`GStreamer` Application Development Manual

Wim Taymans

Steve Baker

Andy Wingo

`GStreamer` Application Development Manual

by Wim Taymans, Steve Baker, and Andy Wingo

Table of Contents

Chapter 1. Introduction

This chapter gives you an overview of the technologies described in this book.

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 either audio or video or both. The pipeline design is made to have no extra overhead above what the applied filters induce. This makes GStreamer a good framework for designing even high-end audio applications which puts 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, MPEG1, MPEG2, AVI, Quicktime, mod and so on. 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.

This book is about GStreamer from a 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.

Chapter 2. Motivation

Linux has historically lagged behind other operating systems in the multimedia arena. Microsoft's Windows[tm] and Apple's MacOS[tm] both have strong support for multimedia devices, multimedia content creation, playback, and realtime processing. Linux, on the other hand, has a poorly integrated collection of multimedia utilities and applications available, which can hardly compete with the professional level of software available for MS Windows and MacOS.

Current problems

We descibe the typical problems in todays media handling on Linux.

Multitude of duplicate code

The Linux user who wishes to hear a sound file must hunt through their collection of sound file players in order to play the tens of sound file formats in wide use today. Most of these players basically reimplement the same code over and over again.

The Linux developer who wishes to embed a video clip in their application must use crude hacks to run an external video player. There is no library available that a developer can use to create a custom media player.

'One goal' media players/libraries

Your typical MPEG player was designed to play MPEG video and audio. Most of these players have implemented a complete infrastructure focused on achieving their only goal: playback. No provisions were made to add filters or special effects to the video or audio data.

If I wanted to convert an MPEG2 video stream into an AVI file, my best option would be to take all of the MPEG2 decoding algorithms out of the player and duplicate them into my own AVI encoder. These algorithms cannot easily be shared accross applications.

Attempts have been made to create libraries for handling various media types. Because they focus on a very specific media type (avifile, libmpeg2, ...), significant work is needed to integrate them due to a lack of a common API. GStreamer allows you to wrap these libraries with a common API, which significantly simplifies integration and reuse.

Non unified plugin mechanisms

Your typical media player might have a plugin for different media types. Two media players will typically implement their own plugin mechanism so that the codecs cannot be easily exchanged. The plugin system of the typical media player is also very tailored to the specific needs of the application.

The lack of a unified plugin mechanism also seriously hinders the creation of binary only codecs. No company is willing to port their code to all the different plugin mechanisms.

While GStreamer also uses it own plugin system it offers a very rich framework for the plugin developper and ensures the plugin can be used in a wide range of applications, transparently interacting with other plugins. The Framework that GStreamer provides for the plugins is flexible enough to host even the most demanding plugins.

Provision for network transparency

No infrastructure is present to allow network transparent media handling. A distributed MPEG encoder will typically duplicate the same encoder algorithms found in a non-distributed encoder.

No provisions have been made for emerging technologies such as the GNOME object embedding using Bonobo[1].

The GStreamer cores does not use network transparent technologies at the lowest level as it only adds overhead for the local case. That said, it shouldn't be hard to create a wrapper around the core components.

Catch up with the Windows(tm) world

We need solid media handling if we want to see Linux succeed on the desktop.

We must clear the road for commercially backed codecs and multimedia applications so that Linux can become an option for doing multimedia.

Notes

1. http://developer.gnome.org/arch/component/bonobo.html

Chapter 3. Goals

GStreamer was designed to provide a solution to the current Linux media problems.

The design goals

We describe what we try to achieve with GStreamer.

Clean and powerful

GStreamer wants to provide 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 an example too.

Object oriented

Adhere to the GLib 2.0 object model. A programmer familiar with GLib 2.0 or older versions of 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.

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 g_mem_chunk and fast non-blocking allocation algorithms where possible to minimize dynamic memory allocation.
- 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(3). Sub-buffers efficiently split buffers into manageable pieces.
- The use of cothreads to minimize the threading overhead. Cothreads are a simple and fast user-space method for switching between subtasks. Cothreads were measured to consume as little as 600 cpu cycles.
- Allowing hardware acceleration by the use of 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.
- All critical data passing is free of locks and mutexes.

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 tarkin and vorbis[1].

Notes

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

Chapter 4. Initializing GStreamer

When writing a GStreamer application, you can simply include gst/gst.h to get access to the library functions.

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 would start like this:

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

...

int
main (int argc, char *argv[])
{
  ...
  gst_init (&argc, &argv);
  ...
}
```

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.

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

Use the GST_VERSION_MAJOR, GST_VERSION_MINOR and GST_VERSION_MICRO macros to get the GStreamer version you are building against or use gst_version() to get the version you are linked against.

The popt interface

You can also use a popt table to initialize your own parameters as shown in the next code fragment:

```
int
main(int argc, char *argv[])
{
  gboolean silent = FALSE;
  gchar *savefile = NULL;
  struct poptOption options[] = {
    {"silent", 's',  POPT_ARG_NONE|POPT_ARGFLAG_STRIP,   &silent,   0,
     "do not output status information", NULL},
    {"output", 'o',  POPT_ARG_STRING|POPT_ARGFLAG_STRIP, &savefile, 0,
     "save xml representation of pipeline to FILE and exit", "FILE"},
    POPT_TABLEEND
  };

  gst_init_with_popt_table (&argc, &argv, options);

  ...
```

As shown in this fragment, you can use a popt[1] table to define your application-specific command line options, and pass this table to the function gst_init_with_popt_table. Your application options will be parsed in addition to the standard GStreamer options.

Notes

1. http://developer.gnome.org/doc/guides/popt/

Chapter 5. GstElement

The most important object in `GStreamer` for the application programmer is the `GstElement` object.

What is a GstElement

`GstElement` is the basic building block for the media pipeline. All the different components you are going to use are derived from `GstElement`. This means that a lot of functions you are going to use operate on objects of this class.

Elements, from the perspective of GStreamer, are viewed as "black boxes" with a number of different aspects. One of these aspects is the presence of "pads", or link points. This terminology arises from soldering; pads are where wires can be attached.

Source elements

Source elements generate data for use by a pipeline, for example reading from disk or from a sound card.

Below you see how we will visualize the element. We always draw a source pad to the right of the element.

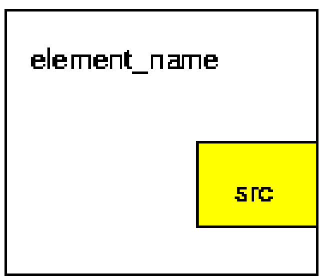

Figure 5-1. Visualisation of a source element

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

Filters and codecs

Filter elements both have input and output pads. They operate on data they receive in their sink pads and produce data on their source pads. For example, MPEG decoders and volume filters would fall into this category.

Elements are not constrained as to the number of pads they might have; for example, a video mixer might have two input pads (the images of the two different video streams) and one output pad.

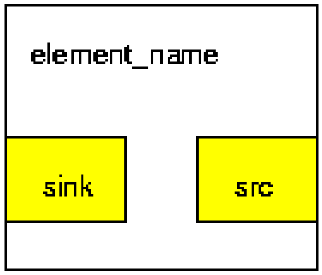

Figure 5-2. Visualisation of a filter element

The above figure shows the visualisation of a filter element. This element has one sink (input) pad and one source (output) pad. Sink pads are drawn on the left of the element.

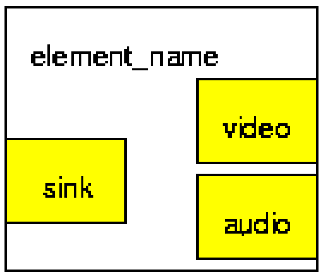

Figure 5-3. Visualisation of a filter element with more than one output pad

The above figure shows the visualisation of a filter element with more than one output pad. An example of such a filter is the AVI splitter (demultiplexer). This element will parse the input data and extract the audio and video data. Most of these filters dynamically send out a signal when a new pad is created so that the application programmer can link an arbitrary element to the newly created pad.

Sink elements

Sink elements are terminal 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.

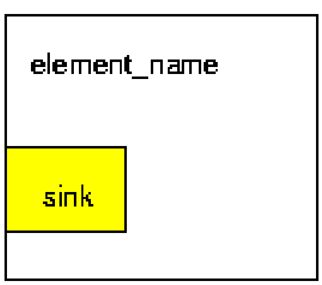

Figure 5-4. Visualisation of a sink element

Creating a GstElement

A `GstElement` object is created from a factory. To create an element, you have to get access to a `GstElementFactory` object using a unique factory name.

The following code example is used to get a factory that can be used to create the 'mad' element, an mp3 decoder.

```
GstElementFactory *factory;

factory = gst_element_factory_find ("mad");
```

Once you have the handle to the element factory, you can create a real element with the following code fragment:

```
GstElement *element;

element = gst_element_factory_create (factory, "decoder");
```

`gst_element_factory_create` will use the element factory to create an element with the given name. The name of the element is something you can use later on to look up the element in a bin, for example. You can pass NULL as the name argument to get a unique, default name.

A simple shortcut exists for creating an element from a factory. The following example creates an element named "decoder" from the element factory named "mad". This convenience function is most widely used to create an element.

```
GstElement *element;

element = gst_element_factory_make ("mad", "decoder");
```

When you don't need the element anymore, you need to unref it, as shown in the following example.

```
GstElement *element;

  ...
gst_element_unref (element);
```

GstElement properties

A `GstElement` 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 of 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.

```
GstElement *element;
GValue value = { 0, }; /* initialize the GValue for g_object_get() */

element = gst_element_factory_make ("mad", "decoder");
g_object_set (G_OBJECT (element), "name", "mydecoder", NULL);
...
```

```
g_value_init (&value, G_TYPE_STRING);
g_object_get_property (G_OBJECT (element), "name", &value);
...
```

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.

For more information about `GObject` properties we recommend you read the GObject manual[1].

GstElement signals

A `GstElement` also provides various `GObject` signals that can be used as a flexible callback mechanism.

More about GstElementFactory

We talk some more about the GstElementFactory object.

Getting information about an element using the factory details

Finding out what pads an element can contain

Different ways of querying the factories

Notes

1. http://developer.gnome.org/doc/API/2.0/gobject/index.html

Chapter 6. Plugins

A plugin is a shared library that contains at least one of the following items:

- one or more element factories
- one or more type definitions
- one or more auto-pluggers
- exported symbols for use in other plugins

All plugins should implement one function, `plugin_init`, that creates all the element factories and registers all the type definitions contained in the plugin. Without this function, a plugin cannot be registered.

The plugins are maintained in the plugin system. Optionally, the type definitions and the element factories can be saved into an XML representation so that the plugin system does not have to load all available plugins in order to know their definition.

The basic plugin structure has the following fields:

```
typedef struct _GstPlugin   GstPlugin;

struct _GstPlugin {
  gchar *name;                  /* name of the plugin */
  gchar *longname;              /* long name of plugin */
  gchar *filename;              /* filename it came from */

  GList *types;                 /* list of types provided */
  gint numtypes;
  GList *elements;              /* list of elements provided */
  gint numelements;
  GList *autopluggers;          /* list of autopluggers provided */
  gint numautopluggers;

  gboolean loaded;              /* if the plugin is in memory */
};
```

You can query a `GList` of available plugins with the function `gst_plugin_get_list` as this example shows:

```
    GList *plugins;

    plugins = gst_plugin_get_list ();

    while (plugins) {
      GstPlugin *plugin = (GstPlugin *)plugins->data;

      g_print ("plugin: %s\n", gst_plugin_get_name (plugin));

      plugins = g_list_next (plugins);
    }
```

Chapter 7. GstPad

As we have seen in the previous chapter (GstElement), the pads are the element's links with the outside world.

The specific type of media that the element can handle will be exposed by the pads. The description of this media type is done with capabilities (`GstCaps`)

Getting pads from an element

Once you have created an element, you can get one of its pads with:

```
GstPad *srcpad;
   ...
srcpad = gst_element_get_pad (element, "src");
   ...
```

This function will get the pad named "src" from the given element.

Alternatively, you can request a GList of pads from the element. The following code example will print the names of all the pads of an element.

