



Supported API

1	Contents	
2	New releases and API stability	2
3	API and ABI stability strategies	3
4	The Android approach	4
5	The iOS approach	4
6	The Apertis/OpenSource approach	5
7	The role of limiting the supported API surface	6
8	How would incompatible changes impact the product and how to han-	
9	dle them?	7
10	The GTK+ upgrade	7
11	When a core library breaks	7
12	When a “leaf”library breaks ABI	8
13	ABI is not just library symbols	8
14	The move to Wayland	9
15	API Support levels	9
16	Custom APIs	10
17	Enabling APIs	10
18	OS APIs	11
19	Internal APIs	11
20	External APIs	11
21	Differing stability levels	11
22	Maintaining API stability	12
23	Components	12
24	Conclusion	14
25	The goal of this document is to explain the relevant issues around API (Applica-	
26	tion Programming Interface) and ABI (Application Binary Interface) stability	
27	and to make explicit the APIs and ABIs that can be and will be guaranteed to	
28	be available in the platform for application development.	
29	It will be explained as well how we are going to deal with situations where	
30	certain components break their API/ABI.	

31 New releases and API stability

32 Software systems are typically composed of several components with some de-
33 pending on others. Components need to make assumptions about how their
34 dependencies behave, in order to use them. These assumptions are categorized
35 in API and ABI depending on whether they are resolved at build time or at run-
36 time, respectively. As components evolve over time and their behavior changes,
37 so may their API and ABI.

38 In systems composed of thousands of components, each time a component
39 changes, potentially hundreds of other components could break. Fixing each
40 of those components could cause other breaks in turn. Without a way to man-
41 age those changes, assembling and maintaining non-trivial systems wouldn't be
42 a practical enterprise.

43 To manage this complexity, components which are to be depended upon by
44 others set an API/ABI stability policy. This policy states under which circum-
45 stances new releases can be expected to break API or ABI. This allows the
46 system integrator to update to newer releases of components with some assur-
47 ance that other components won't break as a result. These guarantees also allow
48 new releases of components to simply depend upon the last "known-good" release
49 of each of their dependencies instead of requiring them to be constantly tested
50 against newer dependencies.

51 Most components will keep stable branches in which API - and often ABI -
52 are not allowed to break, and normally only bug fixes and minor features will
53 be merged into these branches. It is generally recommended that components
54 (particularly, stable ones) depend only on stable branches of their dependencies.
55 Releases in a stable branch are referred to as "backwards compatible" because
56 components that depend upon a given release will continue to work with later
57 releases in that same branch.

58 By libraries keeping API stability in stable branches and by libraries and appli-
59 cations depending on stable versions of libraries, breaks are greatly reduced to
60 manageable levels.

61 An API can consist of multiple parts: for a typical C library, the API will
62 be the C function and type declarations, plus the gobject-introspection (GIR)
63 description of the API. Similarly, an ABI can consist of multiple parts: the C
64 function and type declarations, plus the D-Bus API for a system service, for
65 example.

66 The GIR API is especially relevant for further development of Apertis, as it is
67 planned to allow apps to be written in non-C languages such as JavaScript. In
68 this situation, API stability requires both the C declarations to be stable, plus
69 the conversion of those declarations to a GIR file to be stable —so it is affected
70 by changes in the implementation of the GIR scanner (the g-ir-scanner utility
71 provided by gobject-introspection). This is covered further in [ABI is not just
72 library symbols](#).

73 **API and ABI stability strategies**

74 There is a tension between keeping the development environment stable and
75 keeping up with novelties. Following is an investigation about how various mo-
76 bile platforms have tackled this issue that hopefully provides enough information
77 for a practical strategic decision on how to handle that tension.

78 **The Android approach**

79 Android makes a promise of forward-compatibility for the main Android APIs.
80 Although Android has been built on top of Linux and using a Java virtual
81 machine, no APIs of these platforms are considered to be part of the Android
82 platform.

83 Instead of reusing existing components and libraries Google decided to write
84 almost everything from scratch, including a C library, a graphics subsystem,
85 audio, web and multimedia subsystems and APIs.

86 This approach has the big disadvantage of not reusing and sharing much of the
87 work done by the open source community in similar projects, which means a
88 significant investment and hundreds of thousands of hours of engineering time
89 spent building and maintaining everything. On the plus side, those APIs and
90 the underlying components they are built upon are fully controlled by Google,
91 and submit to whatever requirements the Android platform has, giving Google
92 full control regarding tilting the balance in favour of stability or break-through
93 as it sees fit.

