



Coding Conventions

Contents

2	Summary	2
3	Code formatting	3
4	Reformatting code	3
5	Memory management	4
6	Namespacing	4
7	Modularity	5
8	Pre- and post-condition assertions	6
9	GError usage	7
10	GList	8
11	Magic values	9
12	Asynchronous methods	10
13	Enumerated types and booleans	11
14	GObject properties	11
15	Resource leaks	11

This document specifically relates to software which is or has been created for the Apertis project. It is important that any code added to an existing project utilises the coding conventions as used by that project, maintaining consistency across that projects codebase.

Coding conventions is a nebulous topic, covering code formatting and whitespace, function and variable naming, namespacing, use of common GLib coding patterns, and other things. Since C is quite flexible, this document mostly consists of a series of patterns (which it's recommended code follows) and anti-patterns (which it's recommended code does **not** follow). Any approaches to coding which are not covered by a pattern or anti-pattern are completely valid.

Guidelines which are specific to GLib are included on this page; guidelines specific to other APIs are covered on their respective pages.

Summary

- **Align the happy path to the left edge** and when programming in the C language use the GLib coding style, with vim modelines.
- **Consistently namespace files**, functions and types.
- **Always design code to be modular**, encapsulated and loosely coupled.
 - Especially by keeping object member variables inside the object's private structure.
- Code defensively by **adding pre- and post-conditions assertions** to all public functions.
- Report all user errors (and no programmer errors) **using GError**.
- Use **appropriate container types** for sets of items.
- **Document all constant values** used in the code.
- Use standard GLib patterns for defining **asynchronous methods**.
- Do not call any blocking, **synchronous functions**.

- Do not run blocking operations in separate threads; [use asynchronous calls instead](#).
- [Prefer enumerated types over booleans](#) whenever there is the potential for ambiguity between true and false.
- Ensure [GObject properties](#) have no side-effects.
- [Treat resources as heap-allocated memory](#) and do not leak them.

Code formatting

Using a consistent code formatting style eases maintenance of code, by meaning contributors only have to learn one coding style for all modules, rather than one per module.

Regardless of the programming language, a good guideline for the organization of the control flow is [aligning the happy path to the left edge](#)¹.

The coding style in use is the popular [GLib coding style](#)², which is a slightly modified version of the [GNU coding style](#)³.

Each C and H file should have a vim-style modeline, which lets the programmer's editor know how code in the file should be formatted. This helps keep the coding style consistent as the files evolve. The following modeline should be put as the very first line of the file, immediately before the [copyright comment](#)⁴:

```
/* vim:set et sw=2 cin cino=t0,f0,(0,{s,>2s,n-s,^-s,e2s: */
```

For more information about the copyright comment, see [Applying Licensing](#)⁵.

Reformatting code

If a file or module does not conform to the code formatting style and needs to be reindented, the following command will do most of the work —but it can go wrong, and the file **must** be checked manually afterwards:

```
$ indent -gnu -hnl -nbbo -bbb -sob -bad -nut /path/to/file
```

To apply this to all C and H files in a module:

```
$ git ls-files '*.ch' | \
$ xargs indent -gnu -hnl -nbbo -bbb -sob -bad -nut
```

Alternatively, if you have a recent enough version of Clang (>3.5):

```
$ git ls-files '*.ch' | \
$ xargs clang-format -i -style=file
```

Using a `.clang-format` file (added to git) in the same directory, containing:

¹<https://medium.com/@matryer/line-of-sight-in-code-186dd7cdea88>

²<https://developer.gnome.org/programming-guidelines/unstable/c-coding-style.html.en>

³<http://www.gnu.org/prep/standards/standards.html#Writing-C>

⁴<https://www.apertis.org/guides/licensing/license-applying/#licensing-of-code>

⁵<https://www.apertis.org/guides/licensing/license-applying/>

```

74 # See https://www.apertis.org/policies/coding_conventions/#code-formatting
75 BasedOnStyle: GNU
76 AlwaysBreakAfterDefinitionReturnType: All
77 BreakBeforeBinaryOperators: None
78 BinPackParameters: false
79 SpaceAfterCStyleCast: true
80 # Our column limit is actually 80, but setting that results in clang-format
81 # making a lot of dubious hanging-indent choices; disable it and assume the
82 # developer will line wrap appropriately. clang-format will still check
83 # existing hanging indents.
84 ColumnLimit: 0

```

85 Memory management

86 See [Memory management](#)⁶ for some patterns on handling memory management;
87 particularly [single path cleanup](#)⁷.