```
GList *pads;
   ...
pads = gst_element_get_pad_list (element);
while (pads) {
  GstPad *pad = GST_PAD (pads->data);

  g_print ("pad name %s\n", gst_pad_get_name (pad));

  pads = g_list_next (pads);
}
   ...
```

Useful pad functions

You can get the name of a pad with gst_pad_get_name () and set its name with get_pad_set_name().

gst_pad_get_direction (GstPad *pad) can be used to query if the pad is a sink or a source pad. Remember that a source pad is a pad that can output data and a sink pad is one that accepts data.

You can get the parent of the pad, this is the element that this pad belongs to, with get_pad_get_parent(GstPad *pad). This function will return a pointer to a GstElement.

Dynamic pads

Some elements might not have their pads when they are created. This can happen, for example, with an MPEG2 system demultiplexer. The demultiplexer will create its pads at runtime when it detects the different elementary streams in the MPEG2 system stream.

Running `gst-inspect mpegdemux` will show that the element has only one pad: a sink pad called 'sink'. The other pads are "dormant" as you can see in the padtemplates from the 'Exists: Sometimes' property. Depending on the type of MPEG2 file you play, the pads are created. We will see that this is very important when you are going to create dynamic pipelines later on in this manual.

You can attach a signal to an element to inform you when the element has created a new pad from one of its padtemplates. The following piece of code is an example of how to do this:

```
static void
pad_link_func (GstElement *parser, GstPad *pad, GstElement *pipeline)
{
  g_print("***** a new pad %s was created\n", gst_pad_get_name(pad));

  gst_element_set_state (pipeline, GST_STATE_PAUSED);

  if (strncmp (gst_pad_get_name (pad), "private_stream_1.0", 18) == 0) {
    // set up an AC3 decoder pipeline
    ...
    // link pad to the AC3 decoder pipeline
    ...
  }
  gst_element_set_state (GST_ELEMENT (audio_thread), GST_STATE_READY);
}

int
main(int argc, char *argv[])
{
  GstElement *pipeline;
  GstElement *mpeg2parser;

  // create pipeline and do something useful
  ...

  mpeg2parser = gst_element_factory_make ("mpegdemux", "mpegdemux");
  g_signal_connect (G_OBJECT (mpeg2parser), "new_pad", pad_link_func, pipeline);
  ...

  // start the pipeline
  gst_element_set_state (GST_ELEMENT (pipeline), GST_STATE_PLAYING);
  ...
}
```

Note: You need to set the pipeline to READY or NULL if you want to change it.

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.

The tee element, for example, has one input pad and a request padtemplate for the output pads. Whenever an element wants to get an output pad from the tee element, it has to request the pad.

The following piece of code can be used to get a pad from the tee element. After the pad has been requested, it can be used to link another element to it.

```
    ...
  GstPad *pad;
    ...
  element = gst_element_factory_make ("tee", "element");

  pad = gst_element_get_request_pad (element, "src%d");
  g_print ("new pad %s\n", gst_pad_get_name (pad));
    ...
```

The gst_element_get_request_pad method can be used to get a pad from the element based on the name_template of the padtemplate.

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 gst_element_get_compatible_pad is used to request a compatible pad, as is shown in the next example.

```
  ...
GstPadTemplate *templ;
GstPad *pad;
  ...
element = gst_element_factory_make ("tee", "element");
mad = gst_element_factory_make ("mad", "mad");

templ = gst_element_get_pad_template_by_name (mad, "sink");

pad = gst_element_get_compatible_pad (element, templ);
g_print ("new pad %s\n", gst_pad_get_name (pad));
...
```

Capabilities of a GstPad

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 pad by using capabilities.

We will briefly describe what capabilities are, enough for you to get a basic understanding of the concepts. You will find more information on how to create capabilities in the Plugin Writer's Guide.

What is a capability

A capability is attached to a pad in order to describe what type of media the pad can handle.

A capability is named and consists of a MIME type and a set of properties. Its data structure is:

```
struct _GstCaps {
  gchar *name;                  /* the name of this caps */
  guint16 id;                   /* type id (major type) */

  guint refcount;   /* caps are refcounted */

  GstProps *properties;         /* properties for this capability */

  GstCaps *next;  /* caps can be chained together */
};
```

Below is a dump of the capabilities of the element mad, as shown by **gst-inspect**. You can see two pads: sink and src. Both pads have capability information attached to them.

The sink pad (input pad) is called 'sink' and takes data of MIME type 'audio/mp3'. It also has three properties: layer, bitrate and framed.

The source pad (output pad) is called 'src' and outputs data of MIME type 'audio/raw'. It also has four properties: format, depth, rate and channels.

```
Pads:
  SINK template: 'sink'
    Availability: Always
    Capabilities:
      'mad_sink':
        MIME type: 'audio/mp3':

  SRC template: 'src'
    Availability: Always
    Capabilities:
      'mad_src':
        MIME type: 'audio/raw':
        format: String: int
        endianness: Integer: 1234
        width: Integer: 16
        depth: Integer: 16
        channels: Integer range: 1 - 2
        law: Integer: 0
        signed: Boolean: TRUE
        rate: Integer range: 11025 - 48000
```

What are properties

Properties are used to describe extra information for capabilities. The properties basically consist of a key (a string) and a value. There are different possibile value types that can be used:

- An integer value: the property has this exact value.
- An integer range value. The property denotes a range of possible values. In the case of the mad element, the source pad has a property rate that can go from 11025 to 48000.
- A boolean value.
- a fourcc value: this is a value that is commonly used to describe an encoding for video, as used by the AVI specification.
- A list value: the property can take any value from a list.
- A float value: the property has this exact floating point value.
- A float range value: denotes a range of possible floating point values.
- A string value.

What capabilities are used for

Capabilities describe in great detail the type of media that is handled by the pads. They are mostly used for:

- Autoplugging: automatically finding plugins for a set of capabilities
- Compatibility detection: when two pads are linked, `GStreamer` can verify if the two pads are talking about the same media types.

Getting the capabilities of a pad

A pad can have a chain of capabilities attached to it. You can get the capabilities chain with:

```
GstCaps *caps;
   ...
caps = gst_pad_get_caps (pad);

g_print ("pad name %s\n", gst_pad_get_name (pad));

while (caps) {
  g_print (" Capability name %s, MIME type %s\n",
      gst_caps_get_name (cap),
                               gst_caps_get_mime (cap));

  caps = caps->next;
}
   ...
```

Creating capability structures

While capabilities are mainly used inside a plugin to describe the media type of the pads, the application programmer also has to have basic understanding of capabilities in order to interface with the plugins, specially when using the autopluggers.

As we said, a capability has a name, a mime-type and some properties. The signature of the function to create a new `GstCaps` structure is:

```
GstCaps*    gst_caps_new (const gchar *name, const gchar *mime, GstProps *props);
```

You can therefore create a new capability with no properties like this:

```
GstCaps *newcaps;

newcaps = gst_caps_new ("my_caps", "audio/wav", NULL);
```

`GstProps` basically consist of a set of key-value pairs and are created with a function with this signature:

```
GstProps*     gst_props_new   (const gchar *firstname, ...);
```

The keys are given as strings and the values are given with a set of macros:

- GST_PROPS_INT(a): An integer value
- GST_PROPS_FLOAT(a): A floating point value
- GST_PROPS_FOURCC(a): A fourcc value
- GST_PROPS_BOOLEAN(a): A boolean value
- GST_PROPS_STRING(a): A string value

The values can also be specified as ranges with:

- GST_PROPS_INT_RANGE(a,b): An integer ragne from a to b
- GST_PROPS_FLOAT_RANGE(a,b): A float ragne from a to b

All of the above values can be given with a list too, using:

- GST_PROPS_LIST(a,...): A list of property values.

A more complex capability with properties is created like this:

```
GstCaps *newcaps;

newcaps = gst_caps_new ("my_caps",
                        "audio/wav",
     gst_props_new (
     "bitrate", GST_PROPS_INT_RANGE (11025,22050),
     "depth",   GST_PROPS_INT (16),
     "signed",  GST_PROPS_LIST (
         GST_PROPS_BOOLEAN (TRUE),
         GST_PROPS_BOOLEAN (FALSE)
         ),
     NULL
   );
```

Optionally, the convenient shortcut macro can be used. The above complex capability can be created with:

```
GstCaps *newcaps;

newcaps = GST_CAPS_NEW ("my_caps",
                        "audio/wav",
     "bitrate", GST_PROPS_INT_RANGE (11025,22050),
     "depth",   GST_PROPS_INT (16),
     "signed",  GST_PROPS_LIST (
         GST_PROPS_BOOLEAN (TRUE),
         GST_PROPS_BOOLEAN (FALSE)
         )
   );
```

Chapter 8. Linking elements

You can link the different pads of elements together so that the elements form a chain.

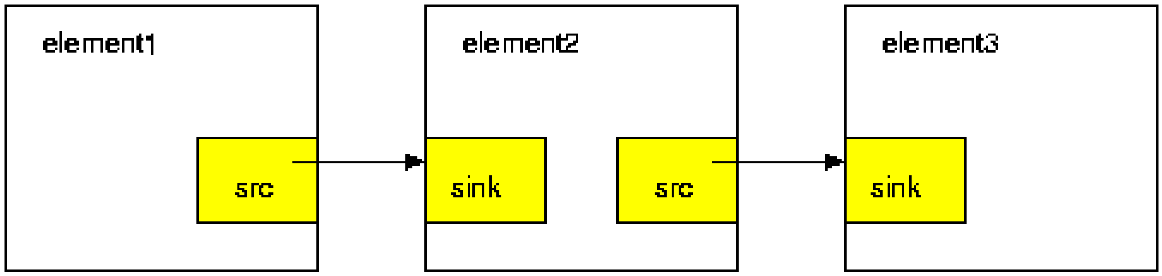

Figure 8-1. Visualisation of three linked elements

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

Imagine the above graph as a simple MPEG audio decoder. The source element is a disk source, the filter element is the MPEG decoder and the sink element is your audiocard. We will use this simple graph to construct an MPEG player later in this manual.

Making simple links

You can link two pads with:

```
GstPad *srcpad, *sinkpad;

srcpad = gst_element_get_pad (element1, "src");
sinpad = gst_element_get_pad (element2, "sink");

// link them
gst_pad_link (srcpad, sinkpad);
  ....
// and unlink them
gst_pad_unlink (srcpad, sinkpad);
```

A convenient shortcut for the above code is done with the gst_element_link_pads () function:

```
// link them
gst_element_link_pads (element1, "src", element2, "sink");
  ....
// and unlink them
gst_element_unlink_pads (element1, "src", element2, "sink");
```

An even more convenient shortcut for single-source, single-sink elements is the gst_element_link () function:

```
// link them
gst_element_link (element1, element2);
```

```
  ....
// and unlink them
gst_element_unlink (element1, element2);
```

If you have more than one element to link, the gst_element_link_many () function takes a NULL-terminated list of elements:

```
// link them
gst_element_link_many (element1, element2, element3, element4, NULL);
  ....
// and unlink them
gst_element_unlink_many (element1, element2, element3, element4, NULL);
```

You can query if a pad is linked with GST_PAD_IS_LINKED (pad).

To query for the `GstPad` a pad is linked to, use gst_pad_get_peer (pad).

Making filtered links

You can also force a specific media type on the link by using gst_pad_link_filtered () and gst_element_link_filtered (). FIXME link to caps documentation.

Chapter 9. Bins

A Bin is a container element. You can add elements to a bin. Since a bin is an `GstElement` itself, it can also be added to another bin.

Bins allow you to combine 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.

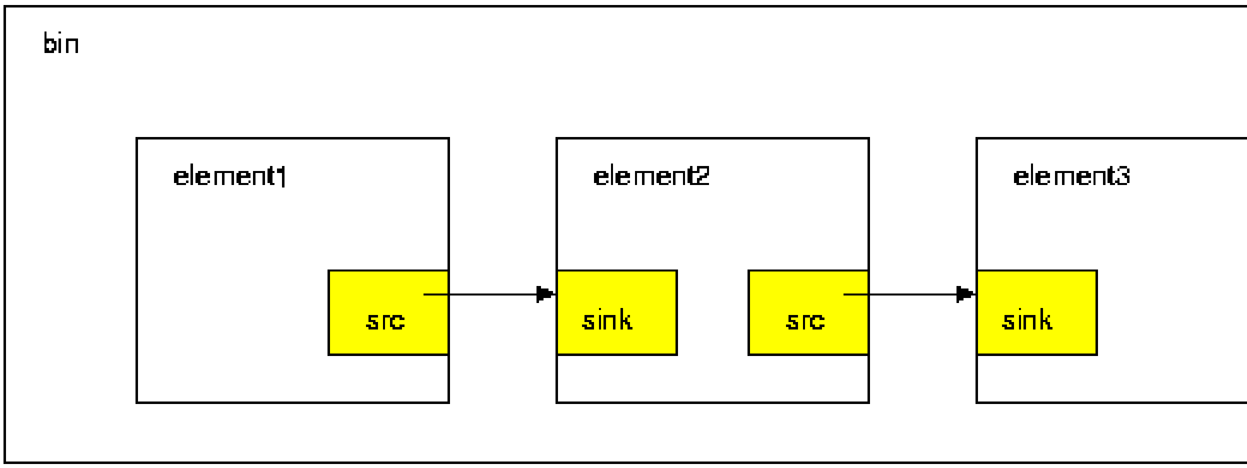

Figure 9-1. Visualisation of a `GstBin` element with some elements in it

There are two standard bins available to the GStreamer programmer:

- A pipeline (`GstPipeline`). Which is a generic container you will use most of the time. The toplevel bin has to be a pipeline.
- A thread (`GstThread`). The plan for the `GstThread` will be run in a separate thread. You will have to use this bin if you have to carefully synchronize audio and video, for example. You will learn more about threads in Chapter 14.

Creating a bin

Bins are created in the same way that other elements are created. ie. using an element factory, or any of the associated convenience functions:

