NAME

glib-mkenums - C language enum description generation utility

SYNOPSIS

glib-mkenums [OPTION...] [FILE...]

DESCRIPTION

glib-mkenums is a small utility that parses C code to extract enum definitions and produces enum descriptions based on text templates specified by the user. Typically, you can use this tool to generate enumeration types for the GType type system, for GObject properties and signal marshalling; additionally, you can use it to generate enumeration values of GSettings schemas.

glib-mkenums takes a list of valid C code files as input. The options specified control the text that generated, substituting various keywords enclosed in @ characters in the templates.

Since version 2.74, GLib provides the G_DEFINE_ENUM_TYPE and G_DEFINE_FLAGS_TYPE C pre-processor macros. These macros can be used to define a GType for projects that have few, small enumeration types without going through the complexities of generating code at build time.

Production text substitutions

Certain keywords enclosed in @ characters will be substituted in the emitted text. For the substitution examples of the keywords below, the following example enum definition is assumed:

```
typedef enum
{
  PREFIX_THE_XVALUE    = 1 << 3,
  PREFIX_ANOTHER_VALUE = 1 << 4
} PrefixTheXEnum;
```

@EnumName@

The name of the enum currently being processed, enum names are assumed to be properly namespaced and to use mixed capitalization to separate words (e.g. PrefixTheXEnum).

@enum_name@

The enum name with words lowercase and word-separated by underscores (e.g. prefix_the_xenum).

@ENUMNAME@

The enum name with words uppercase and word-separated by underscores (e.g. PREFIX_THE_XENUM).

@ENUMSHORT@
: The enum name with words uppercase and word-separated by underscores, prefix stripped (e.g. THE_XENUM).

@ENUMPREFIX@
: The prefix of the enum name (e.g. PREFIX).

@VALUENAME@
: The enum value name currently being processed with words uppercase and word-separated by underscores, this is the assumed literal notation of enum values in the C sources (e.g. PREFIX_THE_XVALUE).

@valuenick@
: A nick name for the enum value currently being processed, this is usually generated by stripping common prefix words of all the enum values of the current enum, the words are lowercase and underscores are substituted by a minus (e.g. the-xvalue).

@valuenum@
: The integer value for the enum value currently being processed. If the evaluation fails then **glib-mkenums** will exit with an error status, but this only happens if @valuenum@ appears in your value production template. (Since: 2.26)

@type@
: This is substituted either by "enum" or "flags", depending on whether the enum value definitions contained bit-shift operators or not (e.g. flags).

@Type@
: The same as @type@ with the first letter capitalized (e.g. Flags).

@TYPE@
: The same as @type@ with all letters uppercased (e.g. FLAGS).

@filename@
: The full path of the input file currently being processed (e.g. /build/environment/project/src/foo.h).

@basename@
: The base name of the input file currently being processed (e.g. foo.h). Typically you want to use @basename@ in place of @filename@ in your templates, to improve the reproducibility of the build. (Since: 2.22)

Trigraph extensions

Some C comments are treated specially in the parsed enum definitions, such comments start out with the trigraph sequence /*< and end with the trigraph sequence >*/.

The following options can be specified per enum definition:

skip
: Indicates this enum definition should be skipped.

flags
: Indicates this enum should be treated as a flags definition.

underscore_name
: Specifies the word separation used in the ***_get_type()** function. For instance, /*< underscore_name=gnome_vfs_uri_hide_options >*/.

since
: Specifies the version tag that will be used to substitute the @enumsince@ keyword in the template, useful when documenting methods generated from the enums (e.g. Since: @enumsince@). (Since: 2.66)

The following options can be specified per value definition:

skip
: Indicates the value should be skipped.

nick
: Specifies the otherwise auto-generated nickname.

Examples:

```
typedef enum /*< skip >*/
{
  PREFIX_FOO
} PrefixThisEnumWillBeSkipped;
typedef enum /*< flags,prefix=PREFIX,since=1.0 >*/
{
  PREFIX_THE_ZEROTH_VALUE,      /*< skip >*/
  PREFIX_THE_FIRST_VALUE,
  PREFIX_THE_SECOND_VALUE,
```

```
  PREFIX_THE_THIRD_VALUE,       /*< nick=the-last-value >*/
} PrefixTheFlagsEnum;
```

OPTIONS

--fhead *TEXT*

Emits *TEXT* prior to processing input files.

