NAME

Glib::devel − Binding developer's overview of Glib's internals

DESCRIPTION

Do you need to know how the gtk2−perl language bindings work, or need to write your own language bindings for a Glib/Gtk2−based library? Then you've come to the right place. If you are just a perl developer wanting to write programs with Glib or Gtk2, then this is probably way over your head.

This document began its life as a post to gtk-perl-list about a redesign of the fundamentals of the bindings; today it is the reference documentation for the developers of the bindings.

To reduce confusion, refer to GLib, the C library, with a capital L, and Glib the perl module with a lower-case l. While the Gtk2 module is the primary client of Glib, it is not necessarily the only one; in fact, the perl bindings for the GStreamer library build directly atop Glib. Therefore, this document describes just the GLib/Glib basics. For details on how Gtk2 extends upon the concepts presented here, see Gtk2::devel.

In various places, we use the name GPerl to refer to the actual binding subsystem.

In order to avoid getting very quickly out of date, this document doesn't go into great detail on APIs. gperl.h is rather heavily commented, and should be considered the canonical source of correct API information.

Basic Philosophy

GLib is a portability library for C programs, providing a common set of APIs and services on various platforms. Along with that you get libgobject, which provides an inheritance-based type system and other spiffy things.

Glib, as a perl module, must decide which portions of GLib's facilities to map to perl and which to abstract and encapsulate.

In the grand scheme, the bindings have been designed with a few basic tenets in mind:

- Stick close to the C API, to allow a perl developer to use knowledge from the C API and API reference docs with the perl bindings; this is overruled in some places by the remaining tenets.
- Be perlish. This is the most important. The user of the perl bindings should not have to worry about memory management, reference counting, freeing objects, and all that stuff, else he might as well go write in C instead.
- Leave out deprecated functionality.
- Don't add new functionality. The exceptions to this rule are consolidation of methods where default parameters may be used, or where the direct analog from C is not practical.
- Be lightweight. As little indirection and bloat as possible. If possible, implement each toplevel module (e.g., Glib, Gtk2, Gnome2, GtkHTML, etc) as one .pm and one .so.
- Be extensible. Export header files and typemaps so that other modules can easily chain off of our base. Do not require the entirely of Gtk2 for someone who needs only to build atop GObject.

The Glib Module

In keeping with the tenet of not requiring the entire car for someone who only needs a single wheel, I broke the glib/gobject library family into its own module and namespace. This has proved to be a godsend, as it has made things very easy to debug; there's a clean separation between the base of the type system and the stuff on top of it.

The Glib module takes care of all the basic types handled by the GObject library −−− GEnum, GFlags, GBoxed, GObject, GValue, GClosure −−− as well has signal marshalling and such in GSignal. I'll discuss each of these separately.

In practice, you will rarely see direct calls to the functions that convert objects in and out of perl. Most code should use the C preprocessor to provide easier-to-remember names that follow the perl API style, e.g., newSVGObject(obj) rather than gperl_new_object(type,obj) and SvGObject(sv) instead of gperl_get_gobject(type, sv). The convention used in all of gtk2−perl is described in Gtk2::devel.

Wrappers

FIXME maybe this section should be rolled into the GBoxed and GObject sections?

In order to use the C data structures from Perl, we need to wrap those objects up in Perl objects. In general, a Perl object is simply a blessed reference. A typical scheme for representing C objects in perl is bless a reference to a scalar holding the C pointer value; perl will destroy the reference-counted scalar when there are no more references to it, and one would normally destroy the underlying data structure at this point. However, GLib is a little more complex than your typical C library, so this easy, typical setup won't work for us.

GBoxed types are opaque wrappers for C structures, providing copy and free functions, to allow the types to be used generically. For the most part we can get away with using the typical scheme described above to provide an opaque object, but in some instances an opaque object is very alien in perl. The Glib::Boxed section explains how we get around this.

