NAME

Glib::xsapi − internal API reference for GPerl.

SYNOPSIS

```
#include <gperl.h>
```

DESCRIPTION

This is the binding developer's API reference for GPerl, automatically generated from the xs source files. This header defines the public interface for use when creating new Perl language bindings for GLib-based C libraries.

gperl.h includes for you all the headers needed for writing XSUBs (EXTERN.h, perl.h, and XSUB.h), as well as all of GLib (via glib−object.h).

API

Miscellaneous

Various useful utilities defined in Glib.xs.

GPERL_CALL_BOOT(name)
: call the boot code of a module by symbol rather than by name.

  in a perl extension which uses several xs files but only one pm, you need to bootstrap the other xs files in order to get their functions exported to perl. if the file has MODULE = Foo::Bar, the boot symbol would be boot_Foo__Bar.

void _gperl_call_XS (pTHX_ void (*subaddr) (pTHX_ CV *), CV * cv, SV ** mark);
: never use this function directly. see`GPERL_CALL_BOOT`.

  for the curious, this calls a perl sub by function pointer rather than by name; call_sv requires that the xsub already be registered, but we need this to call a function which will register xsubs. this is an evil hack and should not be used outside of the GPERL_CALL_BOOT macro. it's implemented as a function to avoid code size bloat, and exported so that extension modules can pull the same trick.

gpointer gperl_alloc_temp (int nbytes)
: Allocate and return a pointer to an *nbytes*−long, zero-initialized, temporary buffer that will be reaped at the next garbage collection sweep. This is handy for allocating things that need to be alloc'ed before a croak (since croak doesn't return and give you the chance to free them). The trick is that the memory is allocated in a mortal perl scalar. See the perl online manual for notes on using this technique.

  Do **not** under any circumstances attempt to call **g_free()**, **free()**, or any other deallocator on this pointer, or you will crash the interpreter.

gchar *gperl_filename_from_sv (SV *sv)
: Return a localized version of the filename in the sv, using g_filename_from_utf8 (and consequently this function might croak). The memory is allocated using gperl_alloc_temp.

SV *gperl_sv_from_filename (const gchar *filename)
: Convert the filename into an utf8 string as used by gtk/glib and perl.

gboolean gperl_str_eq (const char * a, const char * b);
: Compare a pair of ascii strings, considering '−' and '_' to be equivalent. Used for things like enum value nicknames and signal names.

guint gperl_str_hash (gconstpointer key)
: Like **g_str_hash()**, but considers '−' and '_' to be equivalent.

GPerlArgv * gperl_argv_new ()
: Creates a new Perl argv object whose members can then be passed to functions that request argc and argv style arguments.

  If the called function(s) modified argv, you can call gperl_argv_update to update Perl's `@ARGV` in the same way.

Remember to call gperl_argv_free when you're done.

void gperl_argv_update (GPerlArgv *pargv)
Updates `@ARGV` to resemble the stored argv array.

void gperl_argv_free (GPerlArgv *pargv)
Frees any resources associated with *pargv*.

char * gperl_format_variable_for_output (SV * sv)
Formats the variable stored in *sv* for output in error messages. Like **SvPV_nolen()**, but ellipsizes real strings (i.e., not stringified references) at 20 chars to trim things down for error messages.

gboolean gperl_sv_is_defined (SV *sv)
Checks the SV *sv* for definedness just like Perl's ***defined()*** would do. Most importantly, it correctly handles ''magical'' SVs, unlike bare *SvOK*. It's also NULL-safe.

void gperl_hv_take_sv (HV *hv, const char *key, size_t key_length, SV *sv)
Tries to store *sv* in *hv*. Decreases*sv*' s reference count if something goes wrong.

GError Exception Objects

GError is a facility for propagating run-time error / exception information around in C, which is a language without native support for exceptions. GError uses a simple error code, usually defined as an enum. Since the enums will overlap, GError includes the GQuark corresponding to a particular error ''domain'' to tell you which error codes will be used. There's also a string containing a specific error message. The strings are arbitrary, and may be translated, but the domains and codes are definite.

Perl has native support for exceptions, using `eval` as ''try'', `croak` or `die` as ''throw'', and `if ($@)` as ''catch''. `$@` may, in fact, be any scalar, including blessed objects.

So, GPerl maps GLib's GError to Perl exceptions.