```
GstElement *bin, *thread, *pipeline;

/* create a new bin called 'mybin'. this bin will be only for organizational purposes
   GstBin doesn't affect plan generation */
bin = gst_element_factory_make ("bin", "mybin");

/* create a new thread, and give it a unique name */
thread = gst_element_factory_make ("thread", NULL);

/* the core bins (GstBin, GstThread, GstPipeline) also have convenience APIs,
   gst_<bintype>_new (). these are equivalent to the gst_element_factory_make () synt
```

```
pipeline = gst_pipeline_new ("pipeline_name");
```

Adding elements to a bin

Elements are added to a bin with the following code sample:

```
GstElement *element;
GstElement *bin;

bin = gst_bin_new ("mybin");

element = gst_element_factory_make ("mpg123", "decoder");
gst_bin_add (GST_BIN (bin), element);
 ...
```

Bins and threads can be added to other bins too. This allows you to create nested bins. Note that it doesn't make very much sense to add a `GstPipeline` to anything, as it's a toplevel bin that needs to be explicitly iterated.

To get an element from the bin you can use:

```
GstElement *element;

element = gst_bin_get_by_name (GST_BIN (bin), "decoder");
 ...
```

You can see that the name of the element becomes very handy for retrieving the element from a bin by using the element's name. gst_bin_get_by_name () will recursively search nested bins.

To get a list of elements in a bin, use:

```
GList *elements;

elements = gst_bin_get_list (GST_BIN (bin));

while (elements) {
  GstElement *element = GST_ELEMENT (elements->data);

  g_print ("element in bin: %s\n", GST_OBJECT_NAME (GST_OBJECT (element)));

  elements = g_list_next (elements);
}
 ...
```

To remove an element from a bin, use:

```
GstElement *element;

gst_bin_remove (GST_BIN (bin), element);
 ...
```

To add many elements to a bin at the same time, use the gst_bin_add_many () function. Remember to pass NULL as the last argument.

```
GstElement *filesrc, *decoder, *audiosink;
GstBin *bin;

/* instantiate the elements and the bins... */
```

```
gst_bin_add_many (bin, filesrc, decoder, audiosink, NULL);
```

Custom bins

The application programmer can create custom bins packed with elements to perform a specific task. This allows you to write an MPEG audio decoder with just the following lines of code:

```
/* create the mp3player element */
GstElement *mp3player = gst_element_factory_make ("mp3player", "mp3player");
/* set the source mp3 audio file */
g_object_set (G_OBJECT (mp3player), "location", "helloworld.mp3", NULL);
/* start playback */
gst_element_set_state (GST_ELEMENT (mp3player), GST_STATE_PLAYING);
 ...
/* pause playback */
gst_element_set_state (GST_ELEMENT (mp3player), GST_STATE_PAUSED);
 ...
/* stop */
gst_element_set_state (GST_ELEMENT (mp3player), GST_STATE_NULL);
```

Note that the above code assumes that the mp3player bin derives itself from a `GstThread`, which begins to play as soon as its state is set to PLAYING. Other bin types may need explicit iteration. For more information, see Chapter 14.

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 (FIXME ref).

Ghost pads

You can see from figure Figure 9-2 how a bin has no pads of its own. This is where "ghost pads" come into play.

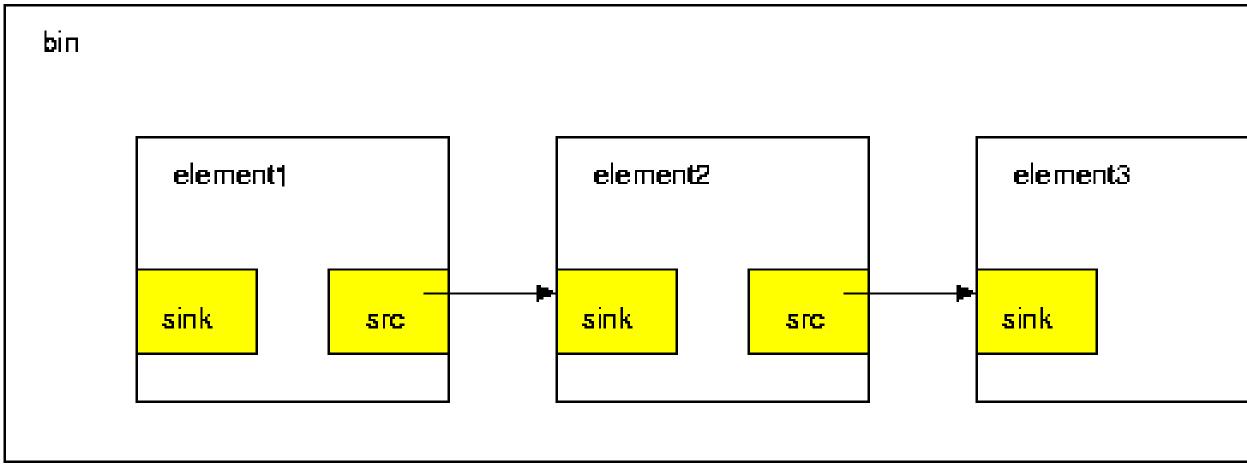

Figure 9-2. Visualisation of a `GstBin` element without ghost pads

A ghost pad is a pad from some element in the bin that has been promoted to the bin. This way, the bin also has a pad. The bin becomes just another element with a pad and you can then use the bin just like any other element. This is a very important feature for creating custom bins.

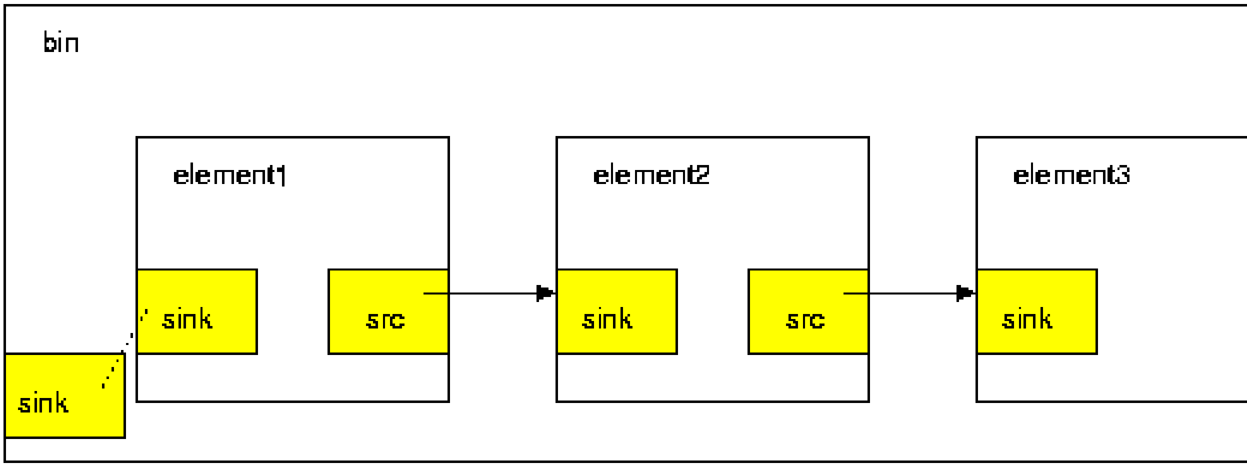

Figure 9-3. Visualisation of a `GstBin` element with a ghost pad

Above is a representation of a ghost pad. The sink pad of element one is now also a pad of the bin.

Ghost pads can actually be added to all `GstElements` and not just `GstBins`. Use the following code example to add a ghost pad to a bin:

```
GstElement *bin;
GstElement *element;

element = gst_element_factory_create ("mad", "decoder");
bin = gst_bin_new ("mybin");

gst_bin_add (GST_BIN (bin), element);

gst_element_add_ghost_pad (bin, gst_element_get_pad (element, "sink"), "sink");
```

In the above example, the bin now also has a pad: the pad called ’sink’ of the given element.

We can now, for example, link the source pad of a filesrc element to the bin with:

```
GstElement *filesrc;

filesrc = gst_element_factory_create ("filesrc", "disk_reader");

gst_element_link_pads (filesrc, "src", bin, "sink");
  ...
```

Chapter 10. 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.

The most important information in the buffer is:

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

GStreamer provides functions to create custom buffer create/destroy algorithms, called a `GstBufferPool`. This makes it possible to efficiently allocate and destroy buffer memory. It also makes it possible to exchange memory between elements by passing the `GstBufferPool`. A video element can, for example, create a custom buffer allocation algorithm that creates buffers with XSHM as the buffer memory. An element can use this algorithm to create and fill the buffer with data.

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 freed and the buffer to be destroyed. A typical MPEG audio decoder works like this.

A more complex case is when the filter modifies the data in place. It does so and simply passes on the buffer to the next element. This is just as easy to deal with. An element that works in place has to be careful when the buffer is used in more than one element; a copy on write has to made in this situation.

Chapter 11. Element states

Once you have created a pipeline packed with elements, nothing will happen right away. This is where the different states come into play.

The different element states

All elements can be in one of the following four states:

- NULL: this is the default state all elements are in when they are created and are doing nothing.
- READY: An element is ready to start doing something.
- PAUSED: The element is paused for a period of time.
- PLAYING: The element is doing something.

All elements start with the NULL state. The elements will go throught the following state changes: NULL -> READY -> PAUSED -> PLAYING. Remember when going from PLAYING to READY, GStreamer will internally go throught the intermediate states.

The state of an element can be changed with the following code:

```
GstElement *bin;

// create a bin, put elements in it and link them
...
gst_element_set_state (bin, GST_STATE_PLAYING);
...
```

You can set the following states to an element:

`GST_STATE_NULL`	Reset the state of an element.
`GST_STATE_READY`	will make the element ready to start processing data.
`GST_STATE_PAUSED`	temporary stops the data flow.
`GST_STATE_PLAYING`	means there really is data flowing through the graph.

The NULL state

When you created the pipeline all of the elements will be in the NULL state. There is nothing spectacular about the NULL state.

> **Note:** Don’t forget to reset the pipeline to the NULL state when you are not going to use it anymore. This will allow the elements to free the resources they might use.

The READY state

You will start the pipeline by first setting it to the READY state. This will allow the pipeline and all the elements contained in it to prepare themselves for the actions they are about to perform.

The typical actions that an element will perform in the READY state might be to open a file or an audio device. Some more complex elements might have a non trivial action to perform in the READY state such as connecting to a media server using a CORBA connection.

> **Note:** You can also go from the NULL to PLAYING state directly without going through the READY state. This is a shortcut; the framework will internally go through the READY and the PAUSED state for you.

The PLAYING state

A Pipeline that is in the READY state can be started by setting it to the PLAYING state. At that time data will start to flow all the way through the pipeline.

The PAUSED state

A pipeline that is playing can be set to the PAUSED state. This will temporarily stop all data flowing through the pipeline.

You can resume the data flow by setting the pipeline back to the PLAYING state.

> **Note:** The PAUSED state is available for temporarily freezing the pipeline. Elements will typically not free their resources in the PAUSED state. Use the NULL state if you want to stop the data flow permanently.

The pipeline has to be in the PAUSED or NULL state if you want to insert or modify an element in the pipeline. We will cover dynamic pipeline behaviour in Chapter 19.

Chapter 12. Your first application

This chapter describes the most rudimentary aspects of a `GStreamer` application, including initializing the libraries, creating elements, packing them into a pipeline and playing, pausing and stopping the pipeline.

Hello world

We will create a simple first application, a complete MP3 player, using standard `GStreamer` components. The player will read from a file that is given as the first argument to the program.

```
/* example-begin helloworld.c */
#include <gst/gst.h>

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

  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_pipeline_new ("pipeline");

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

  /* now it's time to get the decoder */
  decoder = gst_element_factory_make ("mad", "decoder");

  /* and an audio sink */
  audiosink = gst_element_factory_make ("osssink", "play_audio");

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

  /* link src to sink */
  gst_element_link_many (filesrc, decoder, audiosink, NULL);

  /* start playing */
  gst_element_set_state (pipeline, GST_STATE_PLAYING);

  while (gst_bin_iterate (GST_BIN (pipeline)));

  /* stop the pipeline */
  gst_element_set_state (pipeline, GST_STATE_NULL);

  /* we don't need a reference to these objects anymore */
  gst_object_unref (GST_OBJECT (pipeline));
  /* unreffing the pipeline unrefs the contained elements as well */

  exit (0);
}
/* example-end helloworld.c */
```

Let's go through this example step by step.

The first thing you have to do is to include the standard `GStreamer` headers and initialize the framework.

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

  ...

int
main (int argc, char *argv[])
{
  ...
  gst_init(&argc, &argv);
  ...
```

We are going to create three elements and one pipeline. Since all elements share the same base type, `GstElement`, we can define them as:

```
  ...
  GstElement *pipeline, *filesrc, *decoder, *audiosink;
  ...
```

Next, we are going to create an empty pipeline. As you have seen in the basic introduction, this pipeline will hold and manage all the elements we are going to pack into it.

