NAME

Glib − Perl wrappers for the GLib utility and Object libraries

SYNOPSIS

```
 use Glib;
```

ABSTRACT

This module provides perl access to GLib and GLib's GObject libraries. GLib is a portability and utility library; GObject provides a generic type system with inheritance and a powerful signal system. Together these libraries are used as the foundation for many of the libraries that make up the Gnome environment, and are used in many unrelated projects.

DESCRIPTION

This wrapper attempts to provide a perlish interface while remaining as true as possible to the underlying C API, so that any reference materials you can find on using GLib may still apply to using the libraries from perl. This module also provides facilities for creating wrappers for other GObject-based libraries. The "SEE ALSO" section contains pointers to all sorts of good information.

PERL VERSUS C

GLib provides to C programs many of the same facilities Perl offers natively. Where GLib's functionality overlaps Perl's, Perl's is favored. Some concepts have been eliminated entirely, as Perl is a higher-level language than C. In other instances we've had to add or change APIs to make sense in Perl. Here's a quick run-down:

Perl Already Does That

The GLib types GList (a doubly-linked list), GSList (singly-linked list), GHashTable, GArray, etc have all been replaced by native Perl datatypes. In fact, many functions which take GLists or arrays simply accept lists on the Perl stack. For the most part, GIOChannels are no more functional than Perl file handles, so you won't see any GIOChannels. GClosures are not visible at the Perl level, because Perl code references do the same thing. Just about any function taking either a C function pointer or a GClosure will accept a code reference in Perl. (In fact, you can probably get away with just a subroutine name in many spots, provided you aren't using strict subs.)

Don't Worry About That

Some concepts have been eliminated; you need never worry about reference-counting on GObjects or having to free GBoxed structures. Perl is a garbage-collected language, and we've put a lot of work into making the bindings take care of memory for you in a way that feels natural to a Perl developer. You won't see GValues in Perl (that's just a C structure with Perl scalar envy, anyway).

This Is Now That

Other GLib concepts have been converted to an analogous Perl concept.

The GType id will never be seen in Perl, as the package name serves that purpose. Several packages corresponding to the GTypes of the fundamental types have been registered for you:

```
 G_TYPE_STRING     Glib::String
 G_TYPE_INT        Glib::Int
 G_TYPE_UINT       Glib::UInt
 G_TYPE_DOUBLE     Glib::Double
 G_TYPE_BOOLEAN    Glib::Boolean
```

The remaining fundamentals (char/uchar, short, float, etc) are also registered so that we can properly interact with properties of C objects, but perl really only uses ints, uints, and doubles. Oh, and we created a GBoxed type for Perl scalars so you can use scalars where any boxed type would be allowed (e.g. GtkTreeModel columns):

```
 Glib::Scalar
```

Functions that can return false and set a GError in C raise an exception in Perl, using an exception object based on the GError for $@; see Glib::Error. Trapping exceptions in signals is a sticky issue, so they get their own section; see EXCEPTIONS.

Enumerations and flags are treated as strings and arrays of strings, respectively. GLib provides a way to register nicknames for enumeration values, and the Perl bindings use these nicknames for the real values, so that we never have to deal with numbers in Perl. This can get a little cumbersome for bitfields, but it's very nice when you forget a flag value, as the bindings will tell you what values are accepted when you pass something invalid. Also, the bindings consider the − and _ characters to be equivalent, so that signal and property names can be properly stringified by the => operator. For example, the following are equivalent:

```
  # property foo-matic of type FooType, using the
  # value FOO_SOMETHING_COOL.  its nickname would be
  # 'something-cool'.  you may use either the full
  # name or the nickname when supplying values to perl.
  $object->set ('foo-matic', 'FOO_SOMETHING_COOL');
  $object->set ('foo_matic', 'something_cool');
  $object->set (foo_matic => 'something-cool');
```

Beware that Perl will always return to you the nickname form, with the dash.

Flags have some additional magic abilities in the form of overloaded operators:

```
  + or |   union of two flagsets ("add")
  -        difference of two flagsets ("sub", "remove")
  * or &   intersection of two bitsets ("and")
  / or ^   symmetric difference ("xor", you will rarely need this)
  >=       contains-operator ("is the left set a superset of the right set?")
  ==       equality
```

In addition, flags in boolean context indicate whether they are empty or not, which allows you to write common operations naturally:

```
  $widget->set_events ($widget->get_events - "motion_notify_mask");
  $widget->set_events ($widget->get_events - ["motion_notify_mask",
                                               "button_press_mask"]);

  # shift pressed (both work, it's a matter of taste)
  if ($event->state >= "shift-mask") { ...
  if ($event->state * "shift-mask") { ...

  # either shift OR control pressed?
  if ($event->state * ["shift-mask", "control-mask"]) { ...

  # both shift AND control pressed?
  if ($event->state >= ["shift-mask", "control-mask"]) { ...
```

In general, + and − work as expected to add or remove flags. To test whether *any* bits are set in a mask, you use `$mask * ...`, and to test whether *all* bits are set in a mask, you use `$mask >= ...`.