Since, as we described above, error messages are not guaranteed to be unique everywhere, we need to support the use of the error domains and codes. The obvious choice here is to use exception objects; however, to support blessed exception objects, we must perform a little bit of black magic in the bindings. There is no built-in association between an error domain quark and the GType of the corresponding error code enumeration, so the bindings supply both of these when specifying the name of the package into which to bless exceptions of this domain. All GError-based exceptions derive from Glib::Error, of course, and this base class provides all of the functionality, including stringification.

All you'll really ever need to do is register error domains with `gperl_register_error_domain`, and throw errors with `gperl_croak_gerror`.

void gperl_register_error_domain (GQuark domain, GType error_enum, const char * package)
Tell the bindings to bless GErrors with error->domain == *domain* into *package*, and use *error_enum* to find the nicknames for the error codes. This will call `gperl_set_isa` on *package* to add ''Glib::Error'' to *package*'s `@ISA`.

domain may not be 0, and *package* may not be NULL; what would be the point? *err or_enum* may be 0, in which case you'll get no fancy stringified error values.

SV * gperl_sv_from_gerror (GError * error)
You should rarely, if ever, need to call this function. This is what turns a GError into a Perl object.

gperl_gerror_from_sv (SV * sv, GError ** error)
You should rarely need this function. This parses a perl data structure into a GError. If*sv* is undef (or the empty string), sets **error* to NULL, otherwise, allocates a new GError with `g_error_new_literal()` and writes through *error*; the caller is responsible for calling `g_error_free()`. (**gperl_croak_gerror()** does this, for example.)

void gperl_croak_gerror (const char * ignored, GError * err)
Croak with an exception based on *err*. *err* may not be NULL. *ignored* exists for backward compatibility, and is, well, ignored. This function calls **croak()**, which does not return.

Since **croak()** does not return, this function handles the magic behind not leaking the memory

associated with the #GError. To use this you'd do something like

```
PREINIT:
  GError * error = NULL;
CODE:
  if (!funtion_that_can_fail (something, &error))
     gperl_croak_gerror (NULL, error);
```

It's just that simple!

GLog

GLib has a message logging mechanism which it uses for the **g_return_if_fail()** assertion macros, etc.; it's really versatile and allows you to set various levels to be fatal and whatnot. Libraries use these for various types of message reporting.

These functions let you reroute those messages from Perl. By default, the warning, critical, and message levels go through perl's **warn()**, and fatal ones go through **croak()**. [i'm not sure that these get to**cr oak()** before GLib **abort()**s on them...]

gint gperl_handle_logs_for (const gchar * log_domain)

Route all g_logs for *log_domain* through gperl's log handling. You'll have to register domains in each binding submodule, because there's no way we can know about them down here.

And, technically, this traps all the predefined log levels, not any of the ones you (or your library) may define for yourself.

GType / GEnum / GFlags

void gperl_register_fundamental (GType gtype, const char * package)

register a mapping between *gtype* and *package*. this is for ''fundamental'' types which have no other requirements for metadata storage, such as GEnums, GFlags, or real GLib fundamental types like G_TYPE_INT, G_TYPE_FLOAT, etc.

void gperl_register_fundamental_alias (GType gtype, const char * package)

Makes *package* an alias for *type*. This means that the package name specified by*pac kage* will be mapped to *type* by *gperl_fundamental_type_from_package*, but *gperl_fundamental_package_from_type* won't map *type* to *package*. This is useful if you want to change the canonical package name of a type while preserving backwards compatibility with code which uses *package* to specify *type*.

In order for this to make sense, another package name should be registered for *type* with *gperl_register_fundamental* or *gperl_register_fundamental_full*.

