

**NAME**

Glib::CodeGen – code generation utilities for Glib-based bindings.

**SYNOPSIS**

```
# usually in Makefile.PL
use Glib::CodeGen;

# most common, use all defaults
Glib::CodeGen->parse_maps ('myprefix');
Glib::CodeGen->write_boot;

# more exotic, change everything
Glib::CodeGen->parse_maps ('foo',
                           input => 'foo.maps',
                           header => 'foo-autogen.h',
                           typemap => 'foo.typemap',
                           register => 'register-foo.xsh');
Glib::CodeGen->write_boot (filename => 'bootfoo.xsh',
                           glob => 'Foo*.xs',
                           ignore => '^(Foo|Foo::Bar)$');

# add a custom type handler (rarely necessary)
Glib::CodeGen->add_type_handler (FooType => \&gen_foo_stuff);
# (see the section EXTENDING TYPE SUPPORT for more info.)
```

**DESCRIPTION**

This module packages some of the boilerplate code needed for performing code generation typically used by perl bindings for object-based libraries, using the Glib module as a base.

The default output filenames are in the subdirectory 'build', which usually will be present if you are using ExtUtils::Depends (as most Glib-based extensions probably should).

**METHODS**

Glib::CodeGen->write\_boot;

Glib::CodeGen->write\_boot (KEY => VAL, ...)

Many GObject-based libraries to be bound to perl will be too large to put in a single XS file; however, a single PM file typically only bootstraps one XS file's code. `write_boot` generates an XSH file to be included from the BOOT section of that one bootstrapped module, calling the boot code for all the other XS files in the project.

Options are passed to the function in a set of key/val pairs, and all options may default.

filename	the name of the output file to be created. the default is 'build/boot.xsh'.
glob	a glob pattern that specifies the names of the xs files to scan for MODULE lines. the default is 'xs/*.xs'.
xs_files	use this to supply an explicit list of file names (as an array reference) to use instead of a glob pattern. the default is to use the glob pattern.
ignore	regular expression matching any and all module names which should be ignored, i.e. NOT included in the list of symbols to boot. this parameter is extremely important for

avoiding infinite loops at startup; see the discussion for an explanation and rationale. the default is `'^[^:]+$'`, or, any name that contains no colons, i.e., any toplevel package name.

This function performs a glob (using perl's builtin glob operator) on the pattern specified by the `'glob'` option to retrieve a list of file names. It then scans each file in that list for lines matching the pattern `"^MODULE"` — that is, the `MODULE` directive in an XS file. The module name is pulled out and matched against the regular expression specified by the `ignore` parameter. If this module is not to be ignored, we next check to see if the name has been seen. If not, the name will be converted to a boot symbol (basically, `s/_/_/` and prepend `"boot_"`) and this symbol will be added to a call to `GPERRL_CALL_BOOT` in the generated file; it is then marked as seen so we don't call it again.

What is this all about, you ask? In order to bind an XSub to perl, the C function must be registered with the interpreter. This is the function of the `"boot"` code, which is typically called in the bootstrapping process. However, when multiple XS files are used with only one PM file, some other mechanism must call the boot code from each XS file before any of the function therein will be available.

A typical setup for a multiple-XS, single-PM module will be to call the various bits of boot code from the `BOOT:` section of the toplevel module's XS file.

To use `Gtk2` as an example, when you do `'use Gtk2'`, `Gtk2.pm` calls `bootstrap` on `Gtk2`, which calls the C function `boot_Gtk2`. This function calls the boot symbols for all the other xs files in the module. The distinction is that the toplevel module, `Gtk2`, has no colons in its name.

`xsubpp` generates the boot function's name by replacing the colons in the `MODULE` name with underscores and prepending `"boot_"`. We need to be careful not to include the boot code for the bootstrapped module, (say `Toplevel`, or `Gtk2`, or whatever) because the bootstrap code in `Toplevel.pm` will call `boot_Toplevel` when loaded, and `boot_Toplevel` should actually include the file we are creating here.

The default value for the `ignore` parameter ignores any name not containing colons, because it is assumed that this will be a toplevel module, and any other packages/modules it boots will be *below* this namespace, i.e., they will contain colons. This assumption holds true for `Gtk2` and `Gnome2`, but obviously fails for something like `Gnome2::Canvas`. To boot that module properly, you must use a regular expression such as `"^Gnome2::Canvas$"`.

Note that you can, of course, match more than just one name, e.g. `"^(Foo|Foo::Bar)$"`, if you wanted to have `Foo::Bar` be included in the same dynamically loaded object but only be booted when absolutely necessary. (If you get that to work, more power to you.)

Also, since this code scans for `^MODULE`, you must comment the `MODULE` section out with leading `#` marks if you want to hide it from `write_boot`.

Glib::CodeGen->parse\_maps (PREFIX, [KEY => VAL, ...])

Convention within `Glib/Gtk2` and friends is to use preprocessor macros in the style of `SvMyType` and `newSVMMyType` to get values in and out of perl, and to use those same macros from both hand-written code as well as the `typemaps`. However, if you have a lot of types in your library (such as the nearly 200 types in `Gtk+ 2.x`), then writing those macros becomes incredibly tedious, especially so when you factor in all of the variants and such.

