenever a new object has been loaded, the xml_loaded function will be called. This function looks like:

```
static void
xml_loaded (GstXML *xml, GstObject *object, xmlNodePtr self, gpointer data)
{
  xmlNodePtr children = self->xmlChildrenNode;

  while (children) {
    if (!strcmp (children->name, "comment")) {
      xmlNodePtr nodes = children->xmlChildrenNode;

      while (nodes) {
        if (!strcmp (nodes->name, "text")) {
          gchar *name = g_strdup (xmlNodeGetContent (nodes));
          g_print ("object %s loaded with comment '%s'\n",
                   gst_object_get_name (object), name);
        }
        nodes = nodes->next;
      }
    }
    children = children->next;
  }
}
```

As you can see, you'll get a handle to the GstXML object, the newly loaded GstObject and the xmlNodePtr that was used to create this object. In the above example we look for our special tag inside the XML tree that was used to load the object and we print our comment to the console.

Chapter 21. Things to check when writing an application

This chapter contains a fairly random selection of things that can be useful to keep in mind when writing `GStreamer`-based applications. It's up to you how much you're going to use the information provided here. We will shortly discuss how to debug pipeline problems using `GStreamer` applications. Also, we will touch upon how to acquire knowledge about plugins and elements and how to test simple pipelines before building applications around them.

Good programming habits

- Always add a `GstBus` handler to your pipeline. Always report errors in your application, and try to do something with warnings and information messages, too.
- Always check return values of `GStreamer` functions. Especially, check return values of `gst_element_link ()` and `gst_element_set_state ()`.
- Dereference return values of all functions returning a non-base type, such as `gst_element_get_pad ()`. Also, always free non-const string returns, such as `gst_object_get_name ()`.
- Always use your pipeline object to keep track of the current state of your pipeline. Don't keep private variables in your application. Also, don't update your user interface if a user presses the "play" button. Instead, listen for the "state-changed" message on the `GstBus` and only update the user interface whenever this message is received.
- Report all bugs that you find in `GStreamer` bugzilla at http://bugzilla.gnome.org/[1].

Debugging

Applications can make use of the extensive `GStreamer` debugging system to debug pipeline problems. Elements will write output to this system to log what they're doing. It's not used for error reporting, but it is very useful for tracking what an element is doing exactly, which can come in handy when debugging application issues (such as failing seeks, out-of-sync media, etc.).

Most `GStreamer`-based applications accept the commandline option `--gst-debug=LIST` and related family members. The list consists of a comma-separated list of category/level pairs, which can set the debugging level for a specific debugging category. For example, `--gst-debug=oggdemux:5` would turn on debugging for the Ogg demuxer element. You can use wildcards as well. A debugging level of 0 will turn off all debugging, and a level of 5 will turn on all debugging. Intermediate values only turn on some debugging (based on message severity; 2, for example, will only display errors and warnings). Here's a list of all available options:

- `--gst-debug-help` will print available debug categories and exit.

- `--gst-debug-level=`*`LEVEL`* will set the default debug level (which can range from 0 (no output) to 5 (everything)).
- `--gst-debug=`*`LIST`* takes a comma-separated list of category_name:level pairs to set specific levels for the individual categories. Example: `GST_AUTOPLUG:5,avidemux:3`. Alternatively, you can also set the `GST_DEBUG` environment variable, which has the same effect.
- `--gst-debug-no-color` will disable color debugging You can also set the GST_DEBUG_NO_COLOR environment variable to 1 if you want to disable colored debug output permanently. Note that if you are disabling color purely to avoid messing up your pager output, trying using **less -R**.
- `--gst-debug-disable` disables debugging altogether.
- `--gst-plugin-spew` enables printout of errors while loading `GStreamer` plugins.

Conversion plugins

`GStreamer` contains a bunch of conversion plugins that most applications will find useful. Specifically, those are videoscalers (videoscale), colorspace convertors (ffmpegcolorspace), audio format convertors and channel resamplers (audioconvert) and audio samplerate convertors (audioresample). Those convertors don't do anything when not required, they will act in passthrough mode. They will activate when the hardware doesn't support a specific request, though. All applications are recommended to use those elements.