88 Namespacing

89 Consistent and complete namespacing of symbols (functions and types) and files
90 is important for two key reasons:

- 91 1. Establishing a convention which means developers have to learn fewer
92 symbol names to use the library —they can guess them reliably instead.
- 93 2. Ensuring symbols from two projects do not conflict if included in the same
94 file.

95 The second point is important —imagine what would happen if every project
96 exported a function called `create_object()`. The headers defining them could
97 not be included in the same file, and even if that were overcome, the program-
98 mer would not know which project each function comes from. Namespacing
99 eliminates these problems by using a unique, consistent prefix for every symbol
100 and filename in a project, grouping symbols into their projects and separating
101 them from others.

102 The conventions below should be used for namespacing all symbols. They are
103 the [same as used in other GLib-based projects](#)⁸, so should be familiar to a lot
104 of developers:

- 105 • Functions should use `lower_case_with_underscores`.
- 106 • Structures, types and objects should use `CamelCaseWithoutUnderscores`.
- 107 • Macros and `#defines` should use `UPPER_CASE_WITH_UNDERSCORES`.
- 108 • All symbols should be prefixed with a short (2–4 characters) version of the
109 namespace.

⁶https://www.apertis.org/guides/app_devel/memory_management/

⁷https://www.apertis.org/guides/app_devel/memory_management/#Single-path_cleanup

⁸<https://developer.gnome.org/gobject/stable/gtype-conventions.html>

110 • All methods of an object should also be prefixed with the object name.

111 Additionally, public headers should be included from a subdirectory, effectively
112 namespacing the header files. For example, instead of `#include <abc.h>`, a project
113 should allow its users to use `#include <namespace/ns-abc.h>`

114 For example, for a project called ‘Walbottle’, the short namespace ‘Wbl’ would be
115 chosen. If it has a ‘schema’ object and a ‘writer’ object, it would install headers:

116 • `$PREFIX/include/walbottle-$API_MAJOR/walbottle/wbl-schema.h`
117 • `$PREFIX/include/walbottle-$API_MAJOR/walbottle/wbl-writer.h`

118 (The use of `$API_MAJOR` above is for [parallel installability](#)⁹.)

119 For the schema object, the following symbols would be exported (amongst oth-
120 ers), following GObject conventions:

121 • `WblSchema` structure
122 • `WblSchemaClass` structure
123 • `WBL_TYPE_SCHEMA` macro
124 • `WBL_IS_SCHEMA` macro
125 • `wbl_schema_get_type` function
126 • `wbl_schema_new` function
127 • `wbl_schema_load_from_data` function

128 Modularity

129 [Modularity](#)¹⁰, [encapsulation](#)¹¹ and [loose coupling](#)¹² are core computer science
130 concepts which are necessary for development of maintainable systems. Tightly
131 coupled systems require large amounts of effort to change, due to each change
132 affecting a multitude of other, seemingly unrelated pieces of code. Even for
133 smaller projects, good modularity is highly recommended, as these systems may
134 grow to be larger, and refactoring for modularity takes a lot of effort.

135 Assuming the general concepts of modularity, encapsulation and loose coupling
136 are well known, here are some guidelines for implementing them which are
137 specific to GLib and GObject APIs:

- 138 1. The private structure for a GObject should not be in any header files
139 (whether private or public). It should be in the C file defining the object,
140 as should all code which implements that structure and mutates it.
- 141 2. libtool convenience libraries should be used freely to allow internal
142 code to be used by multiple public libraries or binaries. However,
143 libtool convenience libraries must not be installed on the system. Use
144 `noinst_LTLIBRARIES` in `Makefile.am` to declare a convenience library; not
145 `lib_LTLIBRARIES`.