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

We use the standard constructor for a pipeline: gst_pipeline_new ().

We then create a disk source element. The disk source element is able to read from a file. We use the standard GObject property mechanism to set a property of the element: the file to read from.

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

> **Note:** You can check if the filesrc != NULL to verify the creation of the disk source element.

We now create the MP3 decoder element. This assumes that the 'mad' plugin is installed on the system where this application is executed.

```
  /* now it's time to get the decoder */
  decoder = gst_element_factory_make ("mad", "decoder");
```

gst_element_factory_make() takes two arguments: a string that will identify the element you need and a second argument: how you want to name the element. The name of the element is something you can choose yourself and might be used to retrieve the element from a bin/pipeline.

Finally we create our audio sink element. This element will be able to play back the audio using OSS.

```
  /* and an audio sink */
```

```
audiosink = gst_element_factory_make ("audiosink", "play_audio");
```

We then add the elements to the pipeline.

```
/* add objects to the main pipeline */
gst_bin_add_many (GST_BIN (pipeline), filesrc, decoder, audiosink, NULL);
```

We link the different pads of the elements together like this:

```
/* link src to sink */
gst_element_link_many (filesrc, decoder, audiosink, NULL);
```

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

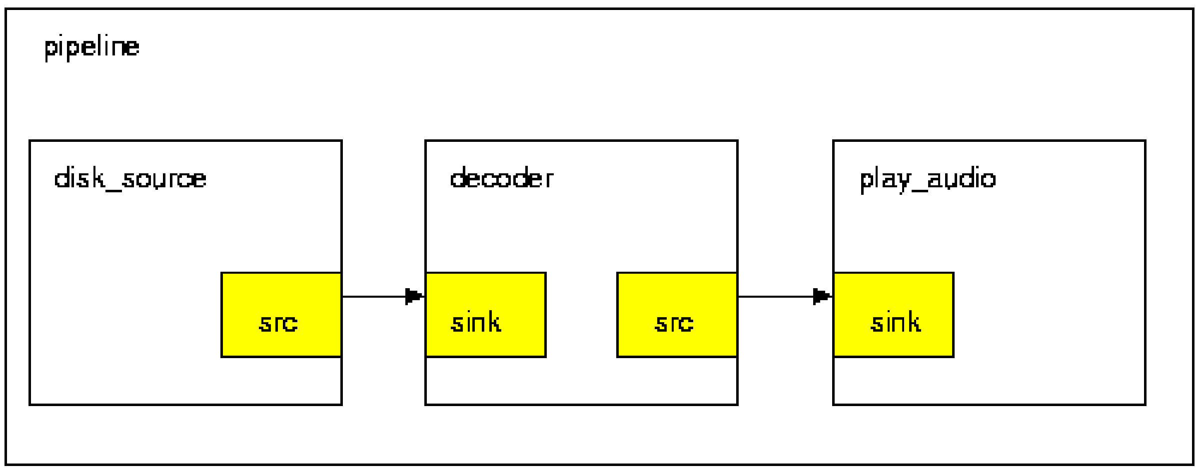

Figure 12-1. The Hello world pipeline

Everything is now set up to start the streaming. We use the following statements to change the state of the pipeline:

```
/* start playing */
gst_element_set_state (pipeline, GST_STATE_PLAYING);
```

> **Note:** `GStreamer` will take care of the READY and PAUSED state for you when going from NULL to PLAYING.

Since we do not use threads, nothing will happen yet. We have to call gst_bin_iterate() to execute one iteration of the pipeline.

```
while (gst_bin_iterate (GST_BIN (pipeline)));
```

The gst_bin_iterate() function will return TRUE as long as something interesting happened inside the pipeline. When the end-of-file has been reached the _iterate function will return FALSE and we can end the loop.

```
/* stop the pipeline */
gst_element_set_state (pipeline, GST_STATE_NULL);
```

```
gst_object_unref (GST_OBJECT (pipeline));

exit (0);
```

Note: Don’t forget to set the state of the pipeline to NULL. This will free all of the resources held by the elements.

Compiling helloworld.c

To compile the helloworld example, use:

```
gcc -Wall ‘pkg-config gstreamer --cflags --libs‘ helloworld.c \
      -o helloworld
```

We use pkg-config to get the compiler flags needed to compile this application. Make sure to have your PKG_CONFIG_PATH environment variable set to the correct location if you are building this application against the uninstalled location.

You can run the example with (substitute helloworld.mp3 with you favorite MP3 file):

```
./helloworld helloworld.mp3
```

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 custom MP3 element with a higher-level API.

It should be clear from the example that we can very easily replace the filesrc element with an httpsrc element, giving you instant network streaming. An element could be built to handle icecast connections, for example.

We can also choose to use another type of sink instead of the audiosink. We could use a disksink to write the raw samples to a file, for example. It should also be clear that inserting filters, like a stereo effect, into the pipeline is not that hard to do. The most important thing is that you can reuse already existing elements.

Chapter 13. More on factories

The small application we created in the previous chapter used the concept of a factory to create the elements. In this chapter we will show you how to use the factory concepts to create elements based on what they do instead of what they are called.

We will first explain the concepts involved before we move on to the reworked helloworld example using autoplugging.

The problems with the helloworld example

If we take a look at how the elements were created in the previous example we used a rather crude mechanism:

```
...
/* now it’s time to get the parser */
decoder = gst_element_factory_make ("mad", "decoder");
...
```

While this mechanism is quite effective it also has some big problems: The elements are created based on their name. Indeed, we create an element, mad, by explicitly stating the mad element’s name. Our little program therefore always uses the mad decoder element to decode the MP3 audio stream, even if there are three other MP3 decoders in the system. We will see how we can use a more general way to create an MP3 decoder element.

We have to introduce the concept of MIME types and capabilities added to the source and sink pads.

More on MIME Types

GStreamer uses MIME types to identify the different types of data that can be handled by the elements. They are the high level mechanisms to make sure that everyone is talking about the right kind of data.

A MIME (Multipurpose Internet Mail Extension) type is a pair of strings that denote a certain type of data. Examples include:

- audio/raw : raw audio samples
- audio/mpeg : MPEG audio
- video/mpeg : MPEG video

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. This allows for very dynamic and extensible element creation as we will see.

As we have seen in the previous chapter, MIME types are added to the Capability structure of a pad.

In our helloworld example the elements we constructed would have the following MIME types associated with their source and sink pads:

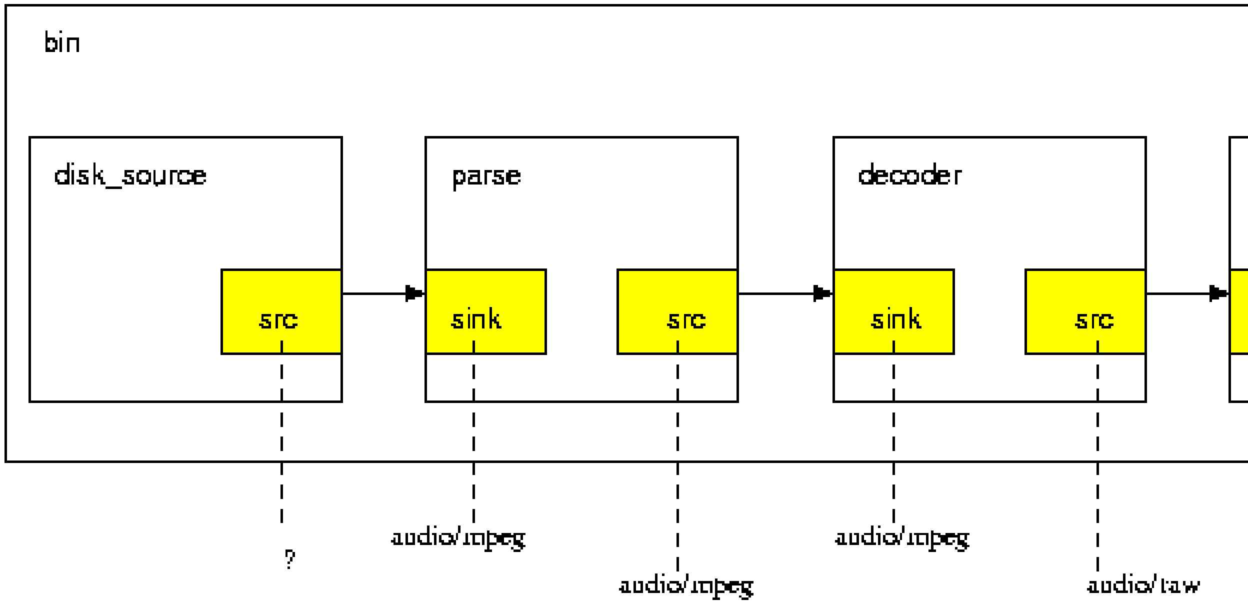

Figure 13-1. The Hello world pipeline with MIME types

We will see how you can create an element based on the MIME types of its source and sink pads. This way the end-user will have the ability to choose his/her favorite audio/mpeg decoder without you even having to care about it.

The typing of the source and sink pads also makes it possible to 'autoplug' a pipeline. We will have the ability to say: "construct me a pipeline that does an audio/mpeg to audio/raw conversion".

> **Note:** The basic GStreamer library does not try to solve all of your autoplug problems. It leaves the hard decisions to the application programmer, where they belong.

GStreamer types

GStreamer assigns a unique number to all registered MIME types. GStreamer also keeps a reference to a function that can be used to determine if a given buffer is of the given MIME type.

There is also an association between a MIME type and a file extension, but the use of typefind functions (similar to file(1)) is preferred.

The type information is maintained in a list of `GstType`. The definition of a `GstType` is like:

```
typedef GstCaps (*GstTypeFindFunc) (GstBuffer *buf,gpointer *priv);

typedef struct _GstType GstType;

struct _GstType {
  guint16 id;                  /* type id (assigned) */

  gchar *mime;                 /* MIME type */
  gchar *exts;                 /* space-delimited list of extensions */

  GstTypeFindFunc typefindfunc; /* typefind function */
};
```

All operations on `GstType` occur via their `guint16 id` numbers, with the `GstType` structure private to the GStreamer library.

MIME type to id conversion

We can obtain the id for a given MIME type with the following piece of code:

```
guint16 id;

id = gst_type_find_by_mime ("audio/mpeg");
```

This function will return 0 if the type was not known.

id to `GstType` conversion

We can obtain the `GstType` for a given id with the following piece of code:

```
GstType *type;

type = gst_type_find_by_id (id);
```

This function will return NULL if the id was not associated with any known `GstType`

extension to id conversion

We can obtain the id for a given file extension with the following piece of code:

```
guint16 id;

id = gst_type_find_by_ext (".mp3");
```

This function will return 0 if the extension was not known.

For more information, see Chapter 21.

Creating elements with the factory

In the previous section we described how you could obtain an element factory using MIME types. One the factory has been obtained, you can create an element using:

```
GstElementFactory *factory;
GstElement *element;

// obtain the factory
factory = ...

element = gst_element_factory_create (factory, "name");
```

This way, you do not have to create elements by name which allows the end-user to select the elements he/she prefers for the given MIME types.

GStreamer basic types

GStreamer only has two builtin types:

- audio/raw : raw audio samples
- video/raw and image/raw : raw video data

All other MIME types are maintained by the plugin elements.

Chapter 14. Threads

GStreamer has support for multithreading through the use of the `GstThread` object. This object is in fact a special `GstBin` that will become a thread when started.

To construct a new thread you will perform something like:

```
GstElement *my_thread;

/* create the thread object */
my_thread = gst_thread_new ("my_thread");
/* you could have used gst_element_factory_make ("thread", "my_thread"); */
g_return_if_fail (my_thread != NULL);

/* add some plugins */
gst_bin_add (GST_BIN (my_thread), GST_ELEMENT (funky_src));
gst_bin_add (GST_BIN (my_thread), GST_ELEMENT (cool_effect));

/* link the elements here... */
...

/* start playing */
gst_element_set_state (GST_ELEMENT (my_thread), GST_STATE_PLAYING);
```

The above program will create a thread with two elements in it. As soon as it is set to the PLAYING state, the thread will start to iterate itself. You never need to explicitly iterate a thread.