94 Although Google has been very successful in keeping its API/ABI stability
95 promises, it has made incompatible changes in almost every release. From API
96 level 13 to 14 (in other words, from Android 3.2 to 4.0) alone there were a few
97 dozen API deprecations and *removals*¹, including methods, class and interface
98 fields, and so on. Each new version brings in its release notes a report of API
99 differences compared to the last version. In addition to these, underlying compo-
100 nent changes have caused applications to misbehave and crash when assuming
101 a certain behaviour that got changed.

102 **The iOS approach**

103 Apple has been known for wanting to control every bit of the products they
104 make. From hardware all the way to third-party application design, Apple tends
105 to influence or enforce its own rules. The iOS is no exception: instead of reusing
106 existing open source APIs, Apple designed and built their own components and
107 APIs from the ground up. The same disadvantages Android's approach has are
108 also present here: instead of sharing the cost of building all of the basic tools
109 with lots of developers world wide, Apple decided to build everything itself,
110 making a significant investment in terms of money and engineering time.

111 The main difference between Android and iOS, though, are that Apple did not
112 have to start from scratch: they had Mac OS X already, and were able to
113 reuse some of the work they have done previously, although that itself brings
114 a disadvantage: the need to balance the needs of the desktop use case and
115 the mobile use case in a single code base. The advantages, though, are the
116 same: Apple is fully in control of the system from the ground up, and can make
117 decisions on tilting the balance between stability and break-through.

¹http://developer.android.com/sdk/api_diff/14/changes/alldiffs_index_removals.html

118 Apple, like Google, has also been successful keeping compatibility, but has had
119 its set of incompatible changes in every release. The [API changes between iOS](#)
120 [4.3 to 5](#)², for instance, has a couple tens of *removed or renamed* classes, fields
121 and methods.

122 **The Apertis/OpenSource approach**

123 Open source projects like GNOME have been very successful at providing bal-
124 ance to the tension by having API/ABI stability promises, but as the need
125 for technology overhauls appeared, keeping backwards compatibility has often
126 proven very costly, and a choice to break compatibility and refresh the platform
127 has been made.

128 That was the case, for instance, with the release of GNOME3. The GNOME
129 project had to some extent maintained compatibility with applications that were
130 written all the way back in 2002, and had accumulated a considerable amount
131 of deprecated functionality and APIs that burdened the project, slowing down
132 progress and requiring a lot of maintenance work. Those had to be left behind
133 the project in order to bring it up-to-date with the expectations of the current
134 decade.

135 The big advantage of using open source components is most of the hard work of
136 building all of the pieces of infrastructure and even some applications has been
137 made, leaving hardware integration, application development, customization,
138 specific features and QA as the main required work before going to market,
139 instead of having a much larger team that would build everything from scratch,
140 or licensing a proprietary components.

141 The main disadvantage to this approach is that the decision on how to tilt
142 the balance between stability and freshness is not under the full control of the
143 company building the product: some decisions will be made by the projects that
144 build the various components that make up the solution that can increase the
145 cost of keeping stability while still maintaining freshness.

146 For instance: Google has full control of Android's underlying graphics stack,
147 Surface Flinger, and is able to ensure its compatibility moving forward; it is
148 also able to make APIs deal transparently with changes in this underlying layer.
149 The same goes for Apple and its iOS. When it comes to the open source graph-
150 ics stack, a move from the current Xorg infrastructure to the next-generation
151 Wayland will break some of the underlying assumptions made by applications.

152 Some of the core libraries that are parts of the graphics stack are also likely to
153 change, taking advantage of the API stability break imposed by the move to
154 a new graphics infrastructure to also perform some changes to their core and
155 APIs. Some projects may also decide to break their stability promises from time
156 to time for technology overhauls, like GNOME did with GNOME 3. We will

²<https://developer.apple.com/library/ios/#releasenotes/General/iOS50APIDiff/index.html>

157 investigate some theoretical and real world cases in order to get a more concrete
158 example of how these overhauls may present themselves, and how they can be
159 handled.

160 There are several options when dealing with backwards-incompatible novelties:
161 delaying the integration of a new release, for instance, is the best way to guar-
162 antee stability, but that will only delay the impact of the changes. Building a
163 set of APIs that abstract some of the platform can also be sensible: applications
164 using high level widgets can be shielded from changes done at the lower levels.

165 To conclude: taking advantage of open source code takes away some of the
166 control over the platform's future. While Google and Apple are able to decide
167 exactly what happens to the components that make up Android and iOS in the
168 future, someone basing their product on an open source platform doesn't. It's
169 important to notice that that is also the case for companies building products
170 based on Android, and maybe even more so: when Google decided that Android
171 Honeycomb would not be released, many companies were left without the latest
172 version of Android to base their products on.