⁹https://www.apertis.org/guides/app_devel/module_setup/#Parallel_installability

¹⁰http://en.wikipedia.org/wiki/Modular_programming

¹¹http://en.wikipedia.org/wiki/Encapsulation_%28object-oriented_programming%29

¹²http://en.wikipedia.org/wiki/Loose_coupling

- 146 3. Restrict the symbols exported by public libraries by using `my_library_LDFLAGS`
147 `= -export-symbols my-library.symbols`, where `my-library.symbols` is a text
148 file listing the names of the functions to export, one per line. This
149 prevents internal or private functions from being exported, which would
150 break encapsulation. See [Exposing and Hiding Symbols](#)¹³.
151 4. Do not put any members (e.g. storage for object state or properties) in
152 a public GObject structure—they should all be encapsulated in a private
153 structure declared using `G_DEFINE_TYPE_WITH_PRIVATE`¹⁴.
154 5. Do not use static variables inside files or functions to preserve function
155 state between calls to it. Instead, store the state in an object (e.g. the
156 object the function is a method of) as a private member variable (in the
157 object's private structure). Using static variables means the state is shared
158 between all instances of the object, which is almost always undesirable, and
159 leads to confusing behaviour.

160 Pre- and post-condition assertions

161 An important part of secure coding is ensuring that incorrect data does not
162 propagate far through code—the further some malicious input can propagate,
163 the more code it sees, and the greater potential there is for an exploit to be
164 possible.

165 A standard way of preventing the propagation of invalid data is to check all
166 inputs to, and outputs from, all publicly visible functions in a library or module.
167 There are two levels of checking:

- 168 • Assertions: Check for programmer errors and abort the program on failure.
- 169 • Validation: Check for invalid input and return an error gracefully on fail-
170 ure.

171 Validation is a complex topic, and is handled using **GErrors**. The remainder of
172 this section discusses pre- and post-condition assertions, which are purely for
173 catching programmer errors. A programmer error is where a function is called
174 in a way which is documented as disallowed. For example, if `NULL` is passed to
175 a parameter which is documented as requiring a non-`NULL` value to be passed;
176 or if a negative value is passed to a function which requires a positive value.
177 Programmer errors can happen on output too—for example, returning `NULL`
178 when it is not documented to, or not setting a `GError` output when it fails.

179 Adding pre- and post-condition assertions to code is as much about ensuring
180 the behaviour of each function is correctly and completely documented as it is
181 about adding the assertions themselves. All assertions should be documented,
182 preferably by using the relevant [gobject-introspection annotations](#)¹⁵, such as
183 `(nullable)`.

¹³<https://autotools.io/libtool/symbols.html>

¹⁴<https://developer.gnome.org/gobject/stable/gobject-Type-Information.html#G-DEFINE-TYPE-WITH-PRIVATE:CAPS>

¹⁵<https://wiki.gnome.org/Projects/GObjectIntrospection/Annotations>

184 Pre- and post-condition assertions are implemented using `g_return_if_fail()`¹⁶
185 and `g_return_val_if_fail()`¹⁷.

186 The pre-conditions should check each parameter at the start of the function,
187 before any other code is executed (even retrieving the private data structure
188 from a GObject, for example, since the GObject pointer could be `NULL`). The
189 post-conditions should check the return value and any output parameters at the
190 end of the function —this requires a single return statement and use of `goto` to
191 merge other control paths into it. See [Single-path cleanup](#)¹⁸ for an example.

192 A fuller example is given in this [writeup of post-conditions](#)¹⁹.

193 GError usage

194 `GError`²⁰ is the standard error reporting mechanism for GLib-using code, and
195 can be thought of as a C implementation of an [exception](#)²¹.

196 Any kind of runtime failure (anything which is not a [programmer error](#)) must
197 be handled by including a `GError**` parameter in the function, and setting a
198 useful and relevant `GError` describing the failure, before returning from the
199 function. Programmer errors must not be handled using `GError`: use assertions,
200 pre-conditions or post-conditions instead.

201 `GError` should be used in preference to a simple return code, as it can con-
202 vey more information, and is also supported by all GLib tools. For example,
203 introspecting an API with [GObject introspection](#)²² will automatically detect
204 all `GError` parameters so that they can be converted to exceptions in other
205 languages.