Utility applications provided with `GStreamer`

`GStreamer` comes with a default set of command-line utilities that can help in application development. We will discuss only **gst-launch** and **gst-inspect** here.

gst-launch

gst-launch is a simple script-like commandline application that can be used to test pipelines. For example, the command **gst-launch audiotestsrc ! audioconvert ! audio/x-raw-int,channels=2 ! alsasink** will run a pipeline which generates a sine-wave audio stream and plays it to your ALSA audio card. **gst-launch** also allows the use of threads (will be used automatically as required or as queue elements are inserted in the pipeline) and bins (using brackets, so "(" and ")"). You can use dots to imply padnames on elements, or even omit the padname to automatically select a pad. Using all this, the pipeline **gst-launch filesrc location=file.ogg ! oggdemux name=d d. ! queue ! theoradec ! ffmpegcolorspace ! xvimagesink d. ! queue ! vorbisdec ! audioconvert ! audioresample ! alsasink** will play an Ogg file containing a Theora video-stream and a Vorbis audio-stream. You can also use autopluggers such as decodebin on the commandline. See the manual page of **gst-launch** for more information.

gst-inspect

gst-inspect can be used to inspect all properties, signals, dynamic parameters and the object hierarchy of an element. This can be very useful to see which `GObject` properties or which signals (and using what arguments) an element supports. Run **gst-inspect fakesrc** to get an idea of what it does. See the manual page of **gst-inspect** for more information.

GstEditor

GstEditor is a set of widgets to display a graphical representation of a pipeline.

Notes

1. http://bugzilla.gnome.org

Chapter 22. Porting 0.8 applications to 0.10

This section of the appendix will discuss shortly what changes to applications will be needed to quickly and conveniently port most applications from `GStreamer`-0.8 to `GStreamer`-0.10, with references to the relevant sections in this Application Development Manual where needed. With this list, it should be possible to port simple applications to `GStreamer`-0.10 in less than a day.

List of changes

- Most functions returning an object or an object property have been changed to return its own reference rather than a constant reference of the one owned by the object itself. The reason for this change is primarily thread safety. This means, effectively, that return values of functions such as `gst_element_get_pad ()`, `gst_pad_get_name ()` and many more like these have to be free'ed or unreferenced after use. Check the API references of each function to know for sure whether return values should be free'ed or not. It is important that all objects derived from GstObject are ref'ed/unref'ed using gst_object_ref() and gst_object_unref() respectively (instead of g_object_ref/unref).
- Applications should no longer use signal handlers to be notified of errors, end-of-stream and other similar pipeline events. Instead, they should use the `GstBus`, which has been discussed in Chapter 7. The bus will take care that the messages will be delivered in the context of a main loop, which is almost certainly the application's main thread. The big advantage of this is that applications no longer need to be thread-aware; they don't need to use `g_idle_add ()` in the signal handler and do the actual real work in the idle-callback. `GStreamer` now does all that internally.
- Related to this, `gst_bin_iterate ()` has been removed. Pipelines will iterate in their own thread, and applications can simply run a `GMainLoop` (or call the mainloop of their UI toolkit, such as `gtk_main ()`).
- State changes can be delayed (ASYNC). Due to the new fully threaded nature of GStreamer-0.10, state changes are not always immediate, in particular changes including the transition from READY to PAUSED state. This means two things in the context of porting applications: first of all, it is no longer always possible to do `gst_element_set_state ()` and check for a return value of GST_STATE_CHANGE_SUCCESS, as the state change might be delayed (ASYNC) and the result will not be known until later. You should still check for GST_STATE_CHANGE_FAILURE right away, it is just no longer possible to assume that everything that is not SUCCESS means failure. Secondly, state changes might not be immediate, so your code needs to take that into account. You can wait for a state change to complete if you use GST_CLOCK_TIME_NONE as timeout interval with `gst_element_get_state ()`.
- In 0.8, events and queries had to manually be sent to sinks in pipelines (unless you were using playbin). This is no longer the case in 0.10. In 0.10, queries and events can be sent to toplevel pipelines, and the pipeline will do the dispatching internally for you. This means less bookkeeping in your application. For a short code example, see Chapter 11. Related, seeking is now threadsafe, and your video

output will show the new video position's frame while seeking, providing a better user experience.