GObject, on the other hand, is a type-aware, reference-counted object with lifetime semantics that differ somewhat from perl SVs. Thus we need something a bit more sophisticated than a plain old opaque wrapper; in fact, we use a blessed hash reference with the pointer to the C object tucked away in attached magic, and a pointer to the SV stored in the GObject's user data. The combined perl/C object does some nifty reference-count borrowing to ensure that object lifetime is managed correctly.

If an object is created by a function that returns directly to perl, then the wrapper returned by that function should ''own'' the object. If no other code assumes ownership of that object (by ref'ing a GObject or copying a GBoxed), then the object should be destroyed when the perl scalar is destroyed (actually, as part of its destruction).

If a function returns a preexisting object owned by someone else, then the bindings should NOT destroy the object with the perl wrapper. How we handle this for the various types is described below.

GType to Package Mappings

GType is the GObject library's unique type identifier; this is a runtime variable, because GLib types may be loaded dynamically. The direct analog in perl is the package name, which uniquely specifies an object's class. Since these do about the same thing, we completely replace the GType with the perl package.

For various reasons, mostly to do with robustness and performance, there is a one-to-one mapping between GType classes and perl package names. These must be registered, usually as part of the module initialization process.

In addition, the type system tries as hard as it can to recover when things don't go well, using the GType system to its advantage. If you return a C object of a type that is not registered with Gperl, such as MyCustomTypeFoo, gperl_new_object (see below) will warn you that it has blessed the unknown MyCustomTypeFoo into the first known package in its ancestry, Gtk2::VBox.

GBoxed and GObject have distinct mapping registries to avoid cross-pollination and mistakes in the type system. See below.

To assist in handling inheritance that isn't specified directly by the GType system, the function gperl_set_isa allows you to add elements to the `@ISA` for a package. gperl_register_object does this for you, but you may need to add additional parents, e.g., for implementing GInterfaces. (see Gtk2/xs/GtkEntry.xs for an example)

You may be thinking that we could use substitution rules to map the GObject classes to perl packages. In practice, this is a bad idea, fraught with problems; the substitution rules are not easily extendable and are easily broken by extension packages which don't follow the naming conventions.

GEnums and GFlags

GLib provides a mechanism for creating runtime type information about enumeration and flag types. Enumerations are lists of specific values, one of which may be used at at time, whereas multiple flag values may be supplied at a time. In C flags are meant to be used with bitfields. A GType is associated with the various valid values for a given GEnum or GFlags type as strings, in both full-name and nickname forms.

GPerl uses this mechanism to avoid the need to know integer values for enum and flag types at the perl

level. An enum value is just a string; a bitfield of flag values is represented as a reference to an array of strings. These strings are the GLib-provided nicknames. For the convenience of a perl developer, the bindings treat '−' and '_' as equivalent when looking up the corresponding integer values during conversion.

A GEnum or GFlags type mapping should be registered with

```
void gperl_register_fundamental (GType gtype, const char * package);
```

so that their package names can be used where a GType is required (for example, as GObject property types or GtkTreeModel column types).

The basic functions for converting between C and perl values are

```
/* croak if val is not part of type, otherwise return
 * corresponding value.  this is the general case. */
gint gperl_convert_enum (GType type, SV * val);

/* return a scalar which is the nickname of the enum value
 * val, or croak if val is not a member of the enum. */
SV * gperl_convert_back_enum (GType type, gint val);

/* collapse a list of strings to an integer with all the
 * correct bits set, croak if anything is invalid. */
gint gperl_convert_flags (GType type, SV * val);

/* convert a bitfield to a list of strings, or croak. */
SV * gperl_convert_back_flags (GType type, gint val);
```

Other utility functions allow for finer-grained control, such as the ability to pass unknown values, which can be necessary in special cases. In general, each of these functions raises an exception when something goes wrong. To be helpful, they croak with a message listing the valid values when they encounter invalid input.

