`GStreamer` Plugin Writer’s Guide

Richard John Boulton

Erik Walthinsen

Steve Baker

Leif Johnson

`GStreamer` Plugin Writer’s Guide

by Richard John Boulton, Erik Walthinsen, Steve Baker, and Leif Johnson

Table of Contents

Chapter 1. Preface

Who Should Read This Guide?

This guide explains how to write new modules for `GStreamer`. The guide is relevant to several groups of people:

- Anyone who wants to add support for new ways of processing data in `GStreamer`. For example, a person in this group might want to create a new data format converter, a new visualization tool, or a new decoder or encoder.
- Anyone who wants to add support for new input and output devices. For example, people in this group might want to add the ability to write to a new video output system or read data from a digital camera or special microphone.
- Anyone who wants to extend `GStreamer` in any way. You need to have an understanding of how the plugin system works before you can understand the constraints that the plugin system places on the rest of the code. Also, you might be surprised after reading this at how much can be done with plugins.

This guide is not relevant to you if you only want to use the existing functionality of `GStreamer`, or if you just want to use an application that uses `GStreamer`. If you are only interested in using existing plugins to write a new application - and there are quite a lot of plugins already - you might want to check the *GStreamer Application Development Manual*. If you are just trying to get help with a `GStreamer` application, then you should check with the user manual for that particular application.

Preliminary Reading

This guide assumes that you are somewhat familiar with the basic workings of `GStreamer`. For a gentle introduction to programming concepts in `GStreamer`, you may wish to read the *GStreamer Application Development Manual* first. Also check out the documentation available on the `GStreamer` web site[1], particularly the documents available in the `GStreamer` wiki[2].

Since `GStreamer` adheres to the GObject programming model, this guide also assumes that you understand the basics of GObject[3] programming. There are several good introductions to the GObject library, including the *GTK+ Tutorial*[4].

Structure of This Guide

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

- Building a Filter - Introduction to the structure of a plugin, using an example audio filter for illustration.

  This part covers all the basic steps you generally need to perform to build a plugin. The discussion begins by giving examples of generating the basic structures with Constructing the Boilerplate. Then you will learn how to write the code to get a basic filter plugin working: These steps include chapters on Chapter 18, Chapter 4, Chapter 5, and (WRITEME: building state).

  After you have finished the first steps, you will be able to create a working plugin, but your new plugin might not have all the functionality you need. To provide some standard functionality, you will learn how to add more features to a new plugin. These features are described in the chapters on (WRITEME) and Chapter 9.

Finally, you will see in (WRITEME) how to write a short test application to try out your new plugin.

- Advanced Filter Concepts - Information on advanced features of `GStreamer` plugin development.

  After learning about the basic steps, you should be able to create a functional audio or video filter plugin with some nice features. However, `GStreamer` offers more for plugin writers. This part of the guide includes chapters on more advanced topics, such as Chapter 19, . Since these features are more advanced, the chapters can basically be read in any order, as you find that your plugins require these features.

- Other Element Types - Explanation of writing other plugin types.

  Because the first two parts of the guide use an audio filter as an example, the concepts introduced apply to filter plugins. But many of the concepts apply equally to other plugin types, including sources, sinks, and autopluggers. This part of the guide presents the issues that arise when working on these more specialized plugin types. The part includes chapters on Writing a Source, Writing a Sink, and Writing an Autoplugger.

- Appendices - Further information for plugin developers.

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

The remainder of this introductory part of the guide presents a short overview of the basic concepts involved in `GStreamer` plugin development. Topics covered include Elements and Plugins, Pads, Buffers, Types and Properties, and Events. If you are already familiar with this information, you can use this short overview to refresh your memory, or you can skip to Building a Filter.

As you can see, there a lot to learn, so let's get started!

- Creating compound and complex elements by extending from a GstBin. This will allow you to create plugins that have other plugins embedded in them.
- Adding new mime-types to the registry along with typedetect functions. This will allow your plugin to operate on a completely new media type.

