



# Creating a Plugin System Using GTypeModule

Michael Natterer  
<[mitch@imendio.com](mailto:mitch@imendio.com)>



# Creating a Plugin System Using GTypeModule

- This tutorial will explain how to build a plugin system using the GTypeModule interface.
- Code samples will lead the way through this process and typical pitfalls and misconceptions will be addressed.
- GModule will be used as the storage backend for plugin code.
- The sample plugin built during the session will give a demonstration of the functionality provided by GTypeModule.



# Overview

- What is GModule?
- What is GTypeModule?
- How are they related?
- GModule basics
- GTypeModule basics
- An example module
- Managing your application's modules
- Common problems and mistakes
- GTypeModule Pros and Cons
- Questions



# What is GModule?

- Abstraction layer around native module loading
- Just like linking against a shared library, only even more dynamic
- API matches `dlopen()/dlsym()/dlclose()`



# What is GTypeModule?

- Enables implementing GTypes in dynamically managed code
- Automatically loads/unloads the code belonging to the GType whenever its needed
  - Normally, loads the implementation on first instance creation...
  - ...and unloads upon finalizing the last instance



## How are they Related? (I)

- They are not related at all
- GTypeModule enables dynamically implemented GTypes
- GModule loads some code from disk
- That's not the same



## How are they Related? (II)

- GTypeModule cannot load the dynamic code itself
- GModule is **one** way to do that
- A GTypeModule implementation could also
  - load the code from a database
  - dynamically choose between several compiled-in implementations
  - It could do just anything we don't care about here



# GModule Basics (I)

- Opening and closing the module

```
GModule *module;  
  
module = g_module_open ("/path/to/libfoo.so", 0);  
  
if (!module)  
{  
    g_printerr ("%s\n", g_module_error ());  
    return;  
}  
  
/* do something with module */  
  
g_module_close (module);
```



## GModule Basics (II)

- Looking up symbols from the module

```
FooBarFunc symbol1;
FooBazFunc symbol2;

if (! g_module_symbol (module,
                      "foo_do_bar",
                      (gpointer *) &symbol1) ||
    ! g_module_symbol (module,
                      "foo_do_baz",
                      (gpointer *) &symbol2))
{
    g_printerr ("%s\n", g_module_error ());
    return;
}
```



# GModule Basics (III)

- The symbols in the module itself

```
G_MODULE_EXPORT void  
foo_do_bar (void)  
{  
    /* do bar things */  
}
```

```
G_MODULE_EXPORT void  
foo_do_baz (void)  
{  
    /* do baz things */  
}
```



# GTypeModule Basics (I)

- We need a GTypeModule subclass and need to keep an instance for each loaded module

```
struct _FooModule
{
    GTypeModule parent_instance;

    gchar      *filename;
    GModule    *library;

    /* module symbols */
    void (* load)    (FooModule *module);
    void (* unload) (FooModule *module);
};

FooModule *foo_module_new (const gchar *filename);
```



# GTypeModule Basics (II)

- We need to implement load() and unload() of GTypeModuleClass

```
struct _GTypeModuleClass
{
    GObjectClass parent_class;

    gboolean (* load)    (GTypeModule *module);
    void      (* unload) (GTypeModule *module);
};
```



# GTypeModule Basics (III)

```
static gboolean
module_load (GTypeModule *module)
{
    FooModule *mod = FOO_MODULE (module);

    mod->library = g_module_open (mod->filename, 0);

    if (! g_module_symbol (mod->library,
                          "foo_module_load",
                          (gpointer *) &mod->load))
    {
        g_module_close (mod->library);
        return FALSE; /* failure */
    }

    mod->load (mod);

    return TRUE; /* success */
}
```



# GTypeModule Basics (IV)

```
static void
module_unload (GTypeModule *module)
{
    FooModule *mod = FOO_MODULE (module);

    mod->unload (mod);

    g_module_close (mod->library);
    mod->library = NULL;

    mod->load      = NULL;
    mod->unload    = NULL;
}
```



# An Example Module (I)

- The API to implement (the parent class)

```
struct _FooFilter
{
    GObject parent_instance;
};

struct _FooFilterClass
{
    GObjectClass parent_class;

    gchar * name;

    void (* filter) (FooFilter *filter,
                    GtkTextBuffer *buffer,
                    GtkTextIter *start,
                    GtkTextIter *end);
};
```



## An Example Module (II)

- The filter implementation (the subclass in the module)

```
struct _FooUppercaseFilter
{
    FooFilter parent_instance;
};
```

```
struct _FooUppercaseFilterClass
{
    FooFilterClass parent_class;
};
```



## An Example Module (III)

- `class_init()`

```
static void
class_init (FooUppercaseFilterClass *klass)
{
    FooFilterClass *filter_class = F00_FILTER_CLASS (klass);

    foo_uppercase_filter_parent_class =
        g_type_class_peek_parent (klass);

    filter_class->name    = "Uppercase";
    filter_class->filter = uppercase_filter_filter;
}
```