GPerlValueWrapperClass

Specifies the vtable that is to be used to convert fundamental types to and from Perl variables.

```
typedef struct _GPerlValueWrapperClass GPerlValueWrapperClass;
struct _GPerlValueWrapperClass {
        GPerlValueWrapFunc   wrap;
        GPerlValueUnwrapFunc unwrap;
};
```

The members are function pointers, each of which serves a specific purpose:

GPerlValueWrapFunc

Turns *value* into an SV. The caller assumes ownership of the SV. *value*is not to be modified.

```
typedef SV*  (*GPerlValueWrapFunc)   (const GValue * value);
```

GPerlValueUnwrapFunc

Turns *sv* into its fundamental representation and stores the result in the pre-configured *value*. *value* must not be overwritten; instead one of the various `g_value_set_*()` functions must be used or the `value->data` pointer must be modified directly.

```
typedef void (*GPerlValueUnwrapFunc) (GValue       * value,
                                      SV           * sv);
```

void gperl_register_fundamental_full (GType gtype, const char * package, GPerlValueWrapperClass * wrapper_class)

Like gperl_register_fundamental, registers a mapping between *gtype* and *package*. In addition, this also installs the function pointers in *wrapper_class* as the handlers for the type. See GPerlValueWrapperClass.

gperl_register_fundamental_full does not copy the contents of *wrapper_class* — it assumes that *wrapper_class* is statically allocated and that it will be valid for the whole lifetime of the program.

GType gperl_fundamental_type_from_package (const char * package)

look up the GType corresponding to a *package* registered by **gperl_register_fundamental()**.

const char * gperl_fundamental_package_from_type (GType gtype)

look up the package corresponding to a *gtype* registered by **gperl_register_fundamental()**.

GPerlValueWrapperClass * gperl_fundamental_wrapper_class_from_type (GType gtype)

look up the wrapper class corresponding to a *gtype* that has previously been registered with **gperl_register_fundamental_full()**.

gboolean gperl_try_convert_enum (GType gtype, SV * sv, gint * val)

return FALSE if *sv* can't be mapped to a valid member of the registered enum type *gtype*; otherwise, return TRUE write the new value to the int pointed to by *val*.

you'll need this only in esoteric cases.

gint gperl_convert_enum (GType type, SV * val)

croak if *val* is not part of *type*, otherwise return corresponding value

SV * gperl_convert_back_enum_pass_unknown (GType type, gint val)

return a scalar containing the nickname of the enum value *val*, or the integer value of *val* if *val* is not a member of the enum *type*.

SV * gperl_convert_back_enum (GType type, gint val)

return a scalar which is the nickname of the enum value val, or croak if val is not a member of the enum.

gboolean gperl_try_convert_flag (GType type, const char * val_p, gint * val)

like **gperl_try_convert_enum()**, but for GFlags.

gint gperl_convert_flag_one (GType type, const char * val)

croak if *val* is not part of *type*, otherwise return corresponding value.

gint gperl_convert_flags (GType type, SV * val)

collapse a list of strings to an integer with all the correct bits set, croak if anything is invalid.

SV * gperl_convert_back_flags (GType type, gint val)

convert a bitfield to a list of strings.

Inheritance management

void gperl_set_isa (const char * child_package, const char * parent_package)

tell perl that *child_package* inherits *parent_package*, after whatever else is already there. equivalent to `push @{$parent_package}::ISA, $child_package;`

void gperl_prepend_isa (const char * child_package, const char * parent_package)

tell perl that *child_package* inherits *parent_package*, but before whatever else is already there. equivalent to `unshift @{$parent_package}::ISA, $child_package;`

GType gperl_type_from_package (const char * package)

Look up the GType associated with *package*, regardless of how it was registered. Returns 0 if no mapping can be found.

const char * gperl_package_from_type (GType gtype)
Look up the name of the package associated with *gtype*, regardless of how it was registered. Returns NULL if no mapping can be found.

Boxed type support for SV

In order to allow GValues to hold perl SVs we need a GBoxed wrapper.

GPERL_TYPE_SV
Evaluates to the GType for SVs. The bindings register a mapping between GPERL_TYPE_SV and the package 'Glib::Scalar' with **gperl_register_boxed()**.

SV * gperl_sv_copy (SV * sv)
implemented as `newSVsv (sv)`.

void gperl_sv_free (SV * sv)
implemented as `SvREFCNT_dec (sv)`.

UTF-8 strings with gchar

By convention, gchar* is assumed to point to UTF8 string data, and char* points to ascii string data. Here we define a pair of wrappers for the boilerplate of upgrading Perl strings. They are implemented as functions rather than macros, because comma expressions in macros are not supported by all compilers.

These functions should be used instead of newSVpv and SvPV_nolen in all cases which deal with gchar* types.

gchar * SvGChar (SV * sv)
extract a UTF8 string from *sv*.