206 Printing warnings to the console must not be done in library code: use a `GError`,
207 and the calling code can propagate it further upwards, decide to handle it, or
208 decide to print it to the console. Ideally, the only code which prints to the
209 console will be top-level application code, and not library code.

210 Any function call which can take a `GError**`, **should** take such a parameter, and
211 the returned `GError` should be handled appropriately. There are very few situ-
212 ations where ignoring a potential error by passing `NULL` to a `GError**` parameter
213 is acceptable.

214 The GLib API documentation contains a [full tutorial for using GError](#)²³.

¹⁶<https://developer.gnome.org/glib/stable/glib-Warnings-and-Assertions.html#g-return-if-fail>

¹⁷<https://developer.gnome.org/glib/stable/glib-Warnings-and-Assertions.html#g-return-val-if-fail>

¹⁸https://www.apertis.org/guides/app_devel/memory_management/#Single-path_cleanup

¹⁹<https://tecnocode.co.uk/2010/12/19/postconditions-in-c/>

²⁰<https://developer.gnome.org/glib/stable/glib-Error-Reporting.html>

²¹http://en.wikipedia.org/wiki/Exception_handling

²²<https://wiki.gnome.org/Projects/GObjectIntrospection>

²³<https://developer.gnome.org/glib/stable/glib-Error-Reporting.html#glib-Error->

215 GList

216 GLib provides several container types for sets of data:

- 217 • [GList](#)²⁴
- 218 • [GSList](#)²⁵
- 219 • [GPtrArray](#)²⁶
- 220 • [GArray](#)²⁷

221 It has been common practice in the past to use GList in all situations where
222 a sequence or set of data needs to be stored. This is inadvisable—in most
223 situations, a GPtrArray should be used instead. It has lower memory overhead
224 (a third to a half of an equivalent list), better cache locality, and the same
225 or lower algorithmic complexity for all common operations. The only typical
226 situation where a GList may be more appropriate is when dealing with ordered
227 data, which requires expensive insertions at arbitrary indexes in the array.

228 [Article on linked list performance](#)²⁸

229 If linked lists are used, be careful to keep the complexity of operations on
230 them low, using standard CS complexity analysis. Any operation which uses
231 [g_list_nth\(\)](#)²⁹ or [g_list_nth_data\(\)](#)³⁰ is almost certainly wrong. For example,
232 iteration over a GList should be implemented using the linking pointers, rather
233 than a incrementing index:

```
234 GList *some_list, *l;  
235  
236 for (l = some_list; l != NULL; l = l->next)  
237 {  
238     gpointer element_data = l->data;  
239  
240     /* Do something with @element_data. */  
241 }
```

242 Using an incrementing index instead results in an exponential decrease in per-
243 formance ($O(2 \times N^2)$ rather than $O(N)$):

```
244 GList *some_list;  
245 guint i;  
246  
247 /* This code is inefficient and should not be used in production. */  
248 for (i = 0; i < g_list_length (some_list); i++)
```

Reporting.description

²⁴<https://developer.gnome.org/glib/stable/glib-Doubly-Linked-Lists.html>

²⁵<https://developer.gnome.org/glib/stable/glib-Singly-Linked-Lists.html>

²⁶<https://developer.gnome.org/glib/stable/glib-Pointer-Arrays.html>

²⁷<https://developer.gnome.org/glib/stable/glib-Arrays.html>

²⁸<http://www.codeproject.com/Articles/340797/Number-crunching-Why-you-should-never-ever-EVER-us>

²⁹<https://developer.gnome.org/glib/2.30/glib-Doubly-Linked-Lists.html#g-list-nth>

³⁰<https://developer.gnome.org/glib/2.30/glib-Doubly-Linked-Lists.html#g-list-nth-data>


```

249     {
250         gpointer element_data = g_list_nth_data (some_list, i);
251
252         /* Do something with @element_data. */
253     }

```

254 The performance penalty comes from `g_list_length()` and `g_list_nth_data()`
255 which both traverse the list ($O(N)$) to perform their operations.

