



Coding Conventions

# 1 Contents

2	Summary . . . . .	2
3	Code formatting . . . . .	2
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 . . . . .	7
11	Magic values . . . . .	9
12	Asynchronous methods . . . . .	9
13	Enumerated types and booleans . . . . .	10
14	GObject properties . . . . .	11
15	Resource leaks . . . . .	11

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

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

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

## 28 Summary

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

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

## 48 Code formatting

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

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

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

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

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

61 For more information about the copyright comment, see [Applying Licensing](#)<sup>5</sup>.

## 62 Reformatting code

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

```
66 $ indent -gnu -hnl -nbbo -bbb -sob -bad -nut /path/to/file
```

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

```
68 $ git ls-files '*.ch' | \  

  69 $ xargs indent -gnu -hnl -nbbo -bbb -sob -bad -nut
```

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

```
71 $ git ls-files '*.ch' | \  

  72 $ xargs clang-format -i -style=file
```

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

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

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

<sup>4</sup><https://jwd.pages.apertis.org/apertis-website/policies/license-applying/#licensing-of-code>

<sup>5</sup><https://jwd.pages.apertis.org/apertis-website/policies/license-applying/>

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

```
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](#)<sup>6</sup> for some patterns on handling memory management;  
87 particularly [single path cleanup](#)<sup>7</sup>.

## 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](#)<sup>8</sup>, 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`.

---

<sup>6</sup>[https://jwd.pages.apertis.org/apertis-website/guides/memory\\_management/](https://jwd.pages.apertis.org/apertis-website/guides/memory_management/)

<sup>7</sup>[https://jwd.pages.apertis.org/apertis-website/guides/memory\\_management/#Single-path\\_cleanup](https://jwd.pages.apertis.org/apertis-website/guides/memory_management/#Single-path_cleanup)

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

- 108 • All symbols should be prefixed with a short (2–4 characters) version of
- 109 the namespace.
- 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  
115 be chosen. If it has a ‘schema’ object and a ‘writer’ object, it would install  
116 headers:

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

119 (The use of `$API_MAJOR` above is for [parallel installability](#)<sup>9</sup>.)

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

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

## 129 Modularity

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

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

- 139 1. The private structure for a GObject should not be in any header files  
140 (whether private or public). It should be in the C file defining the object,  
141 as should all code which implements that structure and mutates it.

---

<sup>9</sup>[https://jwd.pages.apertis.org/apertis-website/guides/module\\_setup/#Parallel\\_installability](https://jwd.pages.apertis.org/apertis-website/guides/module_setup/#Parallel_installability)

<sup>10</sup>[http://en.wikipedia.org/wiki/Modular\\_programming](http://en.wikipedia.org/wiki/Modular_programming)

<sup>11</sup>[http://en.wikipedia.org/wiki/Encapsulation\\_%28object-oriented\\_programming%29](http://en.wikipedia.org/wiki/Encapsulation_%28object-oriented_programming%29)

<sup>12</sup>[http://en.wikipedia.org/wiki/Loose\\_coupling](http://en.wikipedia.org/wiki/Loose_coupling)

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

## 161 Pre- and post-condition assertions

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

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

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

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

---

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

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

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

186 Pre- and post-condition assertions are implemented using `g_return_if_fail()`<sup>16</sup>  
187 and `g_return_val_if_fail()`<sup>17</sup>.

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

194 A fuller example is given in this [writeup of post-conditions](#)<sup>19</sup>.

## 195 GError usage

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

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

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

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

---

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

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

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

<sup>18</sup>[https://jwd.pages.apertis.org/apertis-website/guides/memory\\_management/#Single-path\\_cleanup](https://jwd.pages.apertis.org/apertis-website/guides/memory_management/#Single-path_cleanup)

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

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

<sup>21</sup>[http://en.wikipedia.org/wiki/Exception\\_handling](http://en.wikipedia.org/wiki/Exception_handling)

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

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

216 The GLib API documentation contains a [full tutorial for using `GError`](#)<sup>23</sup>.

## 217 GList

218 GLib provides several container types for sets of data:

- 219 • [GList](#)<sup>24</sup>
- 220 • [GSLList](#)<sup>25</sup>
- 221 • [GPtrArray](#)<sup>26</sup>
- 222 • [GArray](#)<sup>27</sup>

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

230 [Article on linked list performance](#)<sup>28</sup>

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