- The `GstThread` object has been removed. Applications can now simply put elements in a pipeline with optionally some "queue" elements in between for buffering, and `GStreamer` will take care of creating threads internally. It is still possible to have parts of a pipeline run in different threads than others, by using the "queue" element. See Chapter 16 for details.
- Filtered caps -> capsfilter element (the pipeline syntax for gst-launch has not changed though).
- libgstgconf-0.10.la does not exist. Use the "gconfvideosink" and "gconfaudiosink" elements instead, which will do live-updates and require no library linking.
- The "new-pad" and "state-change" signals on `GstElement` were renamed to "pad-added" and "state-changed".
- `gst_init_get_popt_table ()` has been removed in favour of the new GOption command line option API that was added to GLib 2.6. `gst_init_get_option_group ()` is the new GOption-based equivalent to `gst_init_get_ptop_table ()`.

Chapter 23. Integration

GStreamer tries to integrate closely with operating systems (such as Linux and UNIX-like operating systems, OS X or Windows) and desktop environments (such as GNOME or KDE). In this chapter, we'll mention some specific techniques to integrate your application with your operating system or desktop environment of choice.

Linux and UNIX-like operating systems

GStreamer provides a basic set of elements that are useful when integrating with Linux or a UNIX-like operating system.

- For audio input and output, GStreamer provides input and output elements for several audio subsystems. Amongst others, GStreamer includes elements for ALSA (alsasrc, alsamixer, alsasink), OSS (osssrc, ossmixer, osssink) and Sun audio (sunaudiosrc, sunaudiomixer, sunaudiosink).
- For video input, GStreamer contains source elements for Video4linux (v4lsrc, v4lmjpegsrc, v4lelement and v4lmjpegisnk) and Video4linux2 (v4l2src, v4l2element).
- For video output, GStreamer provides elements for output to X-windows (ximagesink), Xv-windows (xvimagesink; for hardware-accelerated video), direct-framebuffer (dfbimagesink) and openGL image contexts (glsink).

GNOME desktop

GStreamer has been the media backend of the GNOME[1] desktop since GNOME-2.2 onwards. Nowadays, a whole bunch of GNOME applications make use of GStreamer for media-processing, including (but not limited to) Rhythmbox[2], Totem[3] and Sound Juicer[4].

Most of these GNOME applications make use of some specific techniques to integrate as closely as possible with the GNOME desktop:

- GNOME applications call `gnome_program_init ()` to parse command-line options and initialize the necessary gnome modules. GStreamer applications would normally call `gst_init ()` to do the same for GStreamer. This would mean that only one of the two can parse command-line options. To work around this issue, GStreamer can provide a GLib `GOptionGroup` which can be passed to `gnome_program_init ()`. The following example requires Gnome-2.14 or newer (previous libgnome versions do not support command line parsing via GOption yet but use the now deprecated popt)

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

static gchar **cmd_filenames = NULL;

static GOptionEntries cmd_options[] = {
  /* here you can add command line options for your application. Check
   * the GOption section in the GLib API reference for a more elaborate
   * example of how to add your own command line options here */

  /* at the end we have a special option that collects all remaining
   * command line arguments (like filenames) for us. If you don’t
   * need this, you can safely remove it */
  { G_OPTION_REMAINING, 0, 0, G_OPTION_ARG_FILENAME_ARRAY, &cmd_filenames,
    "Special option that collects any remaining arguments for us" },

  /* mark the end of the options array with a NULL option */
  { NULL, }
};

/* this should usually be defined in your config.h */
#define VERSION "0.0.1"

gint
main (gint argc, gchar **argv)
{
  GOptionContext *context;
  GOptionGroup *gstreamer_group;
  GnomeProgram *program;

  /* we must initialise the threading system before using any
   * other GLib funtion, such as g_option_context_new() */
  if (!g_thread_supported ())
    g_thread_init (NULL);

  context = g_option_context_new ("gnome-demo-app");

  /* get command line options from GStreamer and add them to the group */
  gstreamer_group = gst_init_get_option_group ();
  g_option_context_add_group (context, gstreamer_group);

  /* add our own options. If you are using gettext for translation of your
   * strings, use GETTEXT_PACKAGE here instead of NULL */
  g_option_context_add_main_entries (context, cmd_options, NULL);

  program = gnome_program_init ("gnome-demo-app", VERSION
                                LIBGNOMEUI_MODULE, argc, argv,
                                GNOME_PARAM_HUMAN_READABLE_NAME, "Gnome Demo",
                                GNOME_PARAM_GOPTION_CONTEXT, context,
                                NULL);