# An Example Module (IV)

- The FooFilter::filter() implementation

```
static void
uppercase_filter_filter (FooFilter      *filter,
                        GtkTextBuffer *buffer,
                        GtkTextIter    *start,
                        GtkTextIter    *end)
{
    /* convert all characters from start to end
     * in buffer to uppercase
     */
}
```



# An Example Module (V)

- The type macros

```
#define F00_TYPE_UPPERCASE_FILTER \  
    (foo_uppercase_filter_type)  
#define F00_UPPERCASE_FILTER(obj) \  
    (G_TYPE_CHECK_INSTANCE_CAST ((obj), \  
    F00_TYPE_UPPERCASE_FILTER, \  
    FooUppercaseFilter))  
#define F00_UPPERCASE_FILTER_CLASS(k) \  
    (G_TYPE_CHECK_CLASS_CAST((k), \  
    F00_TYPE_UPPERCASE_FILTER, \  
    FooUppercaseFilterClass))  
#define F00_IS_UPPERCASE_FILTER(obj) \  
    (G_TYPE_CHECK_INSTANCE_TYPE ((obj), \  
    F00_TYPE_UPPERCASE_FILTER))
```



# An Example Module (VI)

- The `get_type()` function

```
static GType foo_uppercase_filter_type = 0;

static GType
foo_uppercase_filter_get_type (GTypeModule *module)
{
    if (!foo_uppercase_filter_type)
    {
        static const GTypeInfo filter_info = { ... };

        foo_uppercase_filter_type =
            g_type_module_register_type (module,
                                         FOO_TYPE_FILTER,
                                         "FooUppercaseFilter",
                                         &filter_info, 0);
    }

    return foo_uppercase_filter_type;
}
```



# An Example Module (VII)

- Getting the implemented type registered

```
G_MODULE_EXPORT void  
foo_module_load (FooModule *module)  
{  
    foo_uppercase_filter_get_type (G_TYPE_MODULE (module));  
}
```

```
G_MODULE_EXPORT void  
foo_module_unload (FooModule *module)  
{  
}
```



# Managing Your Application's Modules (I)

- Having a module registry

```
struct _FooModuleManager
{
    GObject    parent_instance;

    gchar      *module_path;
    GList      *modules;
};

struct _FooModuleManagerClass
{
    GObjectClass parent_class;
};
```



# Managing Your Application's Modules (II)

- Reading all modules from a folder

```
static void
query_modules (FooModuleManager *manager)
{
    dir = g_dir_open (manager->module_path, 0, NULL);

    while ((name = g_dir_read_name (dir)))
    {
        if (is_valid_module_name (name))
        {
            path = g_build_filename (manager->module_path,
                                     name, NULL);
            query_one_module (manager, path);
            g_free (path);
        }
    }

    g_dir_close (dir);
}
```



# Managing Your Application's Modules (III)

- Skipping non-module files

```
static gboolean
is_valid_module_name (const gchar *basename)
{
#ifdef G_OS_WIN32 && !defined(G_WITH_CYGWIN)

    return g_str_has_prefix (basename, "lib") &&
           g_str_has_suffix (basename, ".so");

#else

    return g_str_has_suffix (basename, ".dll");

#endif
}
```



# Managing Your Application's Modules (IV)

- Querying & Registering one module

```
static void
query_one_module (FooModuleManager *manager,
                  const gchar      *filename)
{
    FooModule *module = foo_module_new (filename);

    if (! g_type_module_use (G_TYPE_MODULE (module)))
    {
        g_printerr ("Failed to load: %s\n", filename);
        g_object_unref (module);
        return;
    }

    g_type_module_unuse (G_TYPE_MODULE (module));

    manager->modules = g_list_prepend (manager->modules,
                                       module);
}
```



# Managing Your Application's Modules (V)

- Finding and using the registered types

```
GType *filters;
gint  n_filters, i;

filters = g_type_children (FOO_TYPE_FILTER, &n_filters);

for (i = 0; i < n_filters; i++)
{
    FooFilterClass *klass = g_type_class_ref (filters[i]);
    GtkWidget      *button;

    button = gtk_button_new_with_label (klass->name);
    g_object_set_data (G_OBJECT (button), "filter-type",
                      (gpointer) filters[i]);
    g_type_class_unref (klass);
}

g_free (filters);
```



# Common Problems and Mistakes

- `G_MODULE_BIND_LAZY / G_MODULE_BIND_LOCAL`
- **Never** unref a `GTypeModule` instance
- **Don't** let the modules use any symbol from the application (not portable)
- They can use library functions and passed-in callbacks
- All static variables in the module **go away** on unloading
  - **Don't** use `g_quark_from_static_string()`
  - **Don't** use `G_PARAM_STATIC_FOO` or anything that assumes the strings are allocated forever
- Can **forbid unloading** by calling `g_module_make_resident()`



# GTypeModule Pros and Cons

- Easy, ready-to-use Framework
- Tightly integrated with the GType system
- Modules live in application's address space (fast!)
- Modules live in application's address space (crash!)
- GTypes can't be unregistered again



# Questions

- Questions?
- Example code:  
<http://people.imendio.com/~mitch/foo-editor-0.1.tar.gz>
- Now please take me to a TV screen and buy me a beer