Notes

1. http://gstreamer.net/docs/
2. http://gstreamer.net/wiki/
3. http://developer.gnome.org/doc/API/2.0/gobject/index.html
4. http://www.gtk.org/tutorial/

Chapter 2. Basic Concepts

This chapter of the guide introduces the basic concepts of `GStreamer`. Understanding these concepts will help you grok the issues involved in extending `GStreamer`. Many of these concepts are explained in greater detail in the *GStreamer Application Development Manual*; the basic concepts presented here serve mainly to refresh your memory.

Elements and Plugins

Elements are at the core of `GStreamer`. In the context of plugin development, an *element* is an object derived from the `GstElement` class. Elements provide some sort of functionality when linked with other elements: For example, a source element provides data to a stream, and a filter element acts on the data in a stream. Without elements, `GStreamer` is just a bunch of conceptual pipe fittings with nothing to link. A large number of elements ship with `GStreamer`, but extra elements can also be written.

Just writing a new element is not entirely enough, however: You will need to encapsulate your element in a *plugin* to enable `GStreamer` to use it. A plugin is essentially a loadable block of code, usually called a shared object file or a dynamically linked library. A single plugin may contain the implementation of several elements, or just a single one. For simplicity, this guide concentrates primarily on plugins containing one element.

A *filter* is an important type of element that processes a stream of data. Producers and consumers of data are called *source* and *sink* elements, respectively. Elements that link other elements together are called *autoplugger* elements, and a *bin* element contains other elements. Bins are often responsible for scheduling the elements that they contain so that data flows smoothly.

The plugin mechanism is used everywhere 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. A plugin registry is used to store the details of the plugins in an XML file. This way, a program using `GStreamer` does not have to load all plugins to determine which are needed. Plugins are only loaded when their provided elements are requested.

See the *GStreamer Library Reference* for the current implementation details of `GstElement`[1] and `GstPlugin`[2].

Pads

Pads are used to negotiate links and data flow between elements in `GStreamer`. A pad can be viewed as a "place" or "port" on an element where links may be made with other elements. Pads have specific data handling capabilities: A pad only knows how to give or receive certain types of data. Links are only allowed when the capabilities of two pads are compatible.

An analogy may be helpful here. A pad is similar to a plug or jack on a physical device. Consider, for example, a home theater system consisting of an amplifier, a DVD player, and a (silent) video projector. Linking the DVD player to the amplifier is allowed because both devices have audio jacks, and linking the projector to the DVD player is allowed because both devices have compatible video jacks. Links between the projector and the amplifier may not be made because the projector and amplifier have different types of jacks. Pads in `GStreamer` serve the same purpose as the jacks in the home theater system.

For the moment, all data in `GStreamer` flows one way through a link between elements. Data flows out of one element through one or more *source pads*, and elements accept incoming data through one or more *sink pads*. Source and sink elements have only source and sink pads, respectively.

See the *GStreamer Library Reference* for the current implementation details of a `GstPad`[3].

Buffers

All streams of data in `GStreamer` are chopped up into chunks that are passed from a source pad on one element to a sink pad on another element. *Buffers* are structures used to hold these chunks of data. Buffers can be of any size, theoretically, and they may contain any sort of data that the two linked pads know how to handle. Normally, a buffer contains a chunk of some sort of audio or video data that flows from one element to another.

Buffers also contain metadata describing the buffer's contents. Some of the important types of metadata are:

- A pointer to the buffer's data.
- An integer indicating the size of the buffer's data.
- A `GstData` object describing the type of the buffer's data.
- A reference count indicating the number of elements currently holding a reference to the buffer. When the buffer reference count falls to zero, the buffer will be unlinked, and its memory will be freed in some sense (see below for more details).

See the *GStreamer Library Reference* for the current implementation details of a `GstBuffer`[4].

Buffer Allocation and Buffer Pools

Buffers can be allocated using various schemes, and they may either be passed on by an element or unreferenced, thus freeing the memory used by the buffer. Buffer allocation and unlinking are important concepts when dealing with real time media processing, since memory allocation is relatively slow on most systems.