  /* any filenames we got passed on the command line? parse them! */
  if (cmd_filenames != NULL) {
    guint i, num;

    num = g_strv_length (cmd_filenames);
    for (i = 0; i < num; ++i) {
      /* do something with the filename ... */
      g_print ("Adding to play queue: %s\n", cmd_filenames[i]);
    }

    g_strfreev (cmd_filenames);
    cmd_filenames = NULL;
  }

[..]

}
```

- GNOME stores the default video and audio sources and sinks in GConf. `GStreamer` provides a number of elements that create audio and video sources and sinks directly based on those GConf settings. Those elements are: gconfaudiosink, gconfvideosink, gconfaudiosrc and gconfvideosrc. You can create them with `gst_element_factory_make ()` and use them directly just like you would use any other source or sink element. All GNOME applications are recommended to use those elements.
- `GStreamer` provides data input/output elements for use with the GNOME-VFS system. These elements are called "gnomevfssrc" and "gnomevfssink".

KDE desktop

`GStreamer` has been proposed for inclusion in KDE-4.0. Currently, `GStreamer` is included as an optional component, and it's used by several KDE applications, including AmaroK[5], JuK[6], KMPlayer[7] and Kaffeine[8].

Although not yet as complete as the GNOME integration bits, there are already some KDE integration specifics available. This list will probably grow as `GStreamer` starts to be used in KDE-4.0:

- AmaroK contains a kiosrc element, which is a source element that integrates with the KDE VFS subsystem KIO.

OS X

`GStreamer` provides native video and audio output elements for OS X. It builds using the standard development tools for OS X.

Windows

Warning

Note: this section is out of date. GStreamer-0.10 has much better support for win32 than previous versions though and should usually compile and work out-of-the-box both using MSYS/MinGW or Microsoft compilers. The GStreamer web site[9] and the mailing list archives[10] are a good place to check the latest win32-related news.

`GStreamer` builds using Microsoft Visual C .NET 2003 and using Cygwin.

Building `GStreamer` under Win32

There are different makefiles that can be used to build GStreamer with the usual Microsoft compiling tools.

The Makefile is meant to be used with the GNU make program and the free version of the Microsoft compiler (http://msdn.microsoft.com/visualc/vctoolkit2003/). You also have to modify your system environment variables to use it from the command-line. You will also need a working Platform SDK for Windows that is available for free from Microsoft.

The projects/makefiles will generate automatically some source files needed to compile GStreamer. That requires that you have installed on your system some GNU tools and that they are available in your system PATH.

The GStreamer project depends on other libraries, namely :

- GLib
- libxml2
- libintl
- libiconv

Work is being done to provide pre-compiled GStreamer-0.10 libraries as a packages for win32. Check the GStreamer web site[12] and check our mailing list [13] for the latest developments in this respect.

> **Notes:** GNU tools needed that you can find on http://gnuwin32.sourceforge.net/
>
> - GNU flex (tested with 2.5.4)
> - GNU bison (tested with 1.35)
>
> and http://www.mingw.org/
>
> - GNU make (tested with 3.80)
>
> the generated files from the -auto makefiles will be available soon separately on the net for convenience (people who don't want to install GNU tools).

Installation on the system

FIXME: This section needs be updated for GStreamer-0.10.

Notes

1. http://www.gnome.org/
2. http://www.rhythmbox.org/
3. http://www.hadess.net/totem.php3

4. http://www.burtonini.com/blog/computers/sound-juicer
5. http://amarok.kde.org/
6. http://developer.kde.org/~wheeler/juk.html
7. http://www.xs4all.nl/~jjvrieze/kmplayer.html
8. http://kaffeine.sourceforge.net/
9. http://gstreamer.freedesktop.org
10. http://news.gmane.org/gmane.comp.video.gstreamer.devel
11. http://msdn.microsoft.com/visualc/vctoolkit2003/
12. http://gstreamer.freedesktop.org
13. http://news.gmane.org/gmane.comp.video.gstreamer.devel
14. http://gnuwin32.sourceforge.net/
15. http://www.mingw.org/

Chapter 24. Licensing advisory

How to license the applications you build with `GStreamer`