SV * newSVGChar (const gchar * str)
copy a UTF8 string into a new SV. if str is NULL, returns &PL_sv_undef.

64 bit integers

On 32 bit machines and even on some 64 bit machines, perl's IV/UV data type can only hold 32 bit values. The following functions therefore convert 64 bit integers to and from Perl strings if normal IV/UV conversion does not suffice.

gint64 SvGInt64 (SV *sv)
Converts the string in *sv* to a signed 64 bit integer. If appropriate, uses`SvIV` instead.

SV * newSVGInt64 (gint64 value)
Creates a PV from the signed 64 bit integer in *value*. If appropriate, uses`newSViv` instead.

guint64 SvGUInt64 (SV *sv)
Converts the string in *sv* to an unsigned 64 bit integer. If appropriate, uses`SvUV` instead.

SV * newSVGUInt64 (guint64 value)
Creates a PV from the unsigned 64 bit integer in *value*. If appropriate, uses`newSVuv` instead.

GBoxed

GPerlBoxedWrapperClass
Specifies the vtable of functions to be used for bringing boxed types in and out of perl. The structure is defined like this:

```
typedef struct _GPerlBoxedWrapperClass GPerlBoxedWrapperClass;
struct _GPerlBoxedWrapperClass {
        GPerlBoxedWrapFunc    wrap;
        GPerlBoxedUnwrapFunc  unwrap;
        GPerlBoxedDestroyFunc destroy;
};
```

The members are function pointers, each of which serves a specific purpose:

GPerlBoxedWrapFunc

turn a boxed pointer into an SV. gtype is the type of the boxed pointer, and package is the package to which that gtype is registered (the lookup has already been done for you at this point). if own is true, the wrapper is responsible for freeing the object; if it is false, some other code owns the object and you must NOT free it.

```
typedef SV*      (*GPerlBoxedWrapFunc)    (GType        gtype,
                                           const char * package,
                                           gpointer     boxed,
                                           gboolean     own);
```

GPerlBoxedUnwrapFunc

turn an SV into a boxed pointer. like GPerlBoxedWrapFunc, gtype and package are the registered type pair, already looked up for you (in the process of finding the proper wrapper class). sv is the sv to unwrap.

```
typedef gpointer (*GPerlBoxedUnwrapFunc)  (GType        gtype,
                                           const char * package,
                                           SV         * sv);
```

GPerlBoxedDestroyFunc

this will be called by Glib::Boxed::DESTROY, when the wrapper is destroyed. it is a hook that allows you to destroy an object owned by the wrapper; note, however, that you will have had to keep track yourself of whether the object was to be freed.

```
typedef void     (*GPerlBoxedDestroyFunc) (SV         * sv);
```

void gperl_register_boxed (GType gtype, const char * package, GPerlBoxedWrapperClass * wrapper_class)

Register a mapping between the GBoxed derivative *gtype* and *package*. The specified,*wr apper_class* will be used to wrap and unwrap objects of this type; you may pass NULL to use the default wrapper (the same one returned by **gperl_default_boxed_wrapper_class()**).

In normal usage, the standard opaque wrapper supplied by the library is sufficient and correct. In some cases, however, you want a boxed type to map directly to a native perl type; for example, some struct may be more appropriately represented as a hash in perl. Since the most necessary place for this conversion to happen is in **gperl_value_from_sv()** and **gperl_sv_from_value()**, the only reliable and robust way to implement this is a hook into **gperl_get_boxed_check()** and **gperl_new_boxed()**; that is exactly the purpose of *wrapper_class*. See`GPerlBoxedWrapperClass`.

gperl_register_boxed does not copy the contents of *wrapper_class* — it assumes that *wrapper_class* is statically allocated and that it will be valid for the whole lifetime of the program.

void gperl_register_boxed_alias (GType gtype, const char * package)

Makes *package* an alias for *type*. This means that the package name specified by*pac kage* will be mapped to *type* by *gperl_boxed_type_from_package*, but *gperl_boxed_package_from_type* won't map *type* to *package*. This is useful if you want to change the canonical package name of a type while preserving backwards compatibility with code which uses *package* to specify *type*.