You can specify this option multiple times, and the *TEXT* will be concatenated.

When used along with a template file, *TEXT* will be prepended to the template's file-header section.

--fprod *TEXT*

Emits *TEXT* every time a new input file is being processed.

You can specify this option multiple times, and the *TEXT* will be concatenated.

When used along with a template file, *TEXT* will be appended to the template's file-production section.

--ftail *TEXT*

Emits *TEXT* after all input files have been processed.

You can specify this option multiple times, and the *TEXT* will be concatenated.

When used along with a template file, *TEXT* will be appended to the template's file-tail section.

--eprod *TEXT*

Emits *TEXT* every time an enum is encountered in the input files.

--vhead *TEXT*

Emits *TEXT* before iterating over the set of values of an enum.

You can specify this option multiple times, and the *TEXT* will be concatenated.

When used along with a template file, *TEXT* will be prepended to the template's value-header section.

--vprod *TEXT*

Emits *TEXT* for every value of an enum.

You can specify this option multiple times, and the *TEXT* will be concatenated.

When used along with a template file, *TEXT* will be appended to the template's value-production section.

--vtail *TEXT*
Emits *TEXT* after iterating over all values of an enum.

You can specify this option multiple times, and the *TEXT* will be concatenated.

When used along with a template file, *TEXT* will be appended to the template's value-tail section.

--comments *TEXT*
Template for auto-generated comments, the default (for C code generations) is "/* @comment@ */".

--template *FILE*
Read templates from the given file. The templates are enclosed in specially-formatted C comments:

```
/*** BEGIN section ***/
/*** END section ***/
```

section may be file-header, file-production, file-tail, enumeration-production, value-header, value-production, value-tail or comment.

--identifier-prefix *PREFIX*
Indicates what portion of the enum name should be interpreted as the prefix (eg, the "Gtk" in "GtkDirectionType"). Normally this will be figured out automatically, but you may need to override the default if your namespace is capitalized oddly.

--symbol-prefix *PREFIX*
Indicates what prefix should be used to correspond to the identifier prefix in related C function names (eg, the "gtk" in "gtk_direction_type_get_type". Equivalently, this is the lowercase version of the prefix component of the enum value names (eg, the "GTK" in "GTK_DIR_UP". The default value is the identifier prefix, converted to lowercase.

--help
Print brief help and exit.

--version

Print version and exit.

--output=FILE
Write output to FILE instead of stdout.

@RSPFILE
When passed as the sole argument, read and parse the actual arguments from RSPFILE. Useful on systems with a low command-line length limit. For example, Windows has a limit of 8191 characters.

USING TEMPLATES

Instead of passing the various sections of the generated file to the command line of **glib-mkenums**, it's strongly recommended to use a template file, especially for generating C sources.

A C header template file will typically look like this:

```
/*** BEGIN file-header ***/
#pragma once

/* Include the main project header */
#include "project.h"

G_BEGIN_DECLS
/*** END file-header ***/

/*** BEGIN file-production ***/

/* enumerations from "@basename@" */
/*** END file-production ***/

/*** BEGIN value-header ***/
GType @enum_name@_get_type (void) G_GNUC_CONST;
#define @ENUMPREFIX@_TYPE_@ENUMSHORT@ (@enum_name@_get_type ())
/*** END value-header ***/

/*** BEGIN file-tail ***/
G_END_DECLS
/*** END file-tail ***/
```