GBoxed

GBoxed provides a way to register functions that create, copy, and destroy opaque structures. For our purposes, we'll allow any perl package to inherit from Glib::Boxed and implement accessors for the struct members, but Glib::Boxed will handle the object and wrapper lifetime issues.

There are two functions for creating boxed wrappers:

```
SV * gperl_new_boxed (gpointer boxed, GType gtype, gboolean own);
SV * gperl_new_boxed_copy (gpointer boxed, GType gtype);
```

If own is TRUE, the wrapper returned by gperl_new_boxed will take boxed with it when it dies. In the case of a copy, own is implied, so there's a separate function which doesn't need the own option.

To get a boxed pointer out of a scalar wrapper, you just call gperl_get_boxed_check −−− this will croak if the sv is undef or not blessed into the specified package.

When you register a boxed type you get the option of supplying a table of function pointers describing how the boxed object should be wrapped, unwrapped, and destroyed. This allows you to decide in the wrapping function what subclass of the boxed type's class the wrapper should actually take (a trick used by Gtk2::Gdk::Event), or represent a boxed type as a native perl type (such as using array references for Gnome2::Canvas::Point objects). All of this happens automagically, behind the scenes, and most types assume the default wrapper class.

See the commentary in gperl.h for more information.

GObject

The GObject knows its own type. Thus, we need only one parameter to create a GObject wrapper. In reality, we ask for two:

```
SV * gperl_new_object (GObject * object, gboolean own);
```

The wrapper SV will be blessed into the package corresponding to the gtype returned by G_OBJECT_TYPE (object), that is, the bottommost type in the inheritance chain. If that bottommost type is not known, the function walks back up the tree until it finds one that's known, blesses the reference into that package, and spits out a warning on stderr. To hush the warning, you need merely call

In general, this process will claim a reference on the GObject (with **g_object_ref()**), so that the C object stays alive so long as there is a perl wrapper for it. If own is set to TRUE, the perl wrapper will claim ownership of the C object by removing that reference; in theory, for a new GObject, fresh from a constructor, this leaves the object with a single reference owned by the perl object. The next question out of your mouth should be, ''But what about GObject derivatives that require sinking or other strange methods to claim ownership?'' For the answer, see the GtkObject section's description of sink functions.

```
void gperl_register_object (GType gtype, const char * package);
```

This magical function also sets up the `@ISA` for the package to point to the package corresponding to g_type_parent (gtype). [Since this requires the parent package to be registered, there is a simple deferral mechanism, which means your `@ISA` might not be set until the next call to gperl_register_object.]

There are two ways to get an object out of an SV (though I think only one is really needed):

```
GObject * gperl_get_object (SV * sv);
GObject * gperl_get_object_check (SV * sv, GType gtype);
```

The second one is like the first, but croaks if the object is not derived from gtype.

You can get and set object data and object parameters just like you'd expect.

GSignal

All of this GObject stuff wouldn't be very useful if you couldn't connect signals and closures. I got most of my handling code from gtk2−perl and pygtk, and it's pretty straightforward. The data member is optional, and must be a scalar.

To connect perl subroutines to GSignals I use GClosures, which require the handling of GValues.

GPerlClosure

Use a GPerlClosure wherever you could use a GClosure and things should work out great. *FIXME say more here*

GPerlCallback

Function pointers are required in many places throughout gtk+, usually for a callback to be used as a ''foreach'' function or for some other purpose. Unfortunately, a majority of these spots aren't designed to work with GClosures (usually by lacking a way to destroy data associated with the callback when it is no longer needed). For this purpose, the GPerlCallback wraps up the gruntwork of using perl's call_sv to use a callback function directly.

SEE ALSO

perl(1), **perlxs**(1), **perlguts**(1), **perlapi**(1), **perlxstut**(1), ExtUtils::Depends(3pm), ExtUtils::PkgConfig(3pm) Glib(3pm), Glib::Object::Subclass(3pm), Glib::xsapi(3pm)

AUTHOR

muppet <scott at asofyet.org>

COPYRIGHT