In order for this to make sense, another package name should be registered for *type* with *gperl_register_boxed*.

void gperl_register_boxed_synonym (GType registered_gtype, GType synonym_gtype)

Registers *synonym_gtype* as a synonym for *registered_gtype*. All boxed objects of type *synonym_gtype* will then be treated as if they were of type *registered_gtype*, and *gperl_boxed_package_from_type* will return the package associated with *registered_gtype*.

registered_gtype must have been registered with *gperl_register_boxed* already.

GType gperl_boxed_type_from_package (const char * package)

Look up the GType associated with package *package*. Returns 0 if*type* is not re gistered.

const char * gperl_boxed_package_from_type (GType type)
Look up the package associated with GBoxed derivative *type*. Returns NULL if *type* is not registered.

GPerlBoxedWrapperClass * gperl_default_boxed_wrapper_class (void)
get a pointer to the default wrapper class; handy if you want to use the normal wrapper, with minor modifications. note that you can just pass NULL to **gperl_register_boxed()**, so you really only need this in fringe cases.

SV * gperl_new_boxed (gpointer boxed, GType gtype, gboolean own)
Export a GBoxed derivative to perl, according to whatever GPerlBoxedWrapperClass is registered for *gtype*. In the default implementation, this means wrapping an opaque perl object around the pointer to a small wrapper structure which stores some metadata, such as whether the boxed structure should be destroyed when the wrapper is destroyed (controlled by *own*; if the wrapper owns the object, the wrapper is in charge of destroying it's data).

This function might end up calling other Perl code, so if you use it in XS code for a generic GType, make sure the stack pointer is set up correctly before the call, and restore it after the call.

SV * gperl_new_boxed_copy (gpointer boxed, GType gtype)
Create a new copy of *boxed* and return an owner wrapper for it. *boxed* may not be NULL. See `gperl_new_boxed`.

gpointer gperl_get_boxed_check (SV * sv, GType gtype)
Extract the boxed pointer from a wrapper; croaks if the wrapper *sv* is not blessed into a derivative of the expected *gtype*. Does not allow undef.

GObject

To deal with the intricate interaction of the different reference-counting semantics of Perl objects versus GObjects, the bindings create a combined PerlObject+GObject, with the GObject's pointer in magic attached to the Perl object, and the Perl object's pointer in the GObject's user data. Thus it's not really a ''wrapper'', but we refer to it as one, because ''combined Perl object + GObject'' is a cumbersome and confusing mouthful.

GObjects are represented as blessed hash references. The GObject user data mechanism is not typesafe, and thus is used only for unsigned integer values; the Perl-level hash is available for any type of user data. The combined nature of the wrapper means that data stored in the hash will stick around as long as the object is alive.

Since the C pointer is stored in attached magic, the C pointer is not available to the Perl developer via the hash object, so there's no need to worry about breaking it from perl.

Propers go to Marc Lehmann for dreaming most of this up.

void gperl_register_object (GType gtype, const char * package)
tell the GPerl type subsystem what Perl package corresponds with a given GObject by GType. automagically sets up *@package*::ISA for you.

note that `@ISA` will not be created for gtype until gtype's parent has been registered. if you are experiencing strange problems with a class' `@ISA` not being set up, change the order in which you register them.

void gperl_register_object_alias (GType gtype, const char * package)
Makes *package* an alias for *type*. This means that the package name specified by*pac kage* will be mapped to *type* by *gperl_object_type_from_package*, but *gperl_object_package_from_type* won't map *type* to *package*. This is useful if you want to change the canonical package name of a type while preserving backwards compatibility with code which uses *package* to specify *type*.

In order for this to make sense, another package name should be registered for *type* with *gperl_register_object*.

void gperl_register_sink_func (GType gtype, GPerlObjectSinkFunc func)
Tell **gperl_new_object()** to use *func* to claim ownership of objects derived from *gtype*.

gperl_new_object() always refs a GObject when wrapping it for the first time. To have the Perl wrapper claim ownership of a GObject as part of **gperl_new_object()**, you unref the object after ref'ing it. however, different GObject subclasses have different ways to claim ownership; for example, GtkObject simply requires you to call **gtk_object_sink()**. To make this concept generic, this function allows you to register a function to be called when then wrapper should claim ownership of the object. The *func* registered for a given *type* will be called on any object for which `g_type_isa (G_TYPE_OBJECT (object), type)`succeeds.

If no sinkfunc is found for an object, **g_object_unref()** will be used.