Constraints placed on the pipeline by the GstThread

Within the pipeline, everything is the same as in any other bin. The difference lies at the thread boundary, at the link between the thread and the outside world (containing bin). Since GStreamer is fundamentally buffer-oriented rather than byte-oriented, the natural solution to this problem is an element that can "buffer" the buffers between the threads, in a thread-safe fashion. This element is the queue, described more fully in Chapter 15. It doesn't matter if the queue is placed in the containing bin or in the thread itself, but it needs to be present on one side or the other to enable inter-thread communication.

When would you want to use a thread?

If you are writing a GUI application, making the top-level bin a thread will make your GUI more responsive. If it were a pipeline instead, it would have to be iterated by your application's event loop, which increases the latency between events (say, keyboard presses) and responses from the GUI. In addition, any slight hang in the GUI would delay iteration of the pipeline, which (for example) could cause pops in the output of the sound card, if it is an audio pipeline.

A thread can be visualised as below

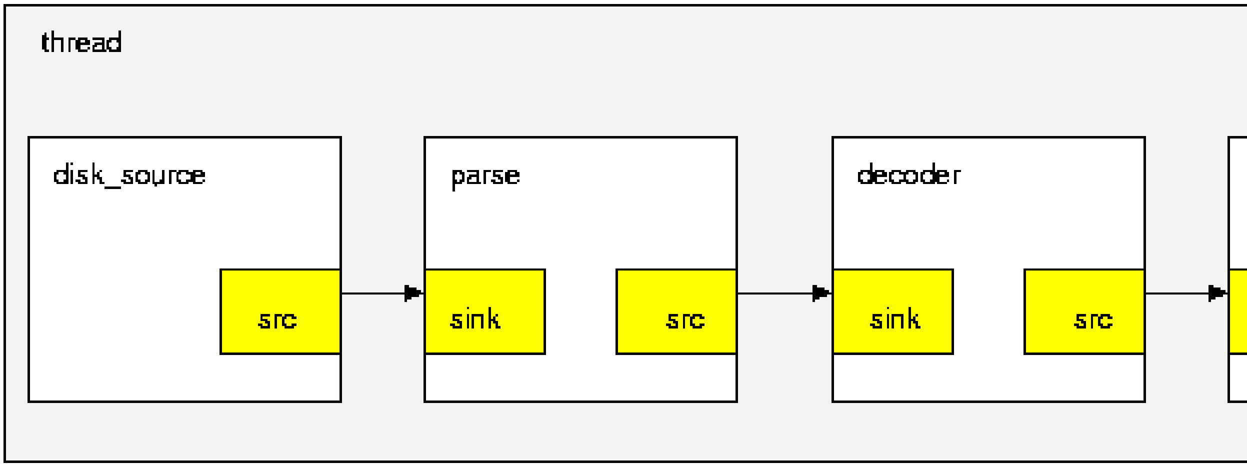

Figure 14-1. A thread

As an example we show the helloworld program using a thread.

```
/* example-begin threads.c */
#include <gst/gst.h>

/* we set this to TRUE right before gst_main (), but there could still
   be a race condition between setting it and entering the function */
gboolean can_quit = FALSE;

/* eos will be called when the src element has an end of stream */
void
eos (GstElement *src, gpointer data)
{
  GstThread *thread = GST_THREAD (data);
  g_print ("have eos, quitting\n");

  /* stop the bin */
  gst_element_set_state (GST_ELEMENT (thread), GST_STATE_NULL);

  while (!can_quit) /* waste cycles */ ;
  gst_main_quit ();
}

int
main (int argc, char *argv[])
{
  GstElement *filesrc, *decoder, *audiosink;
  GstElement *thread;

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

  gst_init (&argc, &argv);

  /* create a new thread to hold the elements */
  thread = gst_thread_new ("thread");
  g_assert (thread != 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);
  g_signal_connect (G_OBJECT (filesrc), "eos",
```

```
                    G_CALLBACK (eos), thread);

  /* create an ogg decoder */
  decoder = gst_element_factory_make ("vorbisfile", "decoder");
  g_assert (decoder != NULL);

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

  /* add objects to the thread */
  gst_bin_add_many (GST_BIN (thread), filesrc, decoder, audiosink, NULL);
  /* link them in the logical order */
  gst_element_link_many (filesrc, decoder, audiosink, NULL);

  /* start playing */
  gst_element_set_state (GST_ELEMENT (thread), GST_STATE_PLAYING);

  /* do whatever you want here, the thread will be playing */
  g_print ("thread is playing\n");

  can_quit = TRUE;
  gst_main ();

  gst_pipeline_destroy (thread);

  exit (0);
}
/* example-end threads.c */
```

Chapter 15. Queues

A `GstQueue` is a filter element. Queues can be used to link two elements in such way that the data can be buffered.

A buffer that is sinked to a Queue will not automatically be pushed to the next linked element but will be buffered. It will be pushed to the next element as soon as a gst_pad_pull () is called on the queue's source pad.

Queues are mostly used in conjunction with a `GstThread` to provide an external link for the thread elements. You could have one thread feeding buffers into a `GstQueue` and another thread repeadedly calling gst_pad_pull () on the queue to feed its internal elements.

Below is a figure of a two-threaded decoder. We have one thread (the main execution thread) reading the data from a file, and another thread decoding the data.

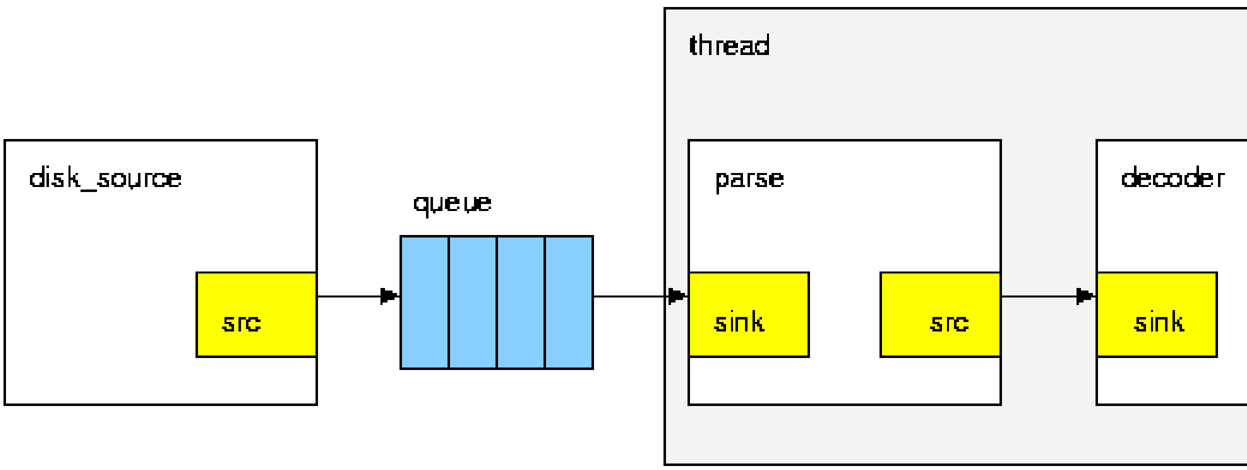

Figure 15-1. a two-threaded decoder with a queue

The standard `GStreamer` queue implementation has some properties that can be changed using the g_objet_set () method. To set the maximum number of buffers that can be queued to 30, do:

```
  g_object_set (G_OBJECT (queue), "max_level", 30, NULL);
```

The following MP3 player shows you how to create the above pipeline using a thread and a queue.

```
/* example-begin queue.c */
#include <stdlib.h>
#include <gst/gst.h>

gboolean playing;

/* eos will be called when the src element has an end of stream */
void
eos (GstElement *element, gpointer data)
{
  g_print ("have eos, quitting\n");

  playing = FALSE;
}

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

```
{
  GstElement *filesrc, *audiosink, *queue, *decode;
  GstElement *bin;
  GstElement *thread;

  gst_init (&argc,&argv);

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

  /* create a new thread to hold the elements */
  thread = gst_thread_new ("thread");
  g_assert (thread != NULL);

  /* create a new bin to hold the elements */
  bin = gst_bin_new ("bin");
  g_assert (bin != 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);
  g_signal_connect (G_OBJECT (filesrc), "eos",
                    G_CALLBACK (eos), thread);

  queue = gst_element_factory_make ("queue", "queue");
  g_assert (queue != NULL);

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

  decode = gst_element_factory_make ("mad", "decode");

  /* add objects to the main bin */
  gst_bin_add_many (GST_BIN (thread), decode, audiosink, NULL);

  gst_bin_add_many (GST_BIN (bin), filesrc, queue, thread, NULL);


  gst_element_link (filesrc, queue);
  gst_element_link_many (queue, decode, audiosink, NULL);

  /* start playing */
  gst_element_set_state (GST_ELEMENT (bin), GST_STATE_PLAYING);

  playing = TRUE;

  while (playing) {
    gst_bin_iterate (GST_BIN (bin));
  }

  gst_element_set_state (GST_ELEMENT (bin), GST_STATE_NULL);

  return 0;
}
/* example-end queue.c */
```

Chapter 16. Cothreads

Cothreads are user-space threads that greatly reduce context switching overhead introduced by regular kernel threads. Cothreads are also used to handle the more complex elements. They differ from other user-space threading libraries in that they are scheduled explictly by GStreamer.

A cothread is created by a `GstBin` whenever an element is found inside the bin that has one or more of the following properties:

- The element is loop-based instead of chain-based
- The element has multiple input pads
- The element has the MULTI_IN flag set

The `GstBin` will create a cothread context for all the elements in the bin so that the elements will interact in cooperative multithreading.

Before proceding to the concept of loop-based elements we will first explain the chain-based elements.

Chain-based elements

Chain based elements receive a buffer of data and are supposed to handle the data and perform a gst_pad_push.

The basic main function of a chain-based element is like:

```
static void
chain_function (GstPad *pad, GstBuffer *buffer)
{
  GstBuffer *outbuffer;

  ....
  // process the buffer, create a new outbuffer
  ...

  gst_pad_push (srcpad, outbuffer);
}
```

Chain based function are mainly used for elements that have a one to one relation between their input and output behaviour. An example of such an element can be a simple video blur filter. The filter takes a buffer in, performs the blur operation on it and sends out the resulting buffer.

Another element, for example, is a volume filter. The filter takes audio samples as input, performs the volume effect and sends out the resulting buffer.

Loop-based elements

As opposed to chain-based elements, loop-based elements enter an infinite loop that looks like this:

```
  GstBuffer *buffer, *outbuffer;

  while (1) {
    buffer = gst_pad_pull (sinkpad);
    ...
    // process buffer, create outbuffer
    while (!done) {
      ....
      // optionally request another buffer
```

```
      buffer = gst_pad_pull (sinkpad);
      ....
    }
    ...
    gst_pad_push (srcpad, outbuffer);
  }
```

The loop-based elements request a buffer whenever they need one.

When the request for a buffer cannot immediatly satisfied, the control will be given to the source element of the loop-based element until it performs a push on its source pad. At that time the control is handed back to the loop-based element, etc... The execution trace can get fairly complex using cothreads when there are multiple input/output pads for the loop-based element. Cothread switches are performed within the call to gst_pad_pull and gst_pad_push; from the perspective of the loop-based element, it just "appears" that gst_pad_push (or _pull) might take a long time to return.

Loop based elements are mainly used for the more complex elements that need a specific amount of data before they can start to produce output. An example of such an element is the MPEG video decoder. The element will pull a buffer, perform some decoding on it and optionally request more buffers to decode, and when a complete video frame has been decoded, a buffer is sent out. For example, any plugin using the bytestream library will need to be loop-based.

There is no problem in putting cothreaded elements into a `GstThread` to create even more complex pipelines with both user and kernel space threads.

Chapter 17. Understanding schedulers

The scheduler is responsible for managing the plugins at runtime. Its main responsibilities are:

- Preparing the plugins so they can be scheduled.
- Monitoring state changes and enabling/disabling the element in the chain.
- Choosing an element as the entry point for the pipeline.
- Selecting and distributing the global clock.

The scheduler is a plugable component; this means that alternative schedulers can be written and plugged into GStreamer. The default scheduler uses cothreads to schedule the plugins in a pipeline. Cothreads are fast and lightweight user-space threads.

There is usually no need to interact with the scheduler directly, however in some cases it is feasible to set a specific clock or force a specific plugin as the entry point in the pipeline.

Chapter 18. Clocks in GStreamer

Chapter 19. Dynamic pipelines

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. Dynamic pipelines cause all sorts of scheduling issues and will remain a topic of research for a long time in GStreamer.

We will show how to create an MPEG1 video player using dynamic pipelines. As you have seen in the pad section, we can attach a signal to an element when a pad is created. We will use this to create our MPEG1 player.

We'll start with a simple main function:

```
/* example-begin dynamic.c */
#include <string.h>
#include <gst/gst.h>

void
eof (GstElement *src)
{
  g_print ("have eos, quitting\n");
  exit (0);
}

gboolean
idle_func (gpointer data)
{
  gst_bin_iterate (GST_BIN (data));
  return TRUE;
}

void
new_pad_created (GstElement *parse, GstPad *pad, GstElement *pipeline)
{
  GstElement *decode_video = NULL;
  GstElement *decode_audio, *play, *color, *show;
  GstElement *audio_queue, *video_queue;
  GstElement *audio_thread, *video_thread;

  g_print ("***** a new pad %s was created\n", gst_pad_get_name (pad));

  gst_element_set_state (GST_ELEMENT (pipeline), GST_STATE_PAUSED);

  /* link to audio pad */
  if (strncmp (gst_pad_get_name (pad), "audio_", 6) == 0) {

    /* construct internal pipeline elements */
    decode_audio = gst_element_factory_make ("mad", "decode_audio");
    g_return_if_fail (decode_audio != NULL);
    play = gst_element_factory_make ("osssink", "play_audio");
    g_return_if_fail (play != NULL);

    /* create the thread and pack stuff into it */
    audio_thread = gst_thread_new ("audio_thread");
    g_return_if_fail (audio_thread != NULL);

    /* construct queue and link everything in the main pipeline */
    audio_queue = gst_element_factory_make ("queue", "audio_queue");
    g_return_if_fail (audio_queue != NULL);

    gst_bin_add_many (GST_BIN (audio_thread),
                      audio_queue, decode_audio, play, NULL);

    /* set up pad links */
```

```
    gst_element_add_ghost_pad (audio_thread,
                               gst_element_get_pad (audio_queue, "sink"),
                               "sink");
    gst_element_link (audio_queue, decode_audio);
    gst_element_link (decode_audio, play);

    gst_bin_add (GST_BIN (pipeline), audio_thread);

    gst_pad_link (pad, gst_element_get_pad (audio_thread, "sink"));

    /* set up thread state and kick things off */
    g_print ("setting to READY state\n");
    gst_element_set_state (GST_ELEMENT (audio_thread), GST_STATE_READY);

  }
  else if (strncmp (gst_pad_get_name (pad), "video_", 6) == 0) {

    /* construct internal pipeline elements */
    decode_video = gst_element_factory_make ("mpeg2dec", "decode_video");
    g_return_if_fail (decode_video != NULL);

    color = gst_element_factory_make ("colorspace", "color");
    g_return_if_fail (color != NULL);


    show = gst_element_factory_make ("xvideosink", "show");
    g_return_if_fail (show != NULL);

    /* construct queue and link everything in the main pipeline */
    video_queue = gst_element_factory_make ("queue", "video_queue");
    g_return_if_fail (video_queue != NULL);

    /* create the thread and pack stuff into it */
    video_thread = gst_thread_new ("video_thread");
    g_return_if_fail (video_thread != NULL);
    gst_bin_add_many (GST_BIN (video_thread), video_queue,
                      decode_video, color, show, NULL);

    /* set up pad links */
    gst_element_add_ghost_pad (video_thread,
                               gst_element_get_pad (video_queue, "sink"),
                               "sink");
    gst_element_link (video_queue, decode_video);
    gst_element_link_many (decode_video, color, show, NULL);

    gst_bin_add (GST_BIN (pipeline), video_thread);

    gst_pad_link (pad, gst_element_get_pad (video_thread, "sink"));

    /* set up thread state and kick things off */
    g_print ("setting to READY state\n");
    gst_element_set_state (GST_ELEMENT (video_thread), GST_STATE_READY);
  }
  gst_element_set_state (GST_ELEMENT (pipeline), GST_STATE_PLAYING);
}

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

  gst_init (&argc, &argv);

  pipeline = gst_pipeline_new ("pipeline");
  g_return_val_if_fail (pipeline != NULL, -1);
```

```
  src = gst_element_factory_make ("filesrc", "src");
  g_return_val_if_fail (src != NULL, -1);
  if (argc < 2)
    g_error ("Please specify a video file to play !");

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

  demux = gst_element_factory_make ("mpegdemux", "demux");
  g_return_val_if_fail (demux != NULL, -1);

  gst_bin_add_many (GST_BIN (pipeline), src, demux, NULL);

  g_signal_connect (G_OBJECT (demux), "new_pad",
                    G_CALLBACK (new_pad_created), pipeline);

  g_signal_connect (G_OBJECT (src), "eos",
                    G_CALLBACK (eof), NULL);

  gst_element_link (src, demux);

  gst_element_set_state (GST_ELEMENT (pipeline), GST_STATE_PLAYING);

  g_idle_add (idle_func, pipeline);

  gst_main ();

  return 0;
}
/* example-end dynamic.c */
```

We create two elements: a file source and an MPEG demuxer. There's nothing special about this piece of code except for the signal 'new_pad' that we linked to the mpegdemux element using:

```
  g_signal_connect (G_OBJECT (demux), "new_pad",
                    G_CALLBACK (new_pad_created), pipeline);
```

When an elementary stream has been detected in the system stream, mpegdemux will create a new pad that will provide the data of the elementary stream. A function 'new_pad_created' will be called when the pad is created.

In the above example, we created new elements based on the name of the newly created pad. We then added them to a new thread. There are other possibilities to check the type of the pad, for example by using the MIME type and the properties of the pad.

Note that the pipeline has to be in the PAUSED state before changes can be made to its structure.

Chapter 20. Typedetection

Sometimes the capabilities of a pad are not specificied. The filesrc element, for example, does not know what type of file it is reading. Before you can attach an element to the pad of the filesrc, you need to determine the media type in order to be able to choose a compatible element.

To solve this problem, a plugin can provide the `GStreamer` core library with a typedefinition library with a typedefinition. The typedefinition will contain the following information:

- The MIME type we are going to define.
- An optional string with a list of possible file extensions this type usually is associated with. the list entries are separated with a space. eg, ".mp3 .mpa .mpg".
- An optional typefind function.

The typefind functions give a meaning to the MIME types that are used in GStreamer. The typefind function is a function with the following definition:

```
typedef GstCaps *(*GstTypeFindFunc) (GstBuffer *buf, gpointer priv);
```

This typefind function will inspect a GstBuffer with data and will output a GstCaps structure describing the type. If the typefind function does not understand the buffer contents, it will return NULL.

`GStreamer` has a typefind element in its core elements that can be used to determine the type of a given pad.

The next example will show how a typefind element can be inserted into a pipeline to detect the media type of a file. It will output the capabilities of the pad into an XML representation.

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

void    type_found      (GstElement *typefind, GstCaps* caps);

int
main(int argc, char *argv[])
{
  GstElement *bin, *filesrc, *typefind;

  gst_init (&argc, &argv);

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

  /* create a new bin to hold the elements */
  bin = gst_bin_new ("bin");
  g_assert (bin != 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);

  /* create the typefind element */
  typefind = gst_element_factory_make ("typefind", "typefind");
  g_assert (typefind != NULL);
```

```
  /* add objects to the main pipeline */
  gst_bin_add (GST_BIN (bin), filesrc);
  gst_bin_add (GST_BIN (bin), typefind);

  g_signal_connect (G_OBJECT (typefind), "have_type",
       G_CALLBACK (type_found), NULL);

  gst_pad_link (gst_element_get_pad (filesrc, "src"),
                gst_element_get_pad (typefind, "sink"));

  /* start playing */
  gst_element_set_state (GST_ELEMENT (bin), GST_STATE_PLAYING);

  gst_bin_iterate (GST_BIN (bin));

  gst_element_set_state (GST_ELEMENT (bin), GST_STATE_NULL);

  exit (0);
}
```

We create a very simple pipeline with only a filesrc and the typefind element in it. The sinkpad of the typefind element has been linked to the source pad of the filesrc.

We attached a signal 'have_type' to the typefind element which will be called when the type of the media stream as been detected.

The typefind function will loop over all the registered types and will execute each of the typefind functions. As soon as a function returns a GstCaps pointer, the type_found function will be called:

```
void
type_found (GstElement *typefind, GstCaps* caps)
{
  xmlDocPtr doc;
  xmlNodePtr parent;

  doc = xmlNewDoc ("1.0");
  doc->root = xmlNewDocNode (doc, NULL, "Capabilities", NULL);

  parent = xmlNewChild (doc->root, NULL, "Caps1", NULL);
  gst_caps_save_thyself (caps, parent);

  xmlDocDump (stdout, doc);
}
```

In the type_found function we can print or inspect the type that has been detected using the GstCaps APIs. In this example, we just print out the XML representation of the caps structure to stdout.

A more useful option would be to use the registry to look up an element that can handle this particular caps structure, or we can also use the autoplugger to link this caps structure to, for example, a videosink.

Chapter 21. Autoplugging

GStreamer provides an API to automatically construct complex pipelines based on source and destination capabilities. This feature is very useful if you want to convert type X to type Y but don't care about the plugins needed to accomplish this task. The autoplugger will consult the plugin repository, select and link the elements needed for the conversion.

The autoplugger API is implemented in an abstract class. Autoplugger implementations reside in plugins and are therefore optional and can be optimized for a specific task. Two types of autopluggers exist: renderer ones and non-renderer ones. The renderer autopluggers will not have any source pads while the non-renderer ones do. The renderer autopluggers are mainly used for media playback while the non renderer ones are used for arbitrary format conversion.

Using autoplugging

You first need to create a suitable autoplugger with gst_autoplug_factory_make(). The name of the autoplugger must be one of the registered autopluggers..

A list of all available autopluggers can be obtained with gst_autoplug_factory_get_list().

If the autoplugger supports the RENDERER API, use the gst_autoplug_to_renderers() function to create a bin that links the source caps to the specified render elements. You can then add the bin to a pipeline and run it.

```
       GstAutoplug *autoplug;
GstElement  *element;
GstElement  *sink;

/* create a static autoplugger */
autoplug = gst_autoplug_factory_make ("staticrender");

/* create an osssink */
sink = gst_element_factory_make ("osssink", "our_sink");

/* create an element that can play audio/mp3 through osssink */
element = gst_autoplug_to_renderers (autoplug,
                                     gst_caps_new (
                                       "sink_audio_caps",
                                       "audio/mp3",
                                       NULL
                                     ),
                                     sink,
                                     NULL);

/* add the element to a bin and link the sink pad */
...
```

If the autoplugger supports the CAPS API, use the gst_autoplug_to_caps() function to link the source caps to the destination caps. The created bin will have source and sink pads compatible with the provided caps.

```
       GstAutoplug *autoplug;
GstElement  *element;

/* create a static autoplugger */
autoplug = gst_autoplug_factory_make ("static");
```

```
/* create an element that converts audio/mp3 to audio/raw */
element = gst_autoplug_to_caps (autoplug,
                                gst_caps_new (
                                  "sink_audio_caps",
                                  "audio/mp3",
                                  NULL
                                ),
                                gst_caps_new (
                                  "src_audio_caps",
                                  "audio/raw",
                                  NULL
                                ),
                                NULL);

/* add the element to a bin and link the src/sink pads */
...
```

Using the `GstAutoplugCache` element

The `GstAutoplugCache` element is used to cache the media stream when performing typedetection. As we have have seen in the previous chapter (typedetection), the type typefind function consumes a buffer to determine the media type of it. After we have set up the pipeline to play the media stream we should be able to 'replay' the previous buffer(s). This is where the autoplugcache is used for.

The basic usage pattern for the autoplugcache in combination with the typefind element is like this:

1. Add the autoplugcache element to a bin and link the sink pad to the source pad of an element with unknown caps.
2. Link the source pad of the autoplugcache to the sink pad of the typefind element.
3. Iterate the pipeline until the typefind element has found a type.
4. Remove the typefind element and add the plugins needed to play back the discovered media type to the autoplugcache source pad.
5. Reset the cache to start playback of the cached data. Connect to the "cache_empty" signal.
6. In the cache_empty signal callback function, remove the autoplugcache and relink the pads.

In the next chapter we will create a new version of our helloworld example using the autoplugger, the autoplugcache and the typefind element.

Another approach to autoplugging

The autoplug API is interesting, but often impractical. It is static; it cannot deal with dynamic pipelines (insert ref here). What you often want is just an element to stick into a pipeline that will DWIM (Do What I Mean)(ref). Enter the spider.

The spider element

The spider element is a generalized autoplugging element. At this point (April 2002), it's the best we've got; it can be inserted anywhere within a pipeline to perform caps conversion, if possible. Consider the following gst-launch line:

```
$ gst-launch filesrc location=my.mp3 ! spider ! osssink
```

The spider will detect the type of the stream, autoplug it to the osssink's caps, and play the pipeline. It's neat.

Spider features

1. Automatically typefinds the incoming stream.
2. Has request pads on the source side. This means that it can autoplug one source stream into many sink streams. For example, an MPEG1 system stream can have audio as well as video; that pipeline would be represented in gst-launch syntax as

```
$ gst-launch filesrc location=my.mpeg1 ! spider ! { queue ! osssin
             { queue ! xvideosink }
```

Chapter 22. Your second application

FIXME: delete this section, talk more about the spider. In a previous chapter we created a first version of the helloworld application. We then explained a better way of creating the elements using factories identified by MIME types and the autoplugger.

