`GStreamer` Plugin Writer’s Guide

Richard John Boulton

Erik Walthinsen

`GStreamer` Plugin Writer's Guide

by Richard John Boulton and Erik Walthinsen

Table of Contents

Chapter 1. Do I care?

This guide explains how to write new modules for GStreamer. It is relevant to:

- Anyone who wants to add support for new input and output devices, often called sources and sinks. For example, adding the ability to write to a new video output system could be done by writing an appropriate sink plugin.
- Anyone who wants to add support for new ways of processing data in GStreamer, often called filters. For example, a new data format converter could be created.
- Anyone who wants to extend GStreamer in any way: you need to have an understanding of how the plugin system works before you can understand the constraints it places on the rest of the code. And you might be surprised at how much can be done with plugins.

This guide is not relevant to you if you only want to use the existing functionality of GStreamer, or use an application which uses GStreamer. You lot can go away. Shoo... (You might find the *GStreamer Application Development Manual* helpful though.)

Chapter 2. Preliminary reading

The reader should be familiar with the basic workings of `GStreamer`. For a gentle introduction to GStreamer, you may wish to read the *GStreamer Application Development Manual*. Since `GStreamer` adheres to the GTK+ programming model, the reader is also assumed to understand the basics of GTK+.

Chapter 3. Plugins

Extensions to `GStreamer` can be made using a plugin mechanism. This is used extensively in `GStreamer` even if only the standard package is being used: a few very basic functions reside in the core library, and all others are implemented in plugins.

Plugins are only loaded when needed: a plugin registry is used to store the details of the plugins so that it is not neccessary to load all plugins to determine which are needed. This registry needs to be updated whenever a new plugin is added to the system: see the *gstreamer-register* utility and the documentation in the *GStreamer Application Development Manual* for more details.

User extensions to `GStreamer` can be installed in the main plugin directory, and will immediately be available for use in applications. *gstreamer-register* should be run to update the repository: but the system should work correctly even if it hasn't been - it will just take longer to load the correct plugin.

User specific plugin directories and registries will be available in future versions of `GStreamer`.

Chapter 4. Elements

Elements are at the core of `GStreamer`. Without elements, `GStreamer` is just a bunch of pipe fittings with nothing to connect. A large number of elements (filters, sources and sinks) ship with `GStreamer`, but extra elements can also be written.

An element may be constructed in several different ways, but all must conform to the same basic rules. A simple filter may be built with the FilterFactory, where the only code that need be written is the actual filter code. A more complex filter, or a source or sink, will need to be written out fully for complete access to the features and performance possible with `GStreamer`.

The implementation of a new element will be contained in a plugin: a single plugin may contain the implementation of several elements, or just a single one.

Chapter 5. Buffers

Buffers are structures used to pass data between elements. All streams of data are chopped up into chunks which are stored in buffers. Buffers can be of any size, and also contain metadata indicating the type of data contained in them. Buffers can be allocated by various different schemes, and may either be passed on by elements or unreferenced (and the memory used by the buffer freed).

Chapter 6. Typing and Properties

A type system is used to ensure that the data passed between elements is in a recognised format, and that the various parameters required to fully specify that format match up correctly. Each connection that is made between elements has a specified type. This is related, but different, to the metadata in buffers which describes the type of data in that particular buffer. See later in this document for details of the available types.

Chapter 7. Metadata

Chapter 8. Scheduling

Chapter 9. Chain vs Loop Elements

Chapter 10. Autopluggers

`GStreamer` has an autoplugging mechanism, which enables application writers to simply specify start and end elements for a path, and the system will then create a path which links these elements, in accordance with the type information provided by the elements.

It is possible to devise many different schemes for generating such pathways, perhaps to optimise based on special criteria, or with some specific constraints. It is thus possible to define new autoplugging systems, using the plugin system.

Chapter 11. The basic types

This is a list of the basic types used for buffers. For each type, we give the name ("mime type") of the type, the list of properties which are associated with the type, the meaning of each property, and the purpose of the type.