Even though GObjects don't need sink funcs, we need to have them in Glib as a hook for upstream objects. If we create a GtkObject (or any other type of object which uses a different way to claim ownership) via Glib::Object−>new, any upstream wrappers, such as **gtk2perl_new_object()**, will **not** be called. Having a sink func facility down here enables us always to do the right thing.

void gperl_object_set_no_warn_unreg_subclass (GType gtype, gboolean nowarn)
In versions 1.00 through 1.10x of Glib, the bindings required all types to be registered ahead of time. Upon encountering an unknown type, the bindings would emit a warning to the effect of ''unknown type 'Foo'; representing as first known parent type 'Bar'''. However, for some types, such as GtkStyle or GdkGC, the actual object returned is an instance of a child type of a private implementation (e.g., a theme engine (''BlueCurveStyle'') or gdk backend (''GdkGCX11'')); we neither can nor should have registered names for these types. Therefore, it is possible to tell the bindings not to warn about these unregistered subclasses, and simply represent them as the parent type.

With 1.12x, the bindings will automatically register unknown classes into the namespace Glib::Object::_Unregistered to avoid possible breakage resulting from unknown ancestors of known children. To preserve the old registered-as-unregistered behavior, the value installed by this function is used to prevent the _Unregistered mapping for such private backend classes.

Note: this assumes *gtype* has already been registered with **gperl_register_object()**.

const char * gperl_object_package_from_type (GType gtype)
Get the package corresponding to *gtype*. If*gtype* is not a GObject or GInterf ace, returns NULL. If *gtype* is not registered to a package name, a new name of the form `Glib::Object::_Unregistered::$c_type_name` will be created, used to register the class, and then returned.

HV * gperl_object_stash_from_type (GType gtype)
Get the stash corresponding to *gtype*; returns NULL if *gtype* is not registered. The stash is useful for `bless`ing.

GType gperl_object_type_from_package (const char * package)
Inverse of **gperl_object_package_from_type()**, returns 0 if*pac kage* is not registered.

SV * gperl_new_object (GObject * object, gboolean own)
Use this function to get the perl part of a GObject. If *object* has never been seen by perl before, a new, empty perl object will be created and added to a private key under *object*'s qdata. If *object* already has a perl part, a new reference to it will be created. The gobject + perl object together form a combined object that is properly refcounted, i.e. both parts will stay alive as long as at least one of them is alive, and only when both perl object and gobject are no longer referenced will both be freed.

The perl object will be blessed into the package corresponding to the GType returned by calling **G_OBJECT_TYPE()** on *object*; if that class has not been registered via **gperl_register_object()**, this function will emit a warning to that effect (with **warn()**), and attempt to bless it into the first known class in the object's ancestry. Since Glib::Object is already registered, you'll get a Glib::Object if you are lazy, and thus this function can fail only if *object* isn't descended from GObject, in which case it croaks. (In reality, if you pass a non-GObject to this function, you'll be lucky if you don't get a segfault, as there's not really a way to trap that.) In practice these warnings can be unavoidable, so you can use **gperl_object_set_no_warn_unreg_subclass()** to quell them on a class-by-class basis.

However, when perl code is calling a GObject constructor (any function which returns a new GObject), call **gperl_new_object()** with *own* set to %TRUE; this will cause the first matching sink function to be called on the GObject to claim ownership of that object, so that it will be destroyed when the perl object goes out of scope. The default sink func is **g_object_unref()**; other types should supply the proper function; e.g., GtkObject should use **gtk_object_sink()** here.

Returns the blessed perl object, or #&PL_sv_undef if object was #NULL.

GObject * gperl_get_object (SV * sv)
retrieve the GObject pointer from a Perl object. Returns NULL if *sv* is not linked to a GObject.

Note, this one is not safe — in general you want to use **gperl_get_object_check()**.

GObject * gperl_get_object_check (SV * sv, GType gtype);
croaks if *sv* is undef or is not blessed into the package corresponding to *gtype*. use this for bringing parameters into xsubs from perl. Returns the same as **gperl_get_object()** (provided it doesn't croak first).

SV * gperl_object_check_type (SV * sv, GType gtype)
Essentially the same as **gperl_get_object_check()**.