Autoplugging helloworld

We will create a second version of the helloworld application using autoplugging. Its source code is a bit more complicated but it can handle many more data types. It can even play the audio track of a video file.

Here is the full program listing. Start by looking at the main () function.

```
/* example-begin helloworld2.c */
#include <gst/gst.h>

static void  gst_play_have_type  (GstElement *typefind, GstCaps *caps, GstElement *pipe
static void  gst_play_cache_empty  (GstElement *element, GstElement *pipeline);

static void
gst_play_have_type (GstElement *typefind, GstCaps *caps, GstElement *pipeline)
{
  GstElement *osssink;
  GstElement *new_element;
  GstAutoplug *autoplug;
  GstElement *autobin;
  GstElement *filesrc;
  GstElement *cache;

  g_print ("GstPipeline: play have type\n");

  gst_element_set_state (pipeline, GST_STATE_PAUSED);

  filesrc = gst_bin_get_by_name (GST_BIN (pipeline), "disk_source");
  autobin = gst_bin_get_by_name (GST_BIN (pipeline), "autobin");
  cache = gst_bin_get_by_name (GST_BIN (autobin), "cache");

  /* unlink the typefind from the pipeline and remove it */
  gst_element_unlink (cache, typefind);
  gst_bin_remove (GST_BIN (autobin), typefind);

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

  autoplug = gst_autoplug_factory_make ("staticrender");
  g_assert (autoplug != NULL);

  new_element = gst_autoplug_to_renderers (autoplug, caps, osssink, NULL);

  if (!new_element) {
    g_print ("could not autoplug, no suitable codecs found...\n");
    exit (-1);
  }

  gst_element_set_name (new_element, "new_element");

  gst_bin_add (GST_BIN (autobin), new_element);

  g_object_set (G_OBJECT (cache), "reset", TRUE, NULL);

  gst_element_link (cache, new_element);

  gst_element_set_state (pipeline, GST_STATE_PLAYING);
```

```
}

static void
gst_play_cache_empty (GstElement *element, GstElement *pipeline)
{
  GstElement *autobin;
  GstElement *filesrc;
  GstElement *cache;
  GstElement *new_element;

  g_print ("have cache empty\n");

  gst_element_set_state (pipeline, GST_STATE_PAUSED);

  filesrc = gst_bin_get_by_name (GST_BIN (pipeline), "disk_source");
  autobin = gst_bin_get_by_name (GST_BIN (pipeline), "autobin");
  cache = gst_bin_get_by_name (GST_BIN (autobin), "cache");
  new_element = gst_bin_get_by_name (GST_BIN (autobin), "new_element");

  gst_element_unlink (filesrc, cache);
  gst_element_unlink (cache, new_element);
  gst_bin_remove (GST_BIN (autobin), cache);
  gst_element_link (filesrc, new_element);

  gst_element_set_state (pipeline, GST_STATE_PLAYING);

  g_print ("done with cache_empty\n");
}

int
main (int argc, char *argv[])
{
  GstElement *filesrc;
  GstElement *pipeline;
  GstElement *autobin;
  GstElement *typefind;
  GstElement *cache;

  gst_init (&argc, &argv);

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

  /* create a new pipeline to hold the elements */
  pipeline = gst_pipeline_new ("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);
  gst_bin_add (GST_BIN (pipeline), filesrc);

  autobin = gst_bin_new ("autobin");
  cache = gst_element_factory_make ("autoplugcache", "cache");
  g_signal_connect (G_OBJECT (cache), "cache_empty",
       G_CALLBACK (gst_play_cache_empty), pipeline);

  typefind = gst_element_factory_make ("typefind", "typefind");
  g_signal_connect (G_OBJECT (typefind), "have_type",
       G_CALLBACK (gst_play_have_type), pipeline);
  gst_bin_add (GST_BIN (autobin), cache);
  gst_bin_add (GST_BIN (autobin), typefind);
```

```
  gst_element_link (cache, typefind);
  gst_element_add_ghost_pad (autobin,
                             gst_element_get_pad (cache, "sink"), "sink");

  gst_bin_add (GST_BIN( pipeline), autobin);
  gst_element_link (filesrc, autobin);

  /* start playing */
  gst_element_set_state( GST_ELEMENT (pipeline), GST_STATE_PLAYING);

  while (gst_bin_iterate (GST_BIN (pipeline)));

  /* stop the pipeline */
  gst_element_set_state (GST_ELEMENT (pipeline), GST_STATE_NULL);

  gst_object_unref (GST_OBJECT (pipeline));

  exit(0);
}
/* example-end helloworld2.c */
```

We start by constructing a 'filesrc' element and an 'autobin' element that holds the autoplugcache and the typefind element.

We attach the "cache_empty" signal to gst_play_cache_empty and the "have_type" to our gst_play_have_type function.

The _have_type function first sets the pipeline to the PAUSED state so that it can safely modify the pipeline. It then finds the elements it is going to manipulate in the pipeline with:

```
  filesrc = gst_bin_get_by_name (GST_BIN (pipeline), "disk_source");
  autobin = gst_bin_get_by_name (GST_BIN (pipeline), "autobin");
  cache = gst_bin_get_by_name (GST_BIN (autobin), "cache");
```

Now we have a handle to the elements we are going to manipulate in the next step.

We don't need the typefind element anymore so we remove it from the pipeline:

```
  /* unlink the typefind from the pipeline and remove it */
  gst_element_unlink (cache, "src", typefind, "sink");
  gst_bin_remove (GST_BIN (autobin), typefind);
```

Our next step is to construct an element that can play the type we just detected. We are going to use the autoplugger to create an element that links the type to an osssink. We add the new element to our autobin.

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

  autoplug = gst_autoplug_factory_make ("staticrender");
  g_assert (autoplug != NULL);

  new_element = gst_autoplug_to_renderers (autoplug,
           caps,
           osssink,
           NULL);

  if (!new_element) {
    g_print ("could not autoplug, no suitable codecs found...\n");
    exit (-1);
  }
```

```
gst_element_set_name (new_element, "new_element");

gst_bin_add (GST_BIN (autobin), new_element);
```

Our next step is to reset the cache so that the buffers used by the typefind element are fed into the new element we just created. We reset the cache by setting the "reset" property of the cache element to TRUE.

```
g_object_set (G_OBJECT (cache), "reset", TRUE, NULL);

gst_element_link (cache, "src", new_element, "sink");
```

Finally we set the pipeline back to the playing state. At this point the cache will replay the buffers. We will be notified when the cache is empty by the gst_play_cache_empty callback function.

The cache empty function simply removes the autoplugcache element from the pipeline and relinks the filesrc to the autoplugged element.

To compile the helloworld2 example, use:

```
gcc -Wall ‘pkg-config gstreamer --cflags --libs‘ helloworld2.c \
      -o helloworld2
```

You can run the example with (substitute helloworld.mp3 with you favorite audio file):

```
./helloworld2 helloworld.mp3
```

You can also try to use an AVI or MPEG file as its input. Using autoplugging, `GStreamer` will automatically figure out how to handle the stream. Remember that only the audio part will be played because we have only added an osssink to the pipeline.

```
./helloworld2 mymovie.mpeg
```

Chapter 23. Dynamic Parameters

Getting Started

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

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

Your application should link to the shared library `gstcontrol`.

The `gstcontrol` 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_control_init(&argc,&argv);
  ...
```

Creating and Attaching Dynamic Parameters

Once you have created your elements you can create and attach dparams to them. First you need to get the element's dparams manager. If you know exactly what kind of element you have, you may be able to get the dparams manager directly. However if this is not possible, you can get the dparams manager by calling `gst_dpman_get_manager`.

Once you have the dparams manager, you must set the mode that the manager will run in. There is currently only one mode implemented called `"synchronous"` - this is used for real-time applications where the dparam value cannot be known ahead of time (such as a slider in a GUI). The mode is called `"synchronous"` because the dparams are polled by the element for changes before each buffer is processed. Another yet-to-be-implemented mode is `"asynchronous"`. This is used when parameter changes are known ahead of time - such as with a timelined editor. The mode is called `"asynchronous"` because parameter changes may happen in the middle of a buffer being processed.

```
  GstElement *sinesrc;
  GstDParamManager *dpman;
  ...
  sinesrc = gst_element_factory_make("sinesrc","sine-source");
  ...
  dpman = gst_dpman_get_manager (sinesrc);
  gst_dpman_set_mode(dpman, "synchronous");
```

If you don't know the names of the required dparams for your element you can call `gst_dpman_list_dparam_specs(dpman)` to get a NULL terminated array of param specs. This array should be freed after use. You can find the name of the required dparam by calling `g_param_spec_get_name` on each param spec in the array. In our example, `"volume"` will be the name of our required dparam.

Each type of dparam currently has its own `new` function. This may eventually be replaced by a factory method for creating new instances. A default dparam instance can be created with the `gst_dparam_new` function. Once it is created it can be attached to a required dparam in the element.

```
  GstDParam *volume;
  ...
  volume = gst_dparam_new(G_TYPE_FLOAT);
  if (gst_dpman_attach_dparam (dpman, "volume", volume)){
    /* the dparam was successfully attached */
    ...
  }
```

Changing Dynamic Parameter Values

All interaction with dparams to actually set the dparam value is done through simple GObject properties. There is a property value for each type that dparams supports - these currently being `"value_float"`, `"value_int"` and `"value_int64"`. To set the value of a dparam, simply set the property which matches the type of your dparam instance.

```
#define ZERO(mem) memset(&mem, 0, sizeof(mem))
...

  gfloat set_to_value;
  GstDParam *volume;
  GValue set_val;
  ZERO(set_val);
  g_value_init(&set_val, G_TYPE_FLOAT);
  ...
  g_value_set_float(&set_val, set_to_value);
  g_object_set_property(G_OBJECT(volume), "value_float", &set_val);
```

Or if you create an actual GValue instance:

```
  gfloat set_to_value;
  GstDParam *volume;
  GValue *set_val;
  set_val = g_new0(GValue,1);
  g_value_init(set_val, G_TYPE_FLOAT);
  ...
  g_value_set_float(set_val, set_to_value);
  g_object_set_property(G_OBJECT(volume), "value_float", set_val);
```

Different Types of Dynamic Parameter

There are currently only two implementations of dparams so far. They are both for real-time use so should be run in the `"synchronous"` mode.

GstDParam - the base dparam type

All dparam implementations will subclass from this type. It provides a basic implementation which simply propagates any value changes as soon as it can. A new instance can be created with the function `GstDParam* gst_dparam_new (GType type)`. It has the following object properties:

- `"value_float"` - the property to set and get if it is a float dparam
- `"value_int"` - the property to set and get if it is an integer dparam
- `"value_int64"` - the property to set and get if it is a 64 bit integer dparam

- `"is_log"` - readonly boolean which is TRUE if the param should be displayed on a log scale
- `"is_rate"` - readonly boolean which is TRUE if the value is a proportion of the sample rate. For example with a sample rate of 44100, 0.5 would be 22050 Hz and 0.25 would be 11025 Hz.

GstDParamSmooth - smoothing real-time dparam

Some parameter changes can create audible artifacts if they change too rapidly. The GstDParamSmooth implementation can greatly reduce these artifacts by limiting the rate at which the value can change. This is currently only supported for float dparams - the other types fall back to the default implementation. A new instance can be created with the function `GstDParam* gst_dpsmooth_new (GType type)`. It has the following object properties:

- `"update_period"` - an int64 value specifying the number nanoseconds between updates. This will be ignored in `"synchronous"` mode since the buffer size dictates the update period.
- `"slope_time"` - an int64 value specifying the time period to use in the maximum slope calculation
- `"slope_delta_float"` - a float specifying the amount a float value can change in the given slope_time.

Audible artifacts may not be completely eliminated by using this dparam. The only way to eliminate artifacts such as "zipper noise" would be for the element to implement its required dparams using the array method. This would allow dparams to change parameters at the sample rate which should eliminate any artifacts.

Timelined dparams

A yet-to-be-implemented subclass of GstDParam will add an API which allows the creation and manipulation of points on a timeline. This subclass will also provide a dparam implementation which uses linear interpolation between these points to find the dparam value at any given time. Further subclasses can extend this functionality to implement more exotic interpolation algorithms such as splines.

Chapter 24. XML in `GStreamer`

`GStreamer` uses XML to store and load its pipeline definitions. XML is also used internally to manage the plugin registry. The plugin registry is a file that contains the definition of all the plugins `GStreamer` knows about to have quick access to the specifics of the plugins.

