NAME

Glib::Type − Utilities for dealing with the GLib Type system

DESCRIPTION

This package defines several utilities for dealing with the GLib type system from Perl. Because of some fundamental differences in how the GLib and Perl type systems work, a fair amount of the binding magic leaks out, and you can find most of that in the `Glib::Type::register*` functions, which register new types with the GLib type system.

Most of the rest of the functions provide introspection functionality, such as listing properties and values and other cool stuff that is used mainly by Glib's reference documentation generator (see Glib::GenPod).

METHODS

list = Glib::Type−>list_ancestors ($package)

- `$package` (string)

List the ancestry of *package*, as seen by the GLib type system. The important difference is that GLib's type system implements only single inheritance, whereas Perl's `@ISA` allows multiple inheritance.

This returns the package names of the ancestral types in reverse order, with the root of the tree at the end of the list.

See also list_interfaces ().

list = Glib::Type−>list_interfaces ($package)

- `$package` (string)

List the GInterfaces implemented by the type associated with *package*. The interfaces are returned as package names.

list = Glib::Type−>list_signals ($package)

- `$package` (string)

List the signals associated with *package*. This lists only the signals for*pac kage*, not any of its parents. The signals are returned as a list of anonymous hashes which mirror the GSignalQuery structure defined in the C API reference.

− signal_id
: Numeric id of a signal. It's rare that you'll need this in Gtk2−Perl.

− signal_name
: Name of the signal, such as what you'd pass to `signal_connect`.

− itype
: The *i*nstance *type* for which this signal is defined.

− signal_flags
: GSignalFlags describing this signal.

− return_type
: The return type expected from handlers for this signal. If undef or not present, then no return is expected. The type name is mapped to the corresponding Perl package name if it is known, otherwise you get the raw C name straight from GLib.

− param_types
: The types of the parameters passed to any callbacks connected to the emission of this signal. The list does not include the instance, which is always first, and the user data from `signal_connect`, which is always last (unless the signal was connected with ''swap'', which swaps the instance and the data, but you get the point).

list = Glib::Type−>list_values ($package)

- `$package` (string)

List the legal values for the GEnum or GFlags type *`$package`*. If*`$package`* is not a package name registered with the bindings, this name is passed on to **g_type_from_name()** to see if it's a registered flags

or enum type that just hasn't been registered with the bindings by `gperl_register_fundamental()` (see Glib::xsapi). If *`$package`* is not the name of an enum or flags type, this function will croak.

Returns the values as a list of hashes, one hash for each value, containing the value, name and nickname, eg. for Glib::SignalFlags

```
{ value => 8,
  name  => 'G_SIGNAL_NO_RECURSE',
  nick  => 'no-recurse'
}
```

string = Glib::Type->package_from_cname ($cname)

- `$cname` (string)

Convert a C type name to the corresponding Perl package name. If no package is registered to that type, returns *`$cname`*.

Glib::Type->register ($parent_class, `$new_class`, ...)

- `$parent_class` (package) type from which to derive
- `$new_class` (package) name of new type
- ... (list) arguments for creation

Register a new type with the GLib type system.

This is a traffic-cop function. If *`$parent_type`* derives from Glib::Object, this passes the arguments through to `register_object`. If*`$parent_type`* is Glib::Flags or Glib::Enum, this strips *`$parent_type`* and passes the remaining args on to `register_enum` or `register_flags`. See those functions' documentation for more information.

Glib::Type->register_enum ($name, ...)

- `$name` (string) package name for new enum type
- ... (list) new enum's values; see description.

Register and initialize a new Glib::Enum type with the provided ''values''. This creates a type properly registered GLib so that it can be used for property and signal parameter or return types created with `Glib::Type->register` or `Glib::Object::Subclass`.

The list of values is used to create the ''nicknames'' that are used in general Perl code; the actual numeric values used at the C level are automatically assigned, starting with 1. If you need to specify a particular numeric value for a nick, use an array reference containing the nickname and the numeric value, instead. You may mix and match the two styles.

```
Glib::Type->register_enum ('MyFoo::Bar',
        'value-one',            # assigned 1
        'value-two',            # assigned 2
        ['value-three' => 15 ], # explicit 15
        ['value-four' => 35 ],  # explicit 35
        'value-five',           # assigned 5
);
```

If you use the array-ref form, beware: the code performs no validation for unique values.

Glib::Type->register_flags ($name, ...)

- `$name` (string) package name of new flags type
- ... (list) flag values, see discussion.

Register and initialize a new Glib::Flags type with the provided ''values''. This creates a type properly registered GLib so that it can be used for property and signal parameter or return types created with `Glib::Type->register` or `Glib::Object::Subclass`.

The list of values is used to create the ''nicknames'' that are used in general Perl code; the actual numeric

values used at the C level are automatically assigned, of the form 1<<i, starting with i = 0. If you need to specify a particular numeric value for a nick, use an array reference containing the nickname and the numeric value, instead. You may mix and match the two styles.

```
Glib::Type->register_flags ('MyFoo::Baz',
        'value-one',               # assigned 1<<0
        'value-two',               # assigned 1<<1
        ['value-three' => 1<<10 ], # explicit 1<<10
        ['value-four' => 0x0f ],   # explicit 0x0f
        'value-five',              # assigned 1<<4
);
```

If you use the array-ref form, beware: the code performs no validation for unique values.

Glib::Type->register_object ($parent_package, $new_package, ...)

- `$parent_package` (string) name of the parent package, which must be a derivative of Glib::Object.
- `$new_package` (string) usually __PACKAGE__.
- ... (list) key/value pairs controlling how the class is created.

Register *new_package* as an officially GLib-sanctioned derivative of the (GObject derivative) *parent_package*. This automatically sets up an`@ISA` entry for you, and creates a ne w GObjectClass under the hood.

The ... parameters are key/value pairs, currently supporting:

signals => HASHREF

The `signals` key contains a hash, keyed by signal names, which describes how to set up the signals for *new_package*.

If the value is a code reference, the named signal must exist somewhere in *parent_package* or its ancestry; the code reference will be used to override the class closure for that signal. This is the officially sanctioned way to override virtual methods on Glib::Objects. The value may be a string rather than a code reference, in which case the sub with that name in *new_package* will be used. (The function should not be inherited.)

If the value is a hash reference, the key will be the name of a new signal created with the properties defined in the hash. All of the properties are optional, with defaults provided:

class_closure => subroutine or undef

Use this code reference (or sub name) as the class closure (that is, the default handler for the signal). If not specified, "do_*signal_name*", in the current package, is used.

return_type => package name or undef

Return type for the signal. If not specified, then the signal has void return.

param_types => ARRAYREF

Reference to a list of parameter types (package names), *omitting the instance and user data*. Callbacks connected to this signal will receive the instance object as the first argument, followed by arguments with the types listed here, and finally by any user data that was supplied when the callback was connected. Not specifying this key is equivalent to supplying an empty list, which actually means instance and maybe data.

flags => Glib::SignalFlags

Flags describing this signal's properties. See the GObject C API reference' description of GSignalFlags for a complete description.

accumulator => subroutine or undef

The signal accumulator is a special callback that can be used to collect return values of the various callbacks that are called during a signal emission. Generally, you can omit this parameter; custom accumulators are used to do things like stopping signal propagation by return

value or creating a list of returns, etc. See "SIGNALS" in Glib::Object::Subclass for details.

properties => ARRAYREF

Array of Glib::ParamSpec objects, each describing an object property to add to the new type. These properties are available for use by all code that can access the object, regardless of implementation language. See Glib::ParamSpec. This list may be empty; if it is not, the functions`GET_PROPERTY` and `SET_PROPERTY` in *`$new_package`* will be called to get and set the values. Note that an object property is just a mechanism for getting and setting a value — it implies no storage. As a convenience, however, Glib::Object provides fallbacks for GET_PROPERTY and SET_PROPERTY which use the property nicknames as hash keys in the object variable for storage.

Additionally, you may specify ParamSpecs as a describing hash instead of as an object; this form allows you to supply explicit getter and setter methods which override GET_PROPERY and SET_PROPERTY. The getter and setter are both optional in the hash form. For example:

```
Glib::Type->register_object ('Glib::Object', 'Foo',
   properties => [
      # specified normally
      Glib::ParamSpec->string (...),
      # specified explicitly
      {
         pspec => Glib::ParamSpec->int (...),
         set => sub {
            my ($object, $newval) = @_;
            ...
         },
         get => sub {
            my ($object) = @_;
            ...
            return $val;
         },
      },
   ]
);
```

You can mix the two declaration styles as you like. If you have individual `get_foo` / `set_foo` methods with the operative code for a property then the `get`/`set` form is a handy way to go straight to that.

interfaces => ARRAYREF

Array of interface package names that the new object implements. Interfaces are the GObject way of doing multiple inheritance, thus, in Perl, the package names will be prepended to `@ISA` and certain inheritable and overrideable ALLCAPS methods will automatically be called whenever needed. Which methods exactly depends on the interface — Gtk2::CellEditable for example uses START_EDITING, EDITING_DONE, and REMOVE_WIDGET.

SEE ALSO

Glib

COPYRIGHT