This croaks if the types aren't compatible.

typedef GObject GObject_noinc
typedef GObject GObject_ornull
newSVGObject(obj)
newSVGObject_noinc(obj)
SvGObject(sv)
SvGObject_ornull(sv)

GValue

GValue is GLib's generic value container, and it is because of GValue that the run time type handling of GObject parameters and GClosure marshaling can function, and most usages of these functions will be from those two points.

Client code will run into uses for **gperl_sv_from_value()** and **gperl_value_from_sv()** when trying to convert lists of parameters into GValue arrays and the like.

gboolean gperl_value_from_sv (GValue * value, SV * sv)
set a *value* from a whatever is in *sv*. *value* must be initialized so the code knows what kind of value to coerce out of *sv*.

Return value is always TRUE; if the code knows how to perform the conversion, it croaks. (The return value is for backward compatibility.) In reality, this really ought to always succeed; a failed conversion should be considered a bug or unimplemented code!

SV * gperl_sv_from_value (const GValue * value)
Coerce whatever is in *value* into a perl scalar and return it.

Croaks if the code doesn't know how to perform the conversion.

Might end up calling other Perl code. So if you use this function in XS code for a generic GType, make sure the stack pointer is set up correctly before the call, and restore it after the call.

GClosure / GPerlClosure

GPerlClosure is a wrapper around the gobject library's GClosure with special handling for marshalling perl subroutines as callbacks. This is specially tuned for use with GSignal and stuff like io watch, timeout, and idle handlers.

For generic callback functions, which need parameters but do not get registered with the type system, this is sometimes overkill. See GPerlCallback, below.

GClosure * gperl_closure_new (SV * callback, SV * data, gboolean swap)
Create and return a new GPerlClosure. *callbac k* and *data* will be copied for storage; *callback* must not be NULL. If*swap* is TR UE, *data* will be swapped with the instance during invocation (this is used to implement **g_signal_connect_swapped()**).

If compiled under a thread-enabled perl, the closure will be created and marshaled in such a way as to ensure that the same interpreter which created the closure will be used to invoke it.

GClosure * gperl_closure_new_with_marshaller (SV * callback, SV * data, gboolean swap, GClosureMarshal marshaller)
Like `gperl_closure_new`, but uses a caller-supplied marshaller. This is provided for use in those sticky circumstances when you just can't do it any other way; in general, you want to use the default marshaller, which you get if you provide NULL for *marshaller*.

If you use you own marshaller, you need to take care of everything yourself, including swapping the instance and data if `GPERL_CLOSURE_SWAP_DATA (closure)` is true, calling `gperl_run_exception_handlers` if ERRSV is true after invoking the perl sub, and ensuring that you properly use the `marshal_data` parameter as the perl interpreter when PERL_IMPLICIT_CONTEXT is defined. See the implementation of the default marshaller, `gperl_closure_marshal`, in Glib/GClosure.xs for inspiration.

GPerlCallback

generic callback functions usually get invoked directly, and are not passed parameter lists as GValues. we could very easily wrap up such generic callbacks with something that converts the parameters to GValues and then channels everything through GClosure, but this has two problems: 1) the above implementation of GClosure is tuned to marshalling signal handlers, which always have an instance object, and 2) it's more work than is strictly necessary.

additionally, generic callbacks aren't always kind to the GClosure paradigm.

so, here's GPerlCallback, which is designed specifically to run generic callback functions. it reads parameters off the C stack and converts them into parameters on the perl stack. (it uses the GValue to/from SV mechanism to do so, but doesn't allocate any temps on the heap.) the callback object itself stores the parameter type list.

unfortunately, since the data element is always last, but the number of arguments is not known until we have the callback object, we can't pass gperl_callback_invoke directly to functions requiring a callback; you'll have to write a proxy callback which calls gperl_callback_invoke.

GPerlCallback * gperl_callback_new (SV * func, SV * data, gint n_params, GType param_types[], GType return_type)
Create and return a new GPerlCallback; use gperl_callback_destroy when you are finished with it.

func: perl subroutine to call. this SV will be copied, so don't worry about reference counts. must **not** be #NULL.

data: scalar to pass to *func* in addition to all other arguments. the SV will be copied, so don't worry about reference counts. may be #NULL.

n_params: the number of elements in *param_types*.

param_types: the #GType of each argument that should be passed from the invocation to *func*. may be #NULL if *n_params* is zero, otherwise it must be *n_params* elements long or nasty things will happen. this array will be copied; see **gperl_callback_invoke()** for how it is used.

return_type: the #GType of the return value, or 0 if the function has void return.

void gperl_callback_destroy (GPerlCallback * callback)
Dispose of *callback*.

void gperl_callback_invoke (GPerlCallback * callback, GValue * return_value, ...)
Marshall the variadic parameters according to *callback*'s param_types, and then invoke *callback*'s subroutine in scalar context, or void context if the return type is G_TYPE_VOID. If*r eturn_value* is

not NULL, then value returned (if any) will be copied into *return_value*.