To improve the latency in a media pipeline, many `GStreamer` elements use a *buffer pool* to handle buffer allocation and unlinking. A buffer pool is a relatively large chunk of memory that is the `GStreamer` process requests early on from the operating system. Later, when elements request memory for a new buffer, the buffer pool can serve the request quickly by giving out a piece of the allocated memory. This saves a call to the operating system and lowers latency. [If it seems at this point like `GStreamer` is acting like an operating system (doing memory management, etc.), don't worry: `GStreamer`OS isn't due out for quite a few years!]

Normally in a media pipeline, most filter elements in `GStreamer` deal with a buffer in place, meaning that they do not create or destroy buffers. Sometimes, however, elements might need to alter the reference count of a buffer, either by copying or destroying the buffer, or by creating a new buffer. These topics are generally reserved for non-filter elements, so they will be addressed at that point.

Types and Properties

`GStreamer` uses a type system to ensure that the data passed between elements is in a recognized format. The type system is also important for ensuring that the parameters required to fully specify a format match up correctly when linking pads between elements. Each link that is made between elements has a specified type.

The Basic Types

`GStreamer` already supports many basic media types. Following is a table of the basic types used for buffers in `GStreamer`. The table contains the name ("mime type") and a description of the type, the properties associated with the type, and the meaning of each property.

Table 2-1. Table of Basic Types

Mime Type	Description	Property	Property Type	Property Values	Property Description
audio/raw	Unstructured and uncompressed raw audio data.	rate	integer	greater than 0	The sample rate of the data, in samples per second.
		channels	integer	greater than 0	The number of channels of audio data.
		format	string	“int” or “float”	The format in which the audio data is passed.
		law	integer	0, 1, or 2	(Valid only if the data is in integer format.) The law used to describe the data. The value 0 indicates “linear”, 1 indicates “mu law”, and 2 indicates “A law”.

Mime Type	Description	Property	Property Type	Property Values	Property Description
		endianness	boolean	0 or 1	(Valid only if the data is in integer format.) The order of bytes in a sample. The value 0 means "little-endian" (bytes are least significant first). The value 1 means "big-endian" (most significant byte first).
		signed	boolean	0 or 1	(Valid only if the data is in integer format.) Whether the samples are signed or not.
		width	integer	greater than 0	(Valid only if the data is in integer format.) The number of bits per sample.

Mime Type	Description	Property	Property Type	Property Values	Property Description
		depth	integer	greater than 0	(Valid only if the data is in integer format.) The number of bits used per sample. This must be less than or equal to the width: If the depth is less than the width, the low bits are assumed to be the ones used. For example, a width of 32 and a depth of 24 means that each sample is stored in a 32 bit word, but only the low 24 bits are actually used.
		layout	string	“gfloat”	(Valid only if the data is in float format.) A string representing the way in which the floating point data is represented.
		intercept	float	any, normally 0	(Valid only if the data is in float format.) A floating point value representing the value that the signal “centers” on.

Mime Type	Description	Property	Property Type	Property Values	Property Description
		slope	float	any, normally 1.0	(Valid only if the data is in float format.) A floating point value representing how far the signal deviates from the intercept. 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.
audio/mp3	Audio data compressed using the mp3 encoding scheme.	framed	boolean	0 or 1	A true value indicates that each buffer contains exactly one frame. A false value indicates that frames and buffers do not necessarily match up.
		layer	integer	1, 2, or 3	The compression scheme layer used to compress the data.

Mime Type	Description	Property	Property Type	Property Values	Property Description
		bitrate	integer	greater than 0	The bitrate, in kilobits per second. For VBR (variable bitrate) mp3 data, this is the average bitrate.
		channels	integer	greater than 0	The number of channels of audio data present.
		joint-stereo	boolean	0 or 1	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. If true, the “channels” attribute must not be zero.
audio/x-ogg	Audio data compressed using the Ogg Vorbis encoding scheme.				FIXME: There are currently no parameters defined for this type.