The licensing of GStreamer is no different from a lot of other libraries out there like GTK+ or glibc: we use the LGPL. What complicates things with regards to GStreamer is its plugin-based design and the heavily patented and proprietary nature of many multimedia codecs. While patents on software are currently only allowed in a small minority of world countries (the US and Australia being the most important of those), the problem is that due to the central place the US hold in the world economy and the computing industry, software patents are hard to ignore wherever you are. Due to this situation, many companies, including major GNU/Linux distributions, get trapped in a situation where they either get bad reviews due to lacking out-of-the-box media playback capabilities (and attempts to educate the reviewers have met with little success so far), or go against their own - and the free software movement's - wish to avoid proprietary software. Due to competitive pressure, most choose to add some support. Doing that through pure free software solutions would have them risk heavy litigation and punishment from patent owners. So when the decision is made to include support for patented codecs, it leaves them the choice of either using special proprietary applications, or try to integrate the support for these codecs through proprietary plugins into the multimedia infrastructure provided by GStreamer. Faced with one of these two evils the GStreamer community of course prefer the second option.

The problem which arises is that most free software and open source applications developed use the GPL as their license. While this is generally a good thing, it creates a dilemma for people who want to put together a distribution. The dilemma they face is that if they include proprietary plugins in GStreamer to support patented formats in a way that is legal for them, they do risk running afoul of the GPL license of the applications. We have gotten some conflicting reports from lawyers on whether this is actually a problem, but the official stance of the FSF is that it is a problem. We view the FSF as an authority on this matter, so we are inclined to follow their interpretation of the GPL license.

So what does this mean for you as an application developer? Well, it means you have to make an active decision on whether you want your application to be used together with proprietary plugins or not. What you decide here will also influence the chances of commercial distributions and Unix vendors shipping your application. The GStreamer community suggest you license your software using a license that will allow proprietary plugins to be bundled with GStreamer and your applications, in order to make sure that as many vendors as possible go with GStreamer instead of less free solutions. This in turn we hope and think will let GStreamer be a vehicle for wider use of free formats like the Xiph.org formats.

If you do decide that you want to allow for non-free plugins to be used with your application you have a variety of choices. One of the simplest is using licenses like LGPL, MPL or BSD for your application instead of the GPL. Or you can add an exception clause to your GPL license stating that you except GStreamer plugins from the obligations of the GPL.

A good example of such a GPL exception clause would be, using the Totem video player project as an example: The authors of the Totem video player project hereby grants permission for non-GPL-compatible GStreamer plugins to be used and distributed together with GStreamer and Totem. This permission goes above and beyond the permissions granted by the GPL license Totem is covered by.

Our suggestion among these choices is to use the LGPL license, as it is what resembles the GPL most and it makes it a good licensing fit with the major GNU/Linux desktop projects like GNOME and KDE. It also allows you to share code more openly with projects that have compatible licenses. Obviously, pure GPL code without the abovementioned clause is not usable in your application as such. By choosing the LGPL, there is no need for an exception clause and thus code can be shared more freely.

I have above outlined the practical reasons for why the GStreamer community suggests you allow non-free plugins to be used with your applications. We feel that in the multimedia arena, the free software community is still not strong enough to set the agenda and that blocking non-free plugins to be used in our infrastructure hurts us more than it hurts the patent owners and their ilk.

This view is not shared by everyone. The Free Software Foundation urges you to use an unmodified GPL for your applications, so as to push back against the temptation to use non-free plug-ins. They say that since not everyone else has the strength to reject them because they are unethical, they ask your help to give them a legal reason to do so.

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

Notes