- *audio/raw* - Unstructured and uncompressed raw audio data.

  rate - The sample rate of the data, in samples per second.

  channels - The number of channels of audio data.

  format - This describes the format in which the audio data is passed. This is a string for which there are currently two valid values: "int" for integer data and "float" for floating point data.

  law - Valid only if format=int. The law used to describe the data. This is an integer for which there are three valid values: 0 for linear, 1 for mu law, 2 for A law.

  endianness - Valid only if format=int. The order of bytes in a sample. This is a boolean: 0 means little-endian (ie, bytes are least significant first), 1 means big-endian (ie, most significant byte first).

  signed - Valid only if format=int. Whether the samples are signed or not. This is a boolean: 0 means unsigned, 1 means signed.

  width - Valid only if format=int. The number of bits per sample. This is extremely likely to be a multiple of 8, but as ever this is up to each element supporting this format to specify.

  depth - Valid only if format=int. The number of bits used per sample. This must be less than or equal to the width: if less than the width, the low bits are assumed to be the ones used. For example, width=32, depth=24 means that each sample is stored in a 32 bit word, but only the low 24 bits are actually used.

  layout - Valid only if format=float. A string representing the way in which the floating point data is represented. For now, the only valid value is gfloat, meaning that the data is passed as a series of gfloat values.

  intercept - Valid only if format=float. A floating point value representing the value that the signal "centres" on.

  slope - Valid only if format=float. A floating point value representing how far the signal deviates from the intercept. So a slope of 1.0 and an intercept of 0.0 would mean an audio signal with minimum and maximum values of -1.0 and 1.0. A slope of 0.5 and intercept of 0.5 would represent values in the range 0.0 to 1.0.

  For example: 16 bit integer, unsigned, linear, monophonic, big-endian, 44100KHz audio would be represented by "format=int,law=0,endianness=1,signed=0,width=16,depth=16,rate=44100,channels=1" and floating point, using gfloat's, in the range -1.0 to 1.0, 8000KHz stereo audio would be represented by "format=float,layout=gfloat,intercept=0.0,slope=1.0,rate=8000,channels=2"

- *audio/mp3* - Audio data compressed using the mp3 encoding scheme.

  framed - This is a boolean. If true (1), each buffer contains exactly one frame. If false (0), frames and buffers do not (necessarily) match up. If the data is not framed, the values of some of the properties will not be available, but others will be assumed to be constant throughout the file, or may be found in other ways.

layer - The compression scheme layer used to compress the data. This is an integer, and can currently have the value 1, 2 or 3.

bitrate - The bitrate, in kilobits per second. For VBR (variable bitrate) mp3 data, this is the average bitrate.

channels - The number of channels of audio data present. This could theoretically be any integer greater than 0, but in practice will be either 1 or 2.

joint-stereo - Boolean. If true, channels must not be zero. If true, this implies that stereo data is stored as a combined signal and the difference between the signals, rather than as two entirely separate signals.

There are many other properties relevant for *audio/mp3* data: these may be added to this specification at a later date.

- *audio/x-ogg* - Audio data compressed using the Ogg Vorbis encoding scheme. There are currently no parameters defined for this type. FIXME.

- *video/raw* - Raw video data.

  fourcc - A FOURCC code identifying the format in which this data is stored. FOURCC (Four Character Code) is a simple system to allow unambiguous identification of a video datastream format. See http://www.webartz.com/fourcc/

  width - The number of pixels wide that each video frame is.

  height - The number of pixels high that each video frame is.

- *video/mpeg* - Video data compressed using an mpeg encoding scheme.

  mpegversion

  systemstream

- *video/avi* - Video data compressed using the AVI encoding scheme. There are currently no parameters defined for this type. FIXME.

Chapter 12. Building a simple format for testing

Chapter 13. A simple MIME type

Chapter 14. Type properties

Chapter 15. Typefind functions and autoplugging

Chapter 16. Constructing the boilerplate

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

Doing it the hard way with GstObject

The standard way of defining the boilerplate is simply to write some code, and fill in some structures. The easiest way to do this is to copy an example and modify according to your needs.

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

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

struct _GstExample {
  GstElement element;

  GstPad *sinkpad,*srcpad;

  gint8 active;
};

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

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

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

Doing it the easy way with FilterFactory

A plan for the future is to create a FilterFactory, to make the process of making a new filter a simple process of specifying a few details, and writing a small amount of code to perform the actual data processing.

Unfortunately, this hasn't yet been implemented. It is also likely that when it is, it will not be possible to cover all the possibilities available by writing the boilerplate yourself, so some plugins will always need to be manually registered.