Mime Type	Description	Property	Property Type	Property Values	Property Description
video/raw	Raw video data.	fourcc	FOURCC code		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/fou
		width	integer	greater than 0	The number of pixels wide that each video frame is.
		height	integer	greater than 0	The number of pixels high that each video frame is.
video/mpeg	Video data compressed using an MPEG encoding scheme.				FIXME: There are currently no parameters defined for this type.
video/avi	Video data compressed using the AVI encoding scheme.				FIXME: There are currently no parameters defined for this type.

Events

Sometimes elements in a media processing pipeline need to know that something has happened. An *event* is a special type of data in `GStreamer` designed to serve this purpose. Events describe some sort of activity that has happened somewhere in an element's pipeline, for example, the end of the media stream or a clock discontinuity. Just like any other data type, an event comes to an element on a sink pad and is contained in a normal buffer. Unlike normal stream buffers, though, an event buffer contains only an event, not any media stream data.

See the *GStreamer Library Reference* for the current implementation details of a `GstEvent`[5].

Notes

1. gstreamer/gstelement.html
2. http://gstreamer.net/docs/current/gstreamer/gstreamer-GstPlugin.html
3. http://gstreamer.net/docs/current/gstreamer/gstreamer-GstPad.html
4. http://gstreamer.net/docs/current/gstreamer/gstreamer-GstBuffer.html
5. http://gstreamer.net/docs/current/gstreamer/gstreamer-GstEvent.html

Chapter 3. Constructing the Boilerplate

In this chapter you will learn how to construct the bare minimum code for a new plugin. Starting from ground zero, you will see how to get the `GStreamer` template source. Then you will learn how to use a few basic tools to copy and modify a template plugin to create a new plugin. If you follow the examples here, then by the end of this chapter you will have a functional audio filter plugin that you can compile and use in `GStreamer` applications.

Getting the Gstreamer Plugin Templates

There are currently two ways to develop a new plugin for `GStreamer`: You can write the entire plugin by hand, or you can copy an existing plugin template and write the plugin code you need. The second method is by far the simpler of the two, so the first method will not even be described here. (Errm, that is, "it is left as an exercise to the reader.")

The first step is to check out a copy of the `gst-template` CVS module to get an important tool and the source code template for a basic `GStreamer` plugin. To check out the `gst-template` module, make sure you are connected to the internet, and type the following commands at a command console:

```
shell $ cd .
shell $ cvs -d:pserver:anonymous@cvs.gstreamer.sourceforge.net:/cvsroot/gstreamer log
Logging in to :pserver:anonymous@cvs.gstreamer.sourceforge.net:2401/cvsroot/gstreame
CVS password:
shell $ cvs -z3 -d:pserver:anonymous@cvs.gstreamer.sourceforge.net:/cvsroot/gstreamer
U gst-template/README
U gst-template/gst-app/AUTHORS
U gst-template/gst-app/ChangeLog
U gst-template/gst-app/Makefile.am
U gst-template/gst-app/NEWS
U gst-template/gst-app/README
U gst-template/gst-app/autogen.sh
U gst-template/gst-app/configure.ac
U gst-template/gst-app/src/Makefile.am
...
```

After the first command, you will have to press **ENTER** to log in to the CVS server. (You might have to log in twice.) The second command will check out a series of files and directories into `./gst-template`. The template you will be using is in `./gst-template/gst-plugin/` directory. You should look over the files in that directory to get a general idea of the structure of a source tree for a plugin.

Using the Project Stamp

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

The standard way of defining the boilerplate is simply to write some code, and fill in some structures. As mentioned in the previous section, the easiest way to do this is to copy a template and add functionality according to your needs. To help you do so, there are some tools in the `./gst-template/tools/` directory. One tool, `gst-quick-stamp`, is a quick command line tool. The other, `gst-project-stamp`, is a full GNOME druid application that takes you through the steps of creating a new project (either a plugin or an application).

