Coding Conventions

Contents

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
```

[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://martyn.pages.apertis.org/apertis-website/guides/license-applying/#licensing-of-code
[5] https://martyn.pages.apertis.org/apertis-website/guides/license-applying/

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

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

Memory management

See Memory management[6] for some patterns on handling memory management; particularly single path cleanup[7].

Namespacing

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

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

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

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

- Functions should use `lower_case_with_underscores`.
- Structures, types and objects should use `CamelCaseWithoutUnderscores`.
- Macros and #defines should use `UPPER_CASE_WITH_UNDERSCORES`.

[6] https://martyn.pages.apertis.org/apertis-website/guides/memory_management/

[7] https://martyn.pages.apertis.org/apertis-website/guides/memory_management/#Single-path_cleanup

[8] https://developer.gnome.org/gobject/stable/gtype-conventions.html

- All symbols should be prefixed with a short (2–4 characters) version of the namespace.
- All methods of an object should also be prefixed with the object name.

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

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

- $PREFIX/include/walbottle-$API_MAJOR/walbottle/wbl-schema.h
- $PREFIX/include/walbottle-$API_MAJOR/walbottle/wbl-writer.h

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

For the schema object, the following symbols would be exported (amongst others), following GObject conventions:

- WblSchema structure
- WblSchemaClass structure
- WBL_TYPE_SCHEMA macro
- WBL_IS_SCHEMA macro
- wbl_schema_get_type function
- wbl_schema_new function
- wbl_schema_load_from_data function

Modularity

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

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

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

[9] https://martyn.pages.apertis.org/apertis-website/guides/module_setup/#Parallel_installability
[10] http://en.wikipedia.org/wiki/Modular_programming
[11] http://en.wikipedia.org/wiki/Encapsulation_%28object-oriented_programming%29
[12] http://en.wikipedia.org/wiki/Loose_coupling

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

Pre- and post-condition assertions

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

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

- Assertions: Check for programmer errors and abort the program on failure.
- Validation: Check for invalid input and return an error gracefully on failure.

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

[13] https://autotools.io/libtool/symbols.html

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

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

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

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

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

GError usage

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

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

GError should be used in preference to a simple return code, as it can convey more information, and is also supported by all GLib tools. For example, introspecting an API with GObject introspection[22] will automatically detect all GError parameters so that they can be converted to exceptions in other languages.

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

[15] https://wiki.gnome.org/Projects/GObjectIntrospection/Annotations

[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

[18] https://martyn.pages.apertis.org/apertis-website/guides/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

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

[22] https://wiki.gnome.org/Projects/GObjectIntrospection

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

The GLib API documentation contains a full tutorial for using GError[23].

GList

GLib provides several container types for sets of data:

- `GList`[24]
- `GSList`[25]
- `GPtrArray`[26]
- `GArray`[27]

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

Article on linked list performance[28]

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

```
GList *some_list, *l;

for (l = some_list; l != NULL; l = l->next)
  {
    gpointer element_data = l->data;

    /* Do something with @element_data. */
  }
```

[23] https://developer.gnome.org/glib/stable/glib-Error-Reporting.html#glib-Error-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

Using an incrementing index instead results in an exponential decrease in performance (O(2×N^2) rather than O(N)):

```
GList *some_list;
guint i;

/* This code is inefficient and should not be used in production. */
for (i = 0; i < g_list_length (some_list); i++)
  {
    gpointer element_data = g_list_nth_data (some_list, i);

    /* Do something with @element_data. */
  }
```

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

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

```
GPtrArray *some_array;
guint i;

for (i = 0; i < some_array->len; i++)
  {
    gpointer element_data = some_array->pdata[i];

    /* Do something with @element_data. */
  }
```

Magic values

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

Magic values should be:

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

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

[31] https://martyn.pages.apertis.org/apertis-website/guides/d-bus_services/
[32] https://martyn.pages.apertis.org/apertis-website/guides/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

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 decisions, and various other system processes which cannot be predicted.

Enumerated types and booleans

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

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

This is documented well in the article Use Enums Not Booleans[37].

GObject properties

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

```
var temp = my_object.some_property
my_object.some_property = "new value"
my_object.some_property = temp
```

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

Resource leaks

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

For example, the source ID returned by `g_timeout_add()`[40] must always be stored

[37] http://c2.com/cgi/wiki?UseEnumsNotBooleans
[38] https://developer.gnome.org/gobject/stable/gobject-properties.html
[39] https://martyn.pages.apertis.org/apertis-website/guides/memory_management/
[40] https://developer.gnome.org/glib/stable/glib-The-Main-Event-Loop.html#g-timeout-add

and removed (using `g_source_remove()`[41]) when the owning object is finalised. This is because it is very rare that we can guarantee the object will live longer than the timeout period — and if the object is finalised, the timeout left uncancelled, and then the timeout triggers, the program will typically crash due to accessing the object's memory after it's been freed.

Similarly for signal connections, the signal handler ID returned by `g_signal_connect()`[42] should always be saved and explicitly disconnected (`g_signal_handler_disconnect()`[43]) unless the object being connected is guaranteed to live longer than the object being connected to (the one which emits the signal):

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

- File descriptors (FDs):
  - `g_open()`[44]
  - `g_close()`[45]
- Threads:
  - `g_thread_new()`[46]
  - `g_thread_join()`[47]
- Subprocesses:
  - `g_spawn_async()`[48]
  - `g_spawn_close_pid()`[49]
- D-Bus name watches:
  - `g_bus_watch_name()`[50]
  - `g_bus_unwatch_name()`[51]
- D-Bus name ownership:
  - `g_bus_own_name()`[52]
  - `g_bus_unown_name()`[53]

---

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

[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