A typical callback handler would look like this:

```
static gint
real_c_callback (Foo * f, Bar * b, int a, gpointer data)
{
        GPerlCallback * callback = (GPerlCallback*)data;
        GValue return_value = {0,};
        gint retval;
        g_value_init (&return_value, callback->return_type);
        gperl_callback_invoke (callback, &return_value,
                               f, b, a);
        retval = g_value_get_int (&return_value);
        g_value_unset (&return_value);
        return retval;
}
```

Exception Handling

Like Event, Tk, and most other callback-using, event-based perl modules, Glib traps exceptions that happen in callbacks. To enable your code to do something about these exceptions, Glib stores a list of exception handlers which will be called on the trapped exceptions. This is completely distinct from the `$SIG{__DIE__}` mechanism provided by Perl itself, for various reasons (not the least of which is that the Perl docs and source code say that `$SIG{__DIE__}` is intended for running as the program is about to exit, and other behaviors may be removed in the future (apparently a source of much debate on p5p)).

int gperl_install_exception_handler (GClosure * closure)

Install a GClosure to be executed when **gperl_closure_invoke()** traps an exception. The closure should return boolean (TRUE if the handler should remain installed) and expect to receive a perl scalar. This scalar will be a private copy of ERRSV ($@) which the handler can mangle to its heart's content.

The return value is an integer id tag that may be passed to **gperl_removed_exception_handler()**.

void gperl_remove_exception_handler (guint tag)

Remove the exception handler identified by *tag*, as returned by **gperl_install_exception_handler()**. If *tag* cannot be found, this does nothing.

WARNING: this function locks a global data structure, so do NOT call it recursively. also, calling this from within an exception handler will result in a deadlock situation. if you want to remove your handler just have it return FALSE.

void gperl_run_exception_handlers (void)

Invoke whatever exception handlers are installed. You will need this if you have written a custom marshaler. Uses the value of the global ERRSV.

GSignal

void gperl_signal_set_marshaller_for (GType instance_type, char * detailed_signal, GClosureMarshal marshaller)

You need this function only in rare cases, usually as workarounds for bad signal parameter types or to implement writable arguments. Use the given *marshaller* to marshal all handlers for *detailed_signal* on *instance_type*. `gperl_signal_connect` will look for marshallers registered here, and apply them to the GPerlClosure it creates for the given callback being connected.

A canonical form of *detailed_signal* will be used so that *marshaller* is applied for all possible spellings of the signal name.

Use the helper macros in gperl_marshal.h to help write your marshaller function. That header, which is installed with the Glib module but not #included through gperl.h, includes commentary and examples which you should follow closely to avoid nasty bugs. Use the Source, Luke.

WARNING: Bend over backwards and turn your head around 720 degrees before attempting to write a

GPerlClosure marshaller without using the macros in gperl_marshal.h. If you absolutely cannot use those macros, be certain to understand what those macros do so you can get the semantics correct, and keep your code synchronized with them, or you may miss very important bugfixes.

gulong gperl_signal_connect (SV * instance, char * detailed_signal, SV * callback, SV * data, GConnectFlags flags)

The actual workhorse behind GObject::signal_connect, the binding for g_signal_connect, for use from within XS. This creates a`GPerlClosure` wrapper for the gi ven *callback* and *data*, and connects that closure to the signal named *detailed_signal* on the given GObject *instance*. This is only good for named signals. *fla gs* is the same as for **g_signal_connect()**. *data* may be NULL, but *callback* must not be.

Returns the id of the installed callback.

SEE ALSO

perlapi (1), **perlguts** (1), GLib Reference Manual, **Glib** (3pm), **Glib::devel** (3pm).

AUTHORS

This file was automatically generated from the source code of the Glib module, which is maintained by the gtk2−perl team.

LICENSE