When dereferenced as an array `@$flags` or `$flags->[...]`, you can access the flag values directly as strings (but you are not allowed to modify the array), and when stringified `"$flags"` a flags value will output a human-readable version of its contents.

It's All the Same

For the most part, the remaining bits of GLib are unchanged. GMainLoop is now Glib::MainLoop, GObject is now Glib::Object, GBoxed is now Glib::Boxed, etc.

FILENAMES, URIS AND ENCODINGS

Perl knows two datatypes, unicode text and binary bytes. Filenames on a system that doesn't use a utf−8 locale are often stored in a local encoding ("binary bytes"). Gtk+ and descendants, however, internally work in unicode most of the time, so when feeding a filename into a GLib/Gtk+ function that expects a filename, you first need to convert it from the local encoding to unicode.

This involves some elaborate guessing, which perl currently avoids, but GLib and Gtk+ do. As an exception, some Gtk+ functions want a filename in local encoding, but the perl interface usually works around this by automatically converting it for you.

In short: Everything should be in unicode on the perl level.

The following functions expose the conversion algorithm that GLib uses.

These functions are only necessary when you want to use perl functions to manage filenames returned by a GLib/Gtk+ function, or when you feed filenames into GLib/Gtk+ functions that have their source outside your program (e.g. commandline arguments, readdir results etc.).

These functions are available as exports by request (see "Exports"), and also support method invocation syntax for pathological consistency with the OO syntax of the rest of the bindings.

`$filename` = filename_to_unicode `$filename_in_local_encoding`
`$filename` = Glib−>filename_to_unicode ($filename_in_local_encoding)

Convert a perl string that supposedly contains a filename in local encoding into a filename represented as unicode, the same way that GLib does it internally.

Example:

```
$gtkfilesel->set_filename (filename_to_unicode $ARGV[1]);
```

This function will **croak**() if the conversion cannot be made, e.g., because the utf−8 is invalid.

`$filename_in_local_encoding` = filename_from_unicode `$filename`
`$filename_in_local_encoding` = Glib−>filename_from_unicode ($filename)

Converts a perl string containing a filename into a filename in the local encoding in the same way GLib does it.

Example:

```
open MY, "<", filename_from_unicode $gtkfilesel->get_filename;
```

It might be useful to know that perl currently has no policy at all regarding filename issues, if your scalar happens to be in utf−8 internally it will use utf−8, if it happens to be stored as bytes, it will use it as-is.

When dealing with filenames that you need to display, there is a much easier way, as of Glib 1.120 and glib 2.6.0:

`$uft8_string` = filename_display_name ($filename)
`$uft8_string` = filename_display_basename ($filename)

Given a *`$filename`* in filename encoding, return the filename, or just the file's basename, in utf−8. Unlike the other functions described above, this one is guaranteed to return valid utf−8, but the conversion is not necessarily reversible. These functions are intended to be used for failsafe display of filenames, for example in gtk+ labels.

Since glib 2.6, Glib 1.12

The following convert filenames to and from URI encoding. (See also URI::file.)

`$string` = filename_to_uri ($filename, `$hostname`)
`$string` = Glib−>filename_to_uri ($filename, `$hostname`)

Return a "file://" schema URI for a filename. Unsafe and non-ascii chars in `$filename` are escaped with URI "%" forms.

`$filename` must be an absolute path as a byte string in local filesystem encoding. `$hostname` is a utf−8 string, or empty or `undef` for no host specified. For example,

```
filename_to_uri ('/my/x%y/<dir>/foo.html', undef);
# returns 'file:///my/x%25y/%3Cdir%3E/foo.html'
```

If `$filename` is a relative path or `$hostname` doesn't look like a hostname then `filename_to_uri` croaks with a `Glib::Error`.

When using the class style `Glib->filename_to_uri` remember that the `$hostname` argument is mandatory. If you forget then it looks like a 2−argument call with filename of "Glib" and hostname of what you meant to be the filename.

`$filename` = filename_from_uri ($uri)
($filename, `$hostname`) = filename_from_uri ($uri)

Extract the filename and hostname from a "file://" schema URI. In scalar context just the filename is returned, in array context both filename and hostname are returned.

The filename returned is bytes in the local filesystem encoding and with the OS path separator character. The hostname returned is utf−8. For example,

```
($f,$h) = filename_from_uri ('file://foo.com/r%26b/bar.html');
# returns '/r&b/bar.html' and 'foo.com' on Unix
```

If `$uri` is not a "file:", or is mal-formed, or the hostname part doesn't look like a host name then `filename_from_uri` croaks with a `Glib::Error`.

EXCEPTIONS

The C language doesn't support exceptions; GLib is a C library, and of course doesn't support exceptions either. In Perl, we use die and eval to raise and trap exceptions as a rather common practice. So, the bindings have to work a little black magic behind the scenes to keep GLib from exploding when the Perl program uses exceptions. Unfortunately, a little of this magic has to leak out to where you can see it at the Perl level.

Signal and event handlers are run in an eval context; if an exception occurs in such a handler and you don't catch it, Perl will report that an error occurred, and then go on about its business like nothing happened.

You may register subroutines as exception handlers, to be called when such an exception is trapped. Another function removes them for you.

```
$tag = Glib->install_exception_handler (\&my_handler);
Glib->remove_exception_handler ($tag);
```

The exception handler will get a fresh copy of the $@ of the offending exception on the argument stack, and is expected to return non-zero if the handler is to remain installed. If it returns false, the handler will be removed.

```
sub my_handler {
    if ($_[0] =~ m/ftang quisinart/) {
         clean_up_after_ftang ();
    }
    1; # live to fight another day
}
```

You can register as many handlers as you like; they will all run independently.

An important thing to remember is that exceptions do not cross main loops. In fact, exceptions are completely distinct from main loops. If you need to quit a main loop when an exception occurs, install a handler that quits the main loop, but also ask yourself if you are using exceptions for flow control or exception handling.

LOG MESSAGES

GLib's g_log function provides a flexible mechanism for reporting messages, and most GLib-based C libraries use this mechanism for warnings, assertions, critical messages, etc. The Perl bindings offer a mechanism for routing these messages through Perl's native system, **warn()** and **die()**. Extensions should register the log domains they wrap for this to happen fluidly. [FIXME say more here]

64 BIT INTEGERS

Since perl's integer data type can only hold 32 bit values on all 32 bit machines and even on some 64 bit machines, Glib converts 64 bit integers to and from strings if necessary. These strings can then be used to feed one of the various big integer modules. Make sure you don't let your strings get into numerical context before passing them into a Glib function because in this case, perl will convert the number to scientific notation which at this point is not understood by Glib's converters.

Here is an overview of what big integer modules are available. First of all, there's Math::BigInt. It has everything you will ever need, but its pure-Perl implementation is also rather slow. There are multiple ways around this, though.

Math::BigInt::FastCalc
: Math::BigInt::FastCalc can help avoid the glacial speed of vanilla Math::BigInt::Calc. Recent versions of Math::BigInt will automatically use Math::BigInt::FastCalc in place of Math::BigInt::Calc when available. Other options include Math::BigInt::GMP or Math::BigInt::Pari, which however have much larger dependencies.

Math::BigInt::Lite
: Then there's Math::BigInt::Lite, which uses native Perl integer operations as long as Perl integers have sufficient range, and upgrades itself to Math::BigInt when Perl integers would overflow. This must be used in place of Math::BigInt.

bigint / bignum / bigfloat
: Finally, there's the bigint/bignum/bigfloat pragmata, which automatically load the corresponding Math:: modules and which will autobox constants. bignum/bigint will automatically use Math::BigInt::Lite if it's available.

EXPORTS

For the most part, gtk2−perl avoids exporting things. Nothing is exported by default, but some functions and constants in Glib are available by request; you can also get all of them with the export tag "all".

Tag: constants

```
TRUE
FALSE
SOURCE_CONTINUE
SOURCE_REMOVE
G_PRIORITY_HIGH
G_PRIORITY_DEFAULT
G_PRIORITY_HIGH_IDLE
G_PRIORITY_DEFAULT_IDLE
G_PRIORITY_LOW
G_PARAM_READWRITE
```

Tag: functions

```
filename_from_unicode
filename_to_unicode
filename_from_uri
filename_to_uri
filename_display_basename
filename_display_name
```

SEE ALSO

Glib::Object::Subclass explains how to create your own gobject subclasses in Perl.

Glib::index lists the automatically-generated API reference for the various packages in Glib.

This module is the basis for the Gtk2 module, so most of the references you'll be able to find about this one are tied to that one. The perl interface aims to be very simply related to the C API, so see the C API reference documentation:

```
GLib − http://developer.gnome.org/doc/API/2.0/glib/
GObject − http://developer.gnome.org/doc/API/2.0/gobject/
```

This module serves as the foundation for any module which needs to bind GLib-based C libraries to perl.

```
Glib::devel - Binding developer's overview of Glib's internals
Glib::xsapi - internal API reference for GPerl
Glib::ParseXSDoc - extract API docs from xs sources.
Glib::GenPod - turn the output of Glib::ParseXSDoc into POD
Glib::MakeHelper - Makefile.PL utilities for Glib-based extensions

Yet another document, available separately, ties it all together:
  http://gtk2-perl.sourceforge.net/doc/binding_howto.pod.html
```

For gtk2−perl itself, see its website at

```
gtk2-perl - http://gtk2-perl.sourceforge.net/
```

A mailing list exists for discussion of using gtk2−perl and related modules. Archives and subscription information are available at http://lists.gnome.org/.

AUTHORS

muppet, <scott at asofyet dot org>, who borrowed heavily from the work of Göran Thyni, <gthyni at kirra dot net> and Guillaume Cottenceau <gc at mandrakesoft dot com> on the first gtk2−perl module, and from the sourcecode of the original gtk-perl and pygtk projects. Marc Lehmann <pcg at goof dot com> did lots of great work on the magic of making Glib::Object wrapper and subclassing work like they should. Ross McFarland <rwmcfa1 at neces dot com> wrote quite a bit of the documentation generation tools. Torsten Schoenfeld <kaffeetisch at web dot de> contributed little patches and tests here and there.

COPYRIGHT AND LICENSE