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 with two threads and then writes out the XML both to stdout and to a file. Use this program with one argument: the MP3 file on disk.

```
/* example-begin xml-mp3.c */
#include <stdlib.h>
#include <gst/gst.h>

gboolean playing;

int
main (int argc, char *argv[])
{
  GstElement *filesrc, *osssink, *queue, *queue2, *decode;
  GstElement *bin;
  GstElement *thread, *thread2;

  gst_init (&argc,&argv);

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

  /* create a new thread to hold the elements */
  thread = gst_element_factory_make ("thread", "thread");
  g_assert (thread != NULL);
  thread2 = gst_element_factory_make ("thread", "thread2");
  g_assert (thread2 != NULL);

  /* create a new bin to hold the elements */
  bin = gst_bin_new ("bin");
  g_assert (bin != 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);

  queue = gst_element_factory_make ("queue", "queue");
  queue2 = gst_element_factory_make ("queue", "queue2");

  /* 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 bin */
  gst_bin_add (GST_BIN (bin), filesrc);
```

```
  gst_bin_add (GST_BIN (bin), queue);

  gst_bin_add (GST_BIN (thread), decode);
  gst_bin_add (GST_BIN (thread), queue2);

  gst_bin_add (GST_BIN (thread2), osssink);

  gst_pad_link (gst_element_get_pad (filesrc,"src"),
                gst_element_get_pad (queue,"sink"));

  gst_pad_link (gst_element_get_pad (queue,"src"),
                gst_element_get_pad (decode,"sink"));
  gst_pad_link (gst_element_get_pad (decode,"src"),
                gst_element_get_pad (queue2,"sink"));

  gst_pad_link (gst_element_get_pad (queue2,"src"),
                gst_element_get_pad (osssink,"sink"));

  gst_bin_add (GST_BIN (bin), thread);
  gst_bin_add (GST_BIN (bin), thread2);

  /* write the bin to stdout */
  gst_xml_write_file (GST_ELEMENT (bin), stdout);

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

  exit (0);
}
/* example-end xml-mp3.c */
```

The most important line is:

```
  gst_xml_write_file (GST_ELEMENT (bin), 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 *bin;
  gboolean ret;

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

```
  xml = gst_xml_new ();

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

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

  gst_element_set_state (bin, GST_STATE_PLAYING);

  while (gst_bin_iterate(GST_BIN(bin)));

  gst_element_set_state (bin, 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 a 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");
    ...
  thread = gst_element_factory_make ("thread", "thread");
  g_signal_connect (G_OBJECT (thread), "object_saved",
         G_CALLBACK (object_saved), g_strdup ("decoder thread"));
    ...
```

When the thread is saved, the object_save method will be caled. 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>thread</gst:name>
          <gst:type>thread</gst:type>
          <gst:version>0.1.0</gst:version>
   ...
        </gst:children>
        <test:comment>
          <test:text>decoder thread</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 25. Debugging

GStreamer has an extensive set of debugging tools for plugin developers.

Command line options

Applications using the GStreamer libraries accept the following set of command line argruments to enable the debugging system.

- `--gst-debug-mask=mask` Sets the mask for the debugging output.
- `--gst-info-mask=mask` Sets the mask for the info output.
- `--gst-mask=mask` Sets the mask for the info *and* the debug output.
- `--gst-mask-help` Print out the meaning of gst-mask-* values.
- `--gst-plugin-spew` Enable printout of errors while loading GST plugins.
- `--gst-plugin-path=PATH` Add a directory to the plugin search path.
- `--help` Print the a short desciption of the options

The following table gives an overview of the mask values and their meaning. (enabled) means that the corresponding flag is set by default. This table is available to any GStreamer application by the --gst-mask-help option.

```
Mask (to be OR'ed)   info/debug         FLAGS
--------------------------------------------------------
0x00000001     (enabled)/              GST_INIT
0x00000002             /               COTHREADS
0x00000004             /               COTHREAD_SWITCH
0x00000008             /               AUTOPLUG
0x00000010             /               AUTOPLUG_ATTEMPT
0x00000020             /               PARENTAGE
0x00000040             /               STATES
0x00000080             /               PLANING
0x00000100             /               SCHEDULING
0x00000200             /               OPERATION
0x00000400             /               BUFFER
0x00000800             /               CAPS
0x00001000             /               CLOCK
0x00002000             /               ELEMENT_PADS
0x00004000             /               ELEMENT_FACTORY
0x00008000             /               PADS
0x00010000             /               PIPELINE
0x00020000             /               PLUGIN_LOADING
0x00040000             /               PLUGIN_ERRORS
0x00080000             /               PLUGIN_INFO
0x00100000             /               PROPERTIES
0x00200000             /               THREAD
0x00400000             /               TYPES
0x00800000             /               XML
0x01000000             /               NEGOTIATION
0x02000000             /               REFCOUNTING
```

Adding a custom debug handler

Chapter 26. Programs

gst-register

gst-register is used to rebuild the database of plugins. It is used after a new plugin has been added to the system. The plugin database can be found, by default, in `/etc/gstreamer/reg.xml`.

gst-launch

This is a tool that will construct pipelines based on a command-line syntax.

A simple commandline looks like:

```
gst-launch filesrc location=hello.mp3 ! mad ! osssink
```

A more complex pipeline looks like:

```
gst-launch filesrc location=redpill.vob ! mpegdemux name=demux \
 demux.audio_00! { ac3parse ! a52dec ! osssink } \
 demux.video_00! { mpeg2dec ! xvideosink }
```

You can also use the parser in you own code. `GStreamer` provides a function gst_parse_launch () that you can use to construct a pipeline. The following program lets you create an MP3 pipeline using the gst_parse_launch () function:

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

int
main (int argc, char *argv[])
{
  GstElement *pipeline;
  GstElement *filesrc;
  GError *error = NULL;

  gst_init (&argc, &argv);

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

  pipeline = gst_parse_launch ("filesrc name=my_filesrc ! mad ! osssink", &error);
  if (!pipeline) {
    g_print ("Parse error: %s\n", error->message);
    exit (1);
  }

  filesrc = gst_bin_get_by_name (GST_BIN (pipeline), "my_filesrc");
  g_object_set (G_OBJECT (filesrc), "location", argv[1], NULL);

  gst_element_set_state (pipeline, GST_STATE_PLAYING);

  while (gst_bin_iterate (GST_BIN (pipeline)));

  gst_element_set_state (pipeline, GST_STATE_NULL);

  return 0;
```

```
}
```

Note how we can retrieve the filesrc element from the constructed bin using the element name.

Grammar Reference

The **gst-launch** syntax is processed by a flex/bison parser. This section is intended to provide a full specification of the grammar; any deviations from this specification is considered a bug.

Elements

```
... mad ...
```

A bare identifier (a string beginning with a letter and containing only letters, numbers, dashes, underscores, percent signs, or colons) will create an element from a given element factory. In this example, an instance of the "mad" MP3 decoding plugin will be created.

Links

```
... !sink ...
```

An exclamation point, optionally having a qualified pad name (an the name of the pad, optionally preceded by the name of the element) on both sides, will link two pads. If the source pad is not specified, a source pad from the immediately preceding element will be automatically chosen. If the sink pad is not specified, a sink pad from the next element to be constructed will be chosen. An attempt will be made to find compatible pads. Pad names may be preceded by an element name, as in `my_element_name.sink_pad`.

Properties

```
... location="http://gstreamer.net" ...
```

The name of a property, optionally qualified with an element name, and a value, separated by an equals sign, will set a property on an element. If the element is not specified, the previous element is assumed. Strings can optionally be enclosed in quotation marks. Characters in strings may be escaped with the backtick (\). If the right-hand side is all digits, it is considered to be an integer. If it is all digits and a decimal point, it is a double. If it is "true", "false", "TRUE", or "FALSE" it is considered to be boolean. Otherwise, it is parsed as a string. The type of the property is determined later on in the parsing, and the value is converted to the target type. This conversion is not guaranteed to work, it relies on the g_value_convert routines. No error message will be displayed on an invalid conversion, due to limitations in the value convert API.

Bins, Threads, and Pipelines

```
( ... )
```

A pipeline description between parentheses is placed into a bin. The open paren may be preceded by a type name, as in `jackbin.( ... )` to make a bin of a specified type. Square brackets make pipelines, and curly braces make threads. The default toplevel bin type is a pipeline, although putting the whole description within parentheses or braces can override this default.

gst-inspect

This is a tool to query a plugin or an element about its properties.

To query the information about the element mad, you would specify:

```
gst-inspect mad
```

Below is the output of a query for the osssink element:

```
Factory Details:
  Long name: Audio Sink (OSS)
  Class: Sink/Audio
  Description: Output to a sound card via OSS
  Version: 0.3.3.1
  Author(s): Erik Walthinsen <omega@cse.ogi.edu>, Wim Taymans <wim.taymans@chello.be>
  Copyright: (C) 1999

GObject
 +----GstObject
       +----GstElement
             +----GstOssSink

Pad Templates:
  SINK template: 'sink'
    Availability: Always
    Capabilities:
      'osssink_sink':
        MIME type: 'audio/raw':
        format: String: int
        endianness: Integer: 1234
        width: List:
          Integer: 8
          Integer: 16
        depth: List:
          Integer: 8
          Integer: 16
        channels: Integer range: 1 - 2
        law: Integer: 0
        signed: List:
          Boolean: FALSE
          Boolean: TRUE
        rate: Integer range: 1000 - 48000


Element Flags:
  GST_ELEMENT_THREADSUGGESTED

Element Implementation:
  No loopfunc(), must be chain-based or not configured yet
  Has change_state() function: gst_osssink_change_state
  Has custom save_thyself() function: gst_element_save_thyself
  Has custom restore_thyself() function: gst_element_restore_thyself

Clocking Interaction:
```

```
  element requires a clock
  element provides a clock: GstOssClock

Pads:
  SINK: 'sink'
    Implementation:
      Has chainfunc(): 0x40056fc0
    Pad Template: 'sink'

Element Arguments:
  name                                    : String (Default "element")
  device                                  : String (Default "/dev/dsp")
  mute                                    : Boolean (Default false)
  format                                  : Integer (Default 16)
  channels                                : Enum "GstAudiosinkChannels" (default 1)
    (0):  Silence
    (1):  Mono
    (2):  Stereo
  frequency                               : Integer (Default 11025)
  fragment                                : Integer (Default 6)
  buffer-size                             : Integer (Default 4096)

Element Signals:
  "handoff" :  void user_function (GstOssSink* object,
        gpointer user_data);
```

To query the information about a plugin, you would do:

```
gst-inspect gstelements
```

gst-play

A sample media player.

Chapter 27. Components

FIXME: This chapter is way out of date.

`GStreamer` includes components that people can include in their programs.

GstPlay

GstPlay is a GtkWidget with a simple API to play, pause and stop a media file.

GstMediaPlay

GstMediaply is a complete player widget.

GstEditor

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

Chapter 28. Gnome integration

GStreamer is fairly easy to integrate with Gnome applications. GStreamer uses libxml 2.0, GLib 2.0 and popt, as do all other Gnome applications. There are however some basic issues you need to address in your Gnome applications.

Command line options

Gnome applications call gnome_program_init () to parse command-line options and initialize the necessary gnome modules. GStreamer applications normally call gst_init (&argc, &argv) to do the same for GStreamer.

Each of these two swallows the program options passed to the program, so we need a different way to allow both Gnome and GStreamer to parse the command-line options. This is shown in the following example.

```
/* example-begin gnome.c */
#include <gnome.h>
#include <gst/gst.h>

int
main (int argc, char **argv)
{
  struct poptOption options[] = {
          { NULL, '\0', POPT_ARG_INCLUDE_TABLE, NULL, 0, "GStreamer", NULL },
            POPT_TABLEEND
        };
  GnomeProgram *program;
  poptContext context;
  const gchar **argvn;

  options[0].arg = (void *) gst_init_get_popt_table ();
  if (! (program = gnome_program_init (PACKAGE, VERSION, LIBGNOMEUI_MODULE,
                                       argc, argv,
                                       GNOME_PARAM_POPT_TABLE, options,
                                       NULL)))
  g_error ("gnome_program_init failed");

  g_object_get (program, "popt-context", &context, NULL);
  argvn = poptGetArgs (context);
  if (!argvn) {
    g_print ("Run this example with some arguments to see how it works.\n");
    return 0;
  }

  while (*argvn) {
    g_print ("argument: %s\n", *argvn);
    ++argvn;
  }

  return 0;
}
/* example-end gnome.c */
```

If you try out this program, you will see that when called with --help, it will print out both GStreamer and Gnome help arguments. All of the arguments that didn't belong to either end up in the argvn pointer array.

FIXME: flesh this out more. How do we get the GStreamer arguments at the end ? FIXME: add a GConf bit.

Chapter 29. 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.openprojects.net: the following are a selection of amusing[1] quotes from our conversations.

14 Sep 2002

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

* *wingo* holds head

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

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

Notes

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