256 Implementing the above with a `GPtrArray` has the same complexity as the first
257 (correct) `GList` implementation, but better cache locality and lower memory
258 consumption, so will perform better for large numbers of elements:

```

259 GPtrArray *some_array;
260 guint i;
261
262 for (i = 0; i < some_array->len; i++)
263     {
264         gpointer element_data = some_array->pdata[i];
265
266         /* Do something with @element_data. */
267     }

```

268 Magic values

269 Do not use constant values in code without documenting them. These values
270 can be known as ‘magic’ values, because it is not clear how they were chosen,
271 what they depend on, or when they need to be updated.

272 Magic values should be:

- 273 • defined as macros using `#define`, rather than being copied to every usage
- 274 site;
- 275 • all defined in an easy-to-find location, such as the top of the source code
- 276 file; and
- 277 • documented, including information about how they were chosen, and what
- 278 that choice depended on.

279 One situation where magic values are used incorrectly is to circumvent the type
280 system. For example, a magic string value which indicates a special state for
281 a string variable. Magic values should not be used for this, as the software
282 state could then be corrupted if user input includes that string (for example).
283 Instead, a separate variable should be used to track the special state. Use the
284 type system to do this work for you —magic values should never be used as a
285 basic dynamic typing system.

Asynchronous methods

Long-running blocking operations should not be run such that they block the UI in a graphical application. This happens when one iteration of the UI's main loop takes significantly longer than the frame refresh rate, so the UI is not refreshed when the user expects it to be. Interactivity reduces and animations stutter. In extreme cases, the UI can freeze entirely until a blocking operation completes. This should be avoided at all costs.

Similarly, in non-graphical applications that respond to network requests or [D-Bus inter-process communication](#)³¹, blocking the main loop prevents the next request from being handled.

There are two possible approaches for preventing the main loop being blocked:

1. Running blocking operations asynchronously in the main thread, using polled I/O.
2. Running blocking operations in separate threads, with the main loop in the main thread.

The second approach (see [Threading](#)³²) typically leads to complex locking and synchronisation between threads, and introduces many bugs. The recommended approach in GLib applications is to use asynchronous operations, implemented using [GTask](#)³³ and [GAsyncResult](#)³⁴. Asynchronous operations must be implemented everywhere for this approach to work: any use of a blocking, synchronous operation will effectively make all calling functions blocking and synchronous too.

The documentation for [GTask](#)³⁵ and [GAsyncResult](#)³⁶ includes examples and tutorials for implementing and using GLib-style asynchronous functions.

Key principles for using them:

1. Never call synchronous methods: always use the `*_async()` and `*_finish()` variant methods.
2. Never use threads for blocking operations if an asynchronous alternative exists.
3. Always wait for an asynchronous operation to complete (i.e. for its `GAsyncReadyCallback` to be invoked) before starting operations which depend on it.
 - Never use a timeout (`g_timeout_add()`) to wait until an asynchronous operation 'should' complete. The time taken by an operation is unpredictable, and can be affected by other applications, kernel scheduling

³¹https://www.apertis.org/guides/app_devel/d-bus_services/

³²https://www.apertis.org/guides/app_devel/threading/

³³<https://developer.gnome.org/gio/stable/GTask.html>

³⁴<https://developer.gnome.org/gio/stable/GAsyncResult.html>

³⁵<https://developer.gnome.org/gio/stable/GTask.html>

³⁶<https://developer.gnome.org/gio/stable/GAsyncResult.html>

321 decisions, and various other system processes which cannot be pre-
322 dicted.

323 Enumerated types and booleans

324 In many cases, enumerated types should be used instead of booleans:

- 325 1. Booleans are not self-documenting in the same way as enums are. When
326 reading code it can be easy to misunderstand the sense of the boolean and
327 get things the wrong way round.
- 328 2. They are not extensible. If a new state is added to a property in future,
329 the boolean would have to be replaced —if an enum is used, a new value
330 simply has to be added to it.

