`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 in a standard set of 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 when 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 will work correctly even if it hasn't been - it will just load the correct plugin faster.

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

Chapter 6. Scheduling

Chapter 7. Chain vs Loop Elements

Chapter 8. Typing and Properties

Chapter 9. Metadata

Chapter 10. 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 11. An identity filter

Building an object with pads

Attaching functions

The chain function

Chapter 12. The plugin_init function

Registering the types

Registering the filter

Having multiple filters in a single plugin

Chapter 13. Initialization

Chapter 14. Instantiating the plugins

(NOTE: we really should have a debugging Sink)

Chapter 15. Connecting the plugins

Chapter 16. Running the pipeline

Chapter 17. How scheduling works

aka pushing and pulling

Chapter 18. How a loopfunc works

aka pulling and pushing

Chapter 19. Adding a second output

Identity is now a tee

Chapter 20. Modifying the test application

Chapter 21. Building a simple format for testing

Chapter 22. A simple MIME type

Chapter 23. Type properties

Chapter 24. Typefind functions and autoplugging

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