173 Also, like GNOME, Windows and Mac OS have started afresh at some point in
174 time, to be able to bring their products to the next level, it is very likely there
175 will come a time in which iOS and Android will go through a similar major
176 change on their foundations, and companies basing their products on Android
177 will have to decide how to handle the upgrade, when it happens.

178 **The role of limiting the supported API surface**

179 While the API and ABI promises made by Android and iOS have been largely
180 successful, it is important to note that they do not cover everything an appli-
181 cation may need. Core services like graphics and networking are covered, but
182 more specific functionality is not. One example is JSON processing. JSON
183 is one of the most widely used formats for exchanging data between apps and
184 servers.

185 There are no APIs at all for this format in iOS. Applications that need to use
186 JSON need to either roll their own implementation or embed a JSON processing
187 library into their application. The same goes for APIs to access Youtube and
188 other Google services through its GData protocol.

189 See < [http://www.appdevmag.com/10-ios-libraries-to-make-your-](http://www.appdevmag.com/10-ios-libraries-to-make-your-life-easier/)
190 [life-easier/](http://www.appdevmag.com/10-ios-libraries-to-make-your-life-easier/)³ for more examples of missing APIs and replace-
191 ments that can be embedded

192 Android has similar limitations. Android devices are not guaranteed to have
193 APIs for Google services, and although add-ons exist to bolt on those APIs,
194 they cannot be redistributed, in some cases. For services that use GData, there

³<http://www.appdevmag.com/10-ios-libraries-to-make-your-life-easier/>

195 is also an add-on library that can be embedded in the application, but there are
196 no API/ABI guarantees.

197 Imposing those limits on which APIs are guaranteed to not change (or change
198 as little as possible in reality) makes it possible for Android and iOS to lower
199 the maintenance costs for the platform, while making it possible to embed li-
200 braries into applications allows applications to not be completely limited by the
201 available standard APIs. Note also that embedded libraries can only be used
202 by the application embedding it, avoiding inter-application dependencies.

203 **How would incompatible changes impact the product and** 204 **how to handle them?**

205 This section aims at investigating some cases where a line was drawn and old
206 APIs were left behind, and how products based on or simply shipping those
207 APIs handled it. The arrival of GNOME 3 in early 2011 drew the line and
208 allowed for the clean up of APIs that were almost 10 years old, with few or
209 no forward compatibility breakages through that period. It provides a lot of
210 insights at how to handle that kind of structural overhaul.

211 **The GTK+ upgrade**

212 GTK+ is the main toolkit used by the GNOME system. The upgrade to GTK+
213 3.0 was very smooth, for such a big upgrade. Applications required changes, but
214 not all applications needed to be ported at once, since everything that made
215 up the library changed name, making it installable in parallel with GTK+ 2.
216 This means simple applications written using the toolkit still work, even if you
217 have GTK+ 3-based applications installed and working. So that is exactly how
218 distributors handled the situation: both libraries are installed as long as there
219 are applications that need the old one.

220 There are several facilities to make this possible available in the debian pack-
221 aging tools used by the base distribution Apertis is built on, and also in the
222 development tools used by those libraries. Provided they are used correctly this
223 specific case should not prove too difficult. Most distributions that handled this
224 kind of breakage spent a lot of time tuning dependencies and other package
225 relationships, and making sure no interfaces other than the binary ones were in
226 disagreement, though.

227 **When a core library breaks**

228 New versions of core libraries might implement functionality in a different way,
229 which might create issues in some scenarios. A good example of this are browser
230 plugins which rely on the browser being written for a specific version of libraries,
231 such as GTK+. The problem appears if the browser uses or switches to a
232 different version of the library, since as soon as the plugin is loaded there will be
233 symbols from both versions of the library, GTK+ 2 and GTK+ 3 for instance,

234 in the symbol resolution table, and that will lead to subtle and hard to debug
235 bugs and crashes. That is one of the reasons why Firefox has delayed the move
236 to GTK+ 3.

237 The same happens with GStreamer plugins. If a library is used by both a
238 GStreamer plugin and an application, and that library changes the same prob-
239 lem described for browser plugins would happen.

240 Plugins are not the only case in which such problems happen. If a core library
241 like glib breaks compatibility similar issues will appear for all of the platform.
242 Almost every application links to glib and so do many libraries, including core
243 ones like Clutter. If a new version of glib is released which