331 This is documented well in the article [Use Enums Not Booleans](#)³⁷.

332 GObject properties

333 [Properties on GObject](#)³⁸ are a key feature of GLib-based object orientation.
334 Properties should be used to expose state variables of the object. A guiding
335 principle for the design of properties is that (in pseudo-code):

```
336 var temp = my_object.some_property  
337 my_object.some_property = "new value"  
338 my_object.some_property = temp
```

339 should leave `my_object` in exactly the same state as it was originally. Specifically,
340 properties should **not** act as parameterless methods, triggering state transitions
341 or other side-effects.

342 Resource leaks

343 As well as [memory leaks](#)³⁹, it is possible to leak resources such as GLib timeouts,
344 open file descriptors or connected GObject signal handlers. Any such resources
345 should be treated using the same principles as allocated memory.

346 For example, the source ID returned by `g_timeout_add()`⁴⁰ must always be stored
347 and removed (using `g_source_remove()`⁴¹) when the owning object is finalised.
348 This is because it is very rare that we can guarantee the object will live longer
349 than the timeout period —and if the object is finalised, the timeout left uncanceled,
350 and then the timeout triggers, the program will typically crash due to
351 accessing the object's memory after it's been freed.

³⁷<http://c2.com/cgi/wiki?UseEnumsNotBooleans>

³⁸<https://developer.gnome.org/gobject/stable/gobject-properties.html>

³⁹https://www.apertis.org/guides/app_devel/memory_management/

⁴⁰<https://developer.gnome.org/glib/stable/glib-The-Main-Event-Loop.html#g-timeout-add>

⁴¹<https://developer.gnome.org/glib/stable/glib-The-Main-Event-Loop.html#g-source-remove>

352 Similarly for signal connections, the signal handler ID returned by
 353 `g_signal_connect()`⁴² should always be saved and explicitly disconnected
 354 (`g_signal_handler_disconnect()`⁴³) unless the object being connected is guaran-
 355 teed to live longer than the object being connected to (the one which emits the
 356 signal):

357 Other resources which can be leaked, plus the functions acquiring and releasing
 358 them (this list is non-exhaustive):

- 359 • File descriptors (FDs):
 - 360 – `g_open()`⁴⁴
 - 361 – `g_close()`⁴⁵
- 362 • Threads:
 - 363 – `g_thread_new()`⁴⁶
 - 364 – `g_thread_join()`⁴⁷
- 365 • Subprocesses:
 - 366 – `g_spawn_async()`⁴⁸
 - 367 – `g_spawn_close_pid()`⁴⁹
- 368 • D-Bus name watches:
 - 369 – `g_bus_watch_name()`⁵⁰
 - 370 – `g_bus_unwatch_name()`⁵¹
- 371 • D-Bus name ownership:
 - 372 – `g_bus_own_name()`⁵²
 - 373 – `g_bus_unown_name()`⁵³

⁴²<https://developer.gnome.org/gobject/stable/gobject-Signals.html#g-signal-connect>

⁴³<https://developer.gnome.org/gobject/stable/gobject-Signals.html#g-signal-handler-disconnect>

⁴⁴<https://developer.gnome.org/glib/stable/glib-File-Utilities.html#g-open>

⁴⁵<https://developer.gnome.org/glib/stable/glib-File-Utilities.html#g-close>

⁴⁶<https://developer.gnome.org/glib/stable/glib-Threads.html#g-thread-new>

⁴⁷<https://developer.gnome.org/glib/stable/glib-Threads.html#g-thread-join>

⁴⁸<https://developer.gnome.org/glib/stable/glib-Spawning-Processes.html#g-spawn-async>

⁴⁹<https://developer.gnome.org/glib/stable/glib-Spawning-Processes.html#g-spawn-close-pid>

⁵⁰<https://developer.gnome.org/gio/stable/gio-Watching-Bus-Names.html#g-bus-watch-name>

⁵¹<https://developer.gnome.org/gio/stable/gio-Watching-Bus-Names.html#g-bus-unwatch-name>

⁵²<https://developer.gnome.org/gio/stable/gio-Owning-Bus-Names.html#g-bus-own-name>

⁵³<https://developer.gnome.org/gio/stable/gio-Owning-Bus-Names.html#g-bus-unown-name>