A C source template file will typically look like this:

```
/*** BEGIN file-header ***/
#include "config.h"
#include "enum-types.h"

/*** END file-header ***/

/*** BEGIN file-production ***/
/* enumerations from "@basename@" */
/*** END file-production ***/

/*** BEGIN value-header ***/
GType
@enum_name@_get_type (void)
{
  static gsize static_g_@type@_type_id;

  if (g_once_init_enter (&static_g_@type@_type_id))
    {
      static const G@Type@Value values[] = {
/*** END value-header ***/

/*** BEGIN value-production ***/
          { @VALUENAME@, "@VALUENAME@", "@valuenick@" },
/*** END value-production ***/

/*** BEGIN value-tail ***/
          { 0, NULL, NULL }
      };

      GType g_@type@_type_id =
        g_@type@_register_static (g_intern_static_string ("@EnumName@"), values);

      g_once_init_leave (&static_g_@type@_type_id, g_@type@_type_id);
    }
  return static_g_@type@_type_id;
}

/*** END value-tail ***/
```

Template files are easier to modify and update, and can be used to generate various types of outputs

using the same command line or tools during the build.

USING GLIB-MKENUMS WITH MESON

Meson supports generating enumeration types using **glib-mkenums** out of the box in its "gnome" module.

In your meson.build file you will typically call the gnome.mkenums_simple() method to generate idiomatic enumeration types from a list of headers to inspect:

```
project_headers = [
  'project-foo.h',
  'project-bar.h',
  'project-baz.h',
]

gnome = import('gnome')
enum_files = gnome.mkenums_simple('enum-types',
  sources: project_headers,
)
```

The enum_files variable will contain an array of two elements in the following order:

- build target for the source file

- build target for the header file

You should use the returned objects to provide a dependency on every other build target that references the source or header file; for instance, if you are using the source to build a library:

```
mainlib = library('project',
  sources: project_sources + enum_files,
  ...
)
```

Additionally, if you are including the generated header file inside a build target that depends on the library you just built, you must ensure that the internal dependency includes the generated header as a required source file:

```
mainlib_dep = declare_dependency(sources: enum_files[1], link_with: mainlib)
```

You should not include the generated source file as well, otherwise it will be built separately for every target that depends on it, causing build failures. To know more about why all this is required, please refer to the **corresponding Meson FAQ entry**[1].

If you are generating C header and source files that require special templates, you can use gnome.mkenums() to provide those headers, for instance:

```
enum_files = gnome.mkenums('enum-types',
  sources: project_headers,
  h_template: 'enum-types.h.in',
  c_template: 'enum-types.c.in',
  install_header: true,
)
```

For more information, see the **Meson documentation for gnome.mkenums()**[2].

USING GLIB-MKENUMS WITH AUTOTOOLS

In order to use **glib-mkenums** in your project when using Autotools as the build system, you will first need to modify your configure.ac file to ensure you find the appropriate command using **pkg-config**, similarly as to how you discover the compiler and linker flags for GLib.

```
PKG_PROG_PKG_CONFIG([0.28])

PKG_CHECK_VAR([GLIB_MKENUMS], [glib-2.0], [glib_mkenums])
```

In your Makefile.am file you will typically use rules like these:

```
# A list of headers to inspect
project_headers = \
        project-foo.h \
        project-bar.h \
        project-baz.h

enum-types.h: $(project_headers) enum-types.h.in
        $(AM_V_GEN)$(GLIB_MKENUMS) \
                --template=enum-types.h.in \
                --output=$@ \
                $(project_headers)
```

```
enum-types.c: $(project_headers) enum-types.c.in enum-types.h
	$(AM_V_GEN)$(GLIB_MKENUMS) \
		--template=enum-types.c.in \
		--output=$@ \
		$(project_headers)

# Build the enum types files before every other target
BUILT_SOURCES += enum-types.h enum-types.c
CLEANFILES += enum-types.h enum-types.c
EXTRA_DIST += enum-types.h.in enum-types.c.in
```

In the example above, we have a variable called project_headers where we reference all header files we want to inspect for generating enumeration GTypes. In the enum-types.h rule we use **glib-mkenums** with a template called enum-types.h.in in order to generate the header file; similarly, in the enum-types.c rule we use a template called enum-types.c.in.

SEE ALSO

glib-genmarshal(1)

NOTES

1. corresponding Meson FAQ entry
   https://mesonbuild.com/FAQ.html#how-do-i-tell-meson-that-my-sources-use-generated-headers

2. Meson documentation for gnome.mkenums()
   https://mesonbuild.com/Gnome-module.html#gnomegenmarshal