```
236 GList *some_list, *l;  
237  
238 for (l = some_list; l != NULL; l = l->next)  
239 {  
240     gpointer element_data = l->data;  
241  
242     /* Do something with @element_data. */  
243 }
```

---

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

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

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

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

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

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

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

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



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

```
246 GList *some_list;  
247 guint i;  
248  
249 /* This code is inefficient and should not be used in production. */  
250 for (i = 0; i < g_list_length (some_list); i++)  
251 {  
252     gpointer element_data = g_list_nth_data (some_list, i);  
253  
254     /* Do something with @element_data. */  
255 }
```

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

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

```
261 GPtrArray *some_array;  
262 guint i;  
263  
264 for (i = 0; i < some_array->len; i++)  
265 {  
266     gpointer element_data = some_array->pdata[i];  
267  
268     /* Do something with @element_data. */  
269 }
```

## 270 Magic values

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

274 Magic values should be:

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

281 One situation where magic values are used incorrectly is to circumvent the type  
282 system. For example, a magic string value which indicates a special state for  
283 a string variable. Magic values should not be used for this, as the software

284 state could then be corrupted if user input includes that string (for example).  
285 Instead, a separate variable should be used to track the special state. Use the  
286 type system to do this work for you — magic values should never be used as a  
287 basic dynamic typing system.

## 288 **Asynchronous methods**

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

295 Similarly, in non-graphical applications that respond to network requests or **D-  
296 Bus inter-process communication**<sup>31</sup>, blocking the main loop prevents the next  
297 request from being handled.

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

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

303 The second approach (see **Threading**<sup>32</sup> typically leads to complex locking and  
304 synchronisation between threads, and introduces many bugs. The recommended  
305 approach in GLib applications is to use asynchronous operations, implemented  
306 using **GTask**<sup>33</sup> and **GAsyncResult**<sup>34</sup>. Asynchronous operations must be imple-  
307 mented everywhere for this approach to work: any use of a blocking, syn-  
308 chronous operation will effectively make all calling functions blocking and syn-  
309 chronous too.

310 The documentation for **GTask**<sup>35</sup> and **GAsyncResult**<sup>36</sup> includes examples and tuto-  
311 rials for implementing and using GLib-style asynchronous functions.

312 Key principles for using them:

- 313 1. Never call synchronous methods: always use the `*_async()` and `*_finish()`  
314 variant methods.
- 315 2. Never use threads for blocking operations if an asynchronous alternative  
316 exists.

---

<sup>31</sup>[https://jwd.pages.apertis.org/apertis-website/guides/d-bus\\_services/](https://jwd.pages.apertis.org/apertis-website/guides/d-bus_services/)

<sup>32</sup><https://jwd.pages.apertis.org/apertis-website/guides/threading/>

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

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

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

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

- 317 3. Always wait for an asynchronous operation to complete (i.e. for its `GAsync-`  
318 `cReadyCallback` to be invoked) before starting operations which depend on  
319 it.
- 320 • Never use a timeout (`g_timeout_add()`) to wait until an asynchronous  
321 operation ‘should’ complete. The time taken by an operation is unpre-  
322 dictable, and can be affected by other applications, kernel scheduling  
323 decisions, and various other system processes which cannot be pre-  
324 dicted.

## 325 Enumerated types and booleans

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

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

333 This is documented well in the article [Use Enums Not Booleans](#)<sup>37</sup>.

## 334 GObject properties

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

```
338 var temp = my_object.some_property  
339 my_object.some_property = "new value"  
340 my_object.some_property = temp
```

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

## 344 Resource leaks

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

348 For example, the source ID returned by `g_timeout_add()`<sup>40</sup> must always be stored

---

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

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

<sup>39</sup>[https://jwd.pages.apertis.org/apertis-website/guides/memory\\_management/](https://jwd.pages.apertis.org/apertis-website/guides/memory_management/)

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

349 and removed (using `g_source_remove()`<sup>41</sup>) when the owning object is finalised.  
350 This is because it is very rare that we can guarantee the object will live longer  
351 than the timeout period — and if the object is finalised, the timeout left uncan-  
352 celled, and then the timeout triggers, the program will typically crash due to  
353 accessing the object’s memory after it’s been freed.

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

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

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

---

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

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

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

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

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

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

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

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

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

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

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

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

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