NAME

glib-genmarshal - C code marshaller generation utility for GLib closures

SYNOPSIS

glib-genmarshal [OPTION...] [FILE...]

DESCRIPTION

glib-genmarshal is a small utility that generates C code marshallers for callback functions of the GClosure mechanism in the GObject sublibrary of GLib. The marshaller functions have a standard signature, they get passed in the invoking closure, an array of value structures holding the callback function parameters and a value structure for the return value of the callback. The marshaller is then responsible to call the respective C code function of the closure with all the parameters on the stack and to collect its return value.

glib-genmarshal takes a list of marshallers to generate as input. The marshaller list is either read from files passed as additional arguments on the command line; or from standard input, by using - as the input file.

Marshaller list format

The marshaller lists are processed line by line, a line can contain a comment in the form of

or a marshaller specification of the form

RTYPE:*PTYPE*
RTYPE:*PTYPE*,*PTYPE*
RTYPE:*PTYPE*,*PTYPE*,*PTYPE*

The *RTYPE* part specifies the callback's return type and the *PTYPE*s right to the colon specify the callback's parameter list, except for the first and the last arguments which are always pointers.

Parameter types

Currently, the following types are supported:

VOID
: indicates no return type, or no extra parameters. If *VOID* is used as the parameter list, no additional parameters may be present.

BOOLEAN
: for boolean types (gboolean)

CHAR
for signed char types (gchar)

UCHAR
for unsigned char types (guchar)

INT
for signed integer types (gint)

UINT
for unsigned integer types (guint)

LONG
for signed long integer types (glong)

ULONG
for unsigned long integer types (gulong)

INT64
for signed 64bit integer types (gint64)

UINT64
for unsigned 64bit integer types (guint64)

ENUM
for enumeration types (gint)

FLAGS
for flag enumeration types (guint)

FLOAT
for single-precision float types (gfloat)

DOUBLE
for double-precision float types (gdouble)

STRING
for string types (gchar*)

BOXED

for boxed (anonymous but reference counted) types (GBoxed*)

PARAM
for GParamSpec or derived types (GParamSpec*)

POINTER
for anonymous pointer types (gpointer)

OBJECT
for GObject or derived types (GObject*)

VARIANT
for GVariant types (GVariant*)

NONE
deprecated alias for *VOID*

BOOL
deprecated alias for *BOOLEAN*

OPTIONS

--header
Generate header file contents of the marshallers. This option is mutually exclusive with the **--body** option.

--body
Generate C code file contents of the marshallers. This option is mutually exclusive with the **--header** option.

--prefix=*PREFIX*
Specify marshaller prefix. The default prefix is ‘g_cclosure_user_marshal’.

--skip-source
Skip source location remarks in generated comments.

--stdinc
Use the standard marshallers of the GObject library, and include glib-object.h in generated header files. This option is mutually exclusive with the **--nostdinc** option.

--nostdinc

Do not use the standard marshallers of the GObject library, and skip glib-object.h include directive in generated header files. This option is mutually exclusive with the **--stdinc** option.

--internal
Mark generated functions as internal, using G_GNUC_INTERNAL.

--valist-marshallers
Generate valist marshallers, for use with **g_signal_set_va_marshaller()**.

-v, **--version**
Print version information.

--g-fatal-warnings
Make warnings fatal, that is, exit immediately once a warning occurs.

-h, **--help**
Print brief help and exit.

-v, **--version**
Print version and exit.

--output=FILE
Write output to *FILE* instead of the standard output.

--prototypes
Generate function prototypes before the function definition in the C source file, in order to avoid a missing-prototypes compiler warning. This option is only useful when using the **--body** option.

--pragma-once
Use the once pragma instead of an old style header guard when generating the C header file. This option is only useful when using the **--header** option.

--include-header=*HEADER*
Adds a #include directive for the given file in the C source file. This option is only useful when using the **--body** option.