As a rough outline of what is planned: the FilterFactory will take a list of appropriate function pointers, and data structures to define a filter. With a reasonable measure of preprocessor magic, the plugin writer will then simply need to provide definitions of the functions and data structures desired, and a name for the filter, and then call a macro from within plugin_init() which will register the new filter. All the fluff that goes into the definition of a filter will thus be hidden from view.

Ideally, we will come up with a way for various FilterFactory-provided functions to be overridden, to the point where you can construct almost the most complex stuff with it, it just saves typing.

Of course, the filter factory can be used to create sources and sinks too: simply create a filter with only source or sink pads.

You may be thinking that this should really be called an ElementFactory. Well, we agree, but there is already something else justifiably ealled an ElementFactory (this is the thing which actually makes instances of elements). There is also already something called a PluginFactory. We just have too many factories and not enough words. And since this isn't yet written, it doesn't get priority for claiming a name.

Chapter 17. Defining an element

A new element is defined by creating an element factory. This is a structure containing all the information needed to create an instance of the element. Creating a factory requires two things: a type for the element to be created (this was defined in the boilerplate above: FIXME - reorganise), and a GstElementDetails structure, which contains some general information about the element to be created.

GstElementDetails

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

- A long, english, name for the element.
- The type of the element, as a heirarchy. The heirarchy is defined by specifying the top level category, followed by a "/", followed by the next level category, etc. The type should be defined according to the guidelines elsewhere in this document. (FIXME: write the guidelines, and give a better reference to them)
- A brief description of the purpose of the element.
- The version number of the element. For elements in the main GStreamer source code, this will often simply be VERSION, which is a macro defined to be the version number of the current GStreamer version. The only requirement, however, is that the version number should increase monotonically.

  Version numbers should be stored in major.minor.patch form: ie, 3 (decimal) numbers, separated by ".".

- The name of the author of the element, optionally followed by a contact email address in angle brackets.
- The copyright details for the element.

For example:

```
static GstElementDetails example_details = {
    "An example plugin",
    "Example/FirstExample",
    "Shows the basic structure of a plugin",
    VERSION,
    "your name <your.name@your.isp>",
    "(C) 2001",
};
```

Constructor functions

Each element has two functions which are used for construction of an element. These are the _class_init() function, which is used to initialise the class (specifying what signals and arguments the class has and setting up global state), and the _init() function, which is used to initialise a specific instance of the class.

Specifying the pads

Attaching functions

The chain function

Adding arguments

Define arguments in enum.

Signals

Define signals in enum.

Chapter 18. Defining a type

A new type is defined by creating an type factory. This is a structure containing all the information needed to create an instance of the type.

Chapter 19. The plugin_init function

Once we have written code defining all the parts of the plugin, we need to write the plugin_init() function. This is a special function, which is called as soon as the plugin is loaded, and must return a pointer to a newly allocated GstPlugin structure. This structure contains the details of all the facilities provided by the plugin, and is the mechanism by which the definitions are made available to the rest of the `GStreamer` system. Helper functions are provided to help fill the structure: for future compatability it is recommended that these functions are used, as documented below, rather than attempting to access the structure directly.

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

Registering new types

```
void gst_plugin_add_type(GstPlugin *plugin,
 GstTypeFactory *factory);
```

Registering new element factories

```
void gst_plugin_add_factory(GstPlugin *plugin,
    GstElementFactory *factory);
```

Multiple element factories can be provided by a single plugin: all it needs to do is call gst_plugin_add_factory() for each element factory it wishes to provide.

Registering new autopluggers

```
void gst_plugin_add_autoplugger(GstPlugin *plugin,
GstAutoplugFactory *factory);
```

Chapter 20. Initialization

Chapter 21. Instantiating the plugins

(NOTE: we really should have a debugging Sink)

Chapter 22. Connecting the plugins

Chapter 23. Running the pipeline

Chapter 24. How scheduling works

aka pushing and pulling

Chapter 25. How a loopfunc works

aka pulling and pushing

Chapter 26. Adding a second output

Identity is now a tee

Chapter 27. Modifying the test application

Anatomy of a Buffer Refcounts and mutability Metadata How Properties work efficiently Metadata mutability (FIXME: this is an unsolved problem) Writing a source Pull vs loop based Region pulling (NOTE: somewhere explain how filters use this) Writing a sink Gee, that was easy What are states? Mangaging filter state Things to check when writing a filter Things to check when writing a source or sink