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

Chapter 25. Quotes from the Developers

As well as being a cool piece of software, `GStreamer` is a lively project, with developers from around the globe very actively contributing. We often hang out on the #gstreamer IRC channel on irc.freenode.net: the following are a selection of amusing[1] quotes from our conversations.

6 Mar 2006

> When I opened my eyes I was in a court room. There were masters McIlroy and Thompson sitting in the jury and master Kernighan too. There were the GStreamer developers standing in the defendant's place, accused of violating several laws of Unix philosophy and customer lock-down via running on a proprietary pipeline, different from that of the Unix systems. I heard Eric Raymond whispering "got to add this case to my book.
>
> *behdad's blog*

22 May 2007

> <*__tim*> Uraeus: amusing, isn't it?
>
> <*Uraeus*> __tim: I wrote that :)
>
> <*__tim*> Uraeus: of course you did; your refusal to surrender to the oppressive regime of the third-person-singular-rule is so unique in its persistence that it's hard to miss :)

12 Sep 2005

> <*wingo*> we just need to get rid of that mmap stuff
>
> <*wingo*> i think gnomevfssrc is faster for files even
>
> <*BBB*> wingo, no
>
> <*BBB*> and no
>
> <*wingo*> good points ronald

23 Jun 2005

> * *wingo* back
>
> * *thomasvs* back
>
> --- You are now known as everybody
>
> * *everybody* back back
>
> <*everybody*> now break it down
>
> --- You are now known as thomasvs
>
> * *bilboed* back
>
> --- bilboed is now known as john-sebastian
>
> * *john-sebastian* bach
>
> --- john-sebastian is now known as bilboed

--- You are now known as scratch_my

* *scratch_my* back

--- bilboed is now known as Illbe

--- You are now known as thomasvs

* *Illbe* back

--- Illbe is now known as bilboed

20 Apr 2005

thomas: jrb, somehow his screenshotsrc grabs whatever X is showing and makes it available as a stream of frames

jrb: thomas: so, is the point that the screenshooter takes a video? but won't the dialog be in the video? oh, nevermind. I'll just send mail...

thomas: jrb, well, it would shoot first and ask questions later

2 Nov 2004

zaheerm: wtay: unfair u fixed the bug i was using as a feature!

14 Oct 2004

* *zaheerm* wonders how he can break gstreamer today :)

ensonic: zaheerm, spider is always a good starting point

14 Jun 2004

teuf: ok, things work much better when I don't write incredibly stupid and buggy code

thaytan: I find that too

23 Nov 2003

Uraeus: ah yes, the sleeping part, my mind is not multitasking so I was still thinking about exercise

dolphy: Uraeus: your mind is multitasking

dolphy: Uraeus: you just miss low latency patches

14 Sep 2002

--- *wingo-party* is now known as *wingo*

* *wingo* holds head

4 Jun 2001

taaz: you witchdoctors and your voodoo mpeg2 black magic...

omega_: um. I count three, no four different cults there <g>

ajmitch: hehe

omega_: witchdoctors, voodoo, black magic, *omega_:* and mpeg

16 Feb 2001

wtay: I shipped a few commerical products to >40000 people now but GStreamer is way more exciting...

16 Feb 2001

* *tool-man* is a gstreamer groupie

14 Jan 2001

Omega: did you run ldconfig? maybe it talks to init?

wtay: not sure, don't think so... I did run gstreamer-register though :-)

Omega: ah, that did it then ;-)

wtay: right

Omega: probably not, but in case GStreamer starts turning into an OS, someone please let me know?

9 Jan 2001

wtay: me tar, you rpm?

wtay: hehe, forgot "zan"

Omega: ?

wtay: me tar"zan", you ...

7 Jan 2001

Omega: that means probably building an agreggating, cache-massaging queue to shove N buffers across all at once, forcing cache transfer.

wtay: never done that before...

Omega: nope, but it's easy to do in gstreamer <g>

wtay: sure, I need to rewrite cp with gstreamer too, someday :-)

7 Jan 2001

wtay: GStreamer; always at least one developer is awake...

5/6 Jan 2001

wtay: we need to cut down the time to create an mp3 player down to seconds...

richardb: :)

Omega: I'm wanting to something more interesting soon, I did the "draw an mp3 player in 15sec" back in October '99.

wtay: by the time Omega gets his hands on the editor, you'll see a complete audio mixer in the editor :-)

richardb: Well, it clearly has the potential...

Omega: Working on it... ;-)

28 Dec 2000

MPAA: We will sue you now, you have violated our IP rights!

wtay: hehehe

MPAA: How dare you laugh at us? We have lawyers! We have Congressmen! We have *LARS*!

wtay: I'm so sorry your honor

MPAA: Hrumph.

* *wtay* bows before thy

Notes

1. No guarantee of sense of humour compatibility is given.