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

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

Note that capitalization is important for the name of the plugin. Under some operating systems, capitalization is also important when specifying directory names. For example, the following commands create the ExampleFilter plugin based on the plugin template and put the output files in a new directory called `~/src/examplefilter/`:

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

Examining the Basic Code

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

Example 3-1. Example Plugin Header File

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

struct _GstExample {
  GstElement element;

  GstPad *sinkpad,*srcpad;

  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);
```

Creating a Filter With FilterFactory (Future)

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

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

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

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 periods (.).
- 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.

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 required 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.

Chapter 4. Specifying the pads

Chapter 5. The chain function

Chapter 6. What are states?

Chapter 7. Mangaging filter state

Chapter 8. Adding Arguments

Define arguments in enum.

Chapter 9. Signals

Define signals in enum.

Chapter 10. Initialization

Chapter 11. Instantiating the plugins

(NOTE: we really should have a debugging Sink)

Chapter 12. Linking the plugins

Chapter 13. Running the pipeline

Chapter 14. How scheduling works

aka pushing and pulling

Chapter 15. How a loopfunc works

aka pulling and pushing

Chapter 16. Adding a second output

Identity is now a tee

Chapter 17. Modifying the test application

Chapter 18. Types and Properties

There is a very large set of possible types that may be used to pass data between elements. Indeed, each new element that is defined may use a new data format (though unless at least one other element recognises that format, it will be most likely be useless since nothing will be able to link with it).

In order for types to be useful, and for systems like autopluggers to work, it is neccessary that all elements agree on the type definitions, and which properties are required for each type. The `GStreamer` framework itself simply provides the ability to define types and parameters, but does not fix the meaning of types and parameters, and does not enforce standards on the creation of new types. This is a matter for a policy to decide, not technical systems to enforce.

For now, the policy is simple:

- Do not create a new type if you could use one which already exists.
- If creating a new type, discuss it first with the other `GStreamer` developers, on at least one of: IRC, mailing lists, the `GStreamer` wiki.
- Try to ensure that the name for a new format is as unlikely to conflict with anything else created already, and is not a more generalised name than it should be. For example: "audio/compressed" would be too generalised a name to represent audio data compressed with an mp3 codec. Instead "audio/mp3" might be an appropriate name, or "audio/compressed" could exist and have a property indicating the type of compression used.
- Ensure that, when you do create a new type, you specify it clearly, and get it added to the list of known types so that other developers can use the type correctly when writing their elements.

Building a Simple Format for Testing

A Simple Mime Type

Type Properties

Typefind Functions and Autoplugging

Chapter 19. Request pads

aka pushing and pulling

Chapter 20. Supporting Dynamic Parameters

Sometimes object properties are not powerful enough to control the parameters that affect the behaviour of your element. When this is the case you can expose these parameters as Dynamic Parameters which can be manipulated by any Dynamic Parameters aware application.

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

Comparing Dynamic Parameters with GObject Properties

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

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

Chapter 21. Getting Started

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

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

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

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

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

  ...
}
```

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

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

  GstDParamManager *dpman;

  ...
};
```

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

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

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

  ...
}
```

Chapter 22. Defining Parameter Specificiations

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

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

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

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

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

The following function will be used as an example.

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

The common parameters to these functions are:

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

Direct Method

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

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

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

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

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

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

Callback Method

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

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

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

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

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

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

The callback function needs to conform to this signiture

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

In our example the callback function looks like this

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

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

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

Array Method

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

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

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

Chapter 23. The Data Processing Loop

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

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

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

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

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

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

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

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

The macro `GST_DPMAN_PREPROCESS` will do the following:

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

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

DParam Manager Modes

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

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

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

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

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

DParam Manager Modes

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

Chapter 24. Writing a Source

FIXME: write.

Chapter 25. Writing a Sink

FIXME: write.

Chapter 26. Writing an Autoplugger

FIXME: write.

Chapter 27. Things to check when writing a filter

Chapter 28. Things to check when writing a source or sink