-D *SYMBOL[=VALUE]*
Adds a #define C pre-processor directive for *SYMBOL* and its given *VALUE*, or "1" if the value is unset. You can use this option multiple times; if you do, all the symbols will be defined in the same order given on the command line, before the symbols undefined using the **-U** option. This

option is only useful when using the **--body** option.

-U *SYMBOL*
Adds a #undef C pre-processor directive to undefine the given *SYMBOL*. You can use this option multiple times; if you do, all the symbols will be undefined in the same order given on the command line, after the symbols defined using the **-D** option. This option is only useful when using the **--body** option.

--quiet
Minimizes the output of **glib-genmarshal**, by printing only warnings and errors. This option is mutually exclusive with the **--verbose** option.

--verbose
Increases the verbosity of **glib-genmarshal**, by printing debugging information. This option is mutually exclusive with the **--quiet** option.

USING GLIB-GENMARSHAL WITH MESON

Meson supports generating closure marshallers using **glib-genmarshal** out of the box in its "gnome" module.

In your meson.build file you will typically call the gnome.genmarshal() method with the source list of marshallers to generate:

```
gnome = import('gnome')
marshal_files = gnome.genmarshal('marshal',
  sources: 'marshal.list',
  internal: true,
)
```

The marshal_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 + marshal_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: marshal_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].

For more information on how to use the method, see the **Meson documentation for gnome.genmarshal()**[2].

USING GLIB-GENMARSHAL WITH AUTOTOOLS

In order to use **glib-genmarshal** 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_GENMARSHAL], [glib-2.0], [glib_genmarshal])
```

In your Makefile.am file you will typically need very simple rules to generate the C files needed for the build.

```
marshal.h: marshal.list
	$(AM_V_GEN)$(GLIB_GENMARSHAL) \
		--header \
		--output=$@ \
		$<

marshal.c: marshal.list marshal.h
	$(AM_V_GEN)$(GLIB_GENMARSHAL) \
		--include-header=marshal.h \
		--body \
```

```
        --output=$@ \
        $<

BUILT_SOURCES += marshal.h marshal.c
CLEANFILES += marshal.h marshal.c
EXTRA_DIST += marshal.list
```

In the example above, the first rule generates the header file and depends on a marshal.list file in order to regenerate the result in case the marshallers list is updated. The second rule generates the source file for the same marshal.list, and includes the file generated by the header rule.

EXAMPLE

To generate marshallers for the following callback functions:

```
void   foo (gpointer data1,
            gpointer data2);
void   bar (gpointer data1,
            gint     param1,
            gpointer data2);
gfloat baz (gpointer data1,
            gboolean param1,
            guchar   param2,
            gpointer data2);
```

The marshaller.list file has to look like this:

```
VOID:VOID
VOID:INT
FLOAT:BOOLEAN,UCHAR
```

and you call glib-genmarshal like this:

```
glib-genmarshal --header marshaller.list > marshaller.h
glib-genmarshal --body marshaller.list > marshaller.c
```

The generated marshallers have the arguments encoded in their function name. For this particular list, they are

```
g_cclosure_user_marshal_VOID__VOID(...),
g_cclosure_user_marshal_VOID__INT(...),
```

g_cclosure_user_marshal_FLOAT__BOOLEAN_UCHAR(...).

They can be used directly for GClosures or be passed in as the GSignalCMarshaller c_marshaller; argument upon creation of signals:

```
GClosure *cc_foo, *cc_bar, *cc_baz;

cc_foo = g_cclosure_new (NULL, foo, NULL);
g_closure_set_marshal (cc_foo, g_cclosure_user_marshal_VOID__VOID);
cc_bar = g_cclosure_new (NULL, bar, NULL);
g_closure_set_marshal (cc_bar, g_cclosure_user_marshal_VOID__INT);
cc_baz = g_cclosure_new (NULL, baz, NULL);
g_closure_set_marshal (cc_baz, g_cclosure_user_marshal_FLOAT__BOOLEAN_UCHAR);
```

SEE ALSO

glib-mkenums(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.genmarshal()
   https://mesonbuild.com/Gnome-module.html#gnomegenmarshal