So, this function can turn a flat file containing terse descriptions of the types into a header containing all the cast macros, a `typemap` file using them, and an `XSH` file containing the proper code to register each of those types (to be included by your module's `BOOT` code).

The `PREFIX` is mandatory, and is used in some of the resulting filenames. You can also override the defaults by providing `key=>val` pairs:

```

input      input file name.  default is 'maps'.  if this
           key's value is an array reference, all the
           filenames in the array will be scanned.
header     name of the header file to create, default is
           build/$prefix-autogen.h
typemap    name of the typemap file to create, default is
           build/$prefix.typemap
register   name of the xsh file to contain all of the
           type registrations, default is build/register.xsh

```

the maps file is a table of type descriptions, one per line, with fields separated by whitespace. the fields should be:

```

TYPE macro    e.g., GTK_TYPE_WIDGET
class name    e.g. GtkWidget, name of the C type
base type     one of GObject, GBoxed, GEnum, GFlags.
              To support other base types, see
              EXTENDING TYPE SUPPORT for info on
              on how to add a custom type handler.
package       name of the perl package to which this
              class name should be mapped, e.g.
              Gtk2::Widget

```

As a special case, you can also use this same format to register error domains; in this case two of the four columns take on slightly different meanings:

```

domain macro    e.g., GDK_PIXBUF_ERROR
enum type macro e.g., GDK_TYPE_PIXBUF_ERROR
base type       GError
package         name of the Perl package to which this
                class name should be mapped, e.g.,
                Gtk2::Gdk::Pixbuf::Error.

```

## EXTENDING TYPE SUPPORT

`parse_maps` uses the base type entry in each maps record to decide how to generate output for that type. In the base module, type support is included for the base types provided by Glib. It is easy to add support for your own types, by merely adding a type handler. This type handler will call utility functions to add typemaps, BOOT lines, and header lines.

```
Glib::CodeGen->add_type_handler($base_type => $handler)
$base_type (string) C name of the base type to handle.
$handler (subroutine) Callback used to handle this type.
```

Use *\$handler* to generate output for records whose base type is *\$base\_type*. *\$base\_type* is the C type name as found in the third column of a maps file entry.

*\$handler* will be called with the (possibly preprocessed) contents of the current maps file record, and should call the `add_typemap`, `add_register`, and `add_header` functions to set up the necessary C/XS glue for that type.

For example:

```

Glib::CodeGen->add_type_handler (CoolThing => sub {
    my ($typemacro, $classname, $base, $package) = @_;

    # $typemacro is the C type macro, like COOL_TYPE_THING.
    # $classname is the actual C type name, like CoolFooThing.
    # $base is the C name of the base type.  If CoolFooThing
    #     isa CoolThing, $base will be CoolThing.  This
    #     parameter is useful when using the same type handler

```

```
#      for multiple base types.
# $package is the package name that corresponds to
#      $classname, as specified in the maps file.
```

```
...
});
```

`add_typemap $type, $typemap[, $input, $output]`

Add a typemap entry for `$type`, named `$typemap`. If `$input` and/or `$output` are defined, their text will be used as the INPUT and/or OUTPUT typemap implementations (respectively) for `$typemap`. Note that in general, you'll use `T_GPERL_GENERIC_WRAPPER` or some other existing typemap for `$typemap`, so `$input` and `$output` are very rarely used.

Example:

```
# map $classname pointers and all their variants to the generic
# wrapper typemap.
add_typemap "$classname *", "T_GPERL_GENERIC_WRAPPER";
add_typemap "const $classname *", "T_GPERL_GENERIC_WRAPPER";
add_typemap "$classname\_ornull *", "T_GPERL_GENERIC_WRAPPER";
add_typemap "const $classname\_ornull *", "T_GPERL_GENERIC_WRAPPER";
add_typemap "$classname\_own *", "T_GPERL_GENERIC_WRAPPER";
add_typemap "$classname\_copy *", "T_GPERL_GENERIC_WRAPPER";
add_typemap "$classname\_own_ornull *", "T_GPERL_GENERIC_WRAPPER";

# custom code for an int-like enum:
add_typemap $class => T_FOO,
    "\$var = foo_unwrap (\$arg);", # input
    "\$arg = foo_wrap (\$var);"; # output
```

`add_register $text`

Add `$text` to the generated `register.xsh`. This is usually used for registering types with the bindings, e.g.:

```
add_register "#ifdef $typemacro\n"
    . "gperl_register_object ($typemacro, \"$package\");\n"
    . "#endif /* $typemacro */";
```

`add_header $text`

Add `$text` to the generated C header. You'll put variant typedefs and wrap/unwrap macros in the header, and will usually want to wrap the declarations in `#ifdef $typemacro` for safety.

## BUGS

GInterfaces are mostly just ignored.

The code is ugly.

## AUTHOR

muppet <scott at asofyet dot org>

## COPYRIGHT

Copyright (C) 2003–2005, 2013 by the gtk2–perl team (see the file AUTHORS for the full list)

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Library General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Library General Public License for more details.

You should have received a copy of the GNU Library General Public License along with this library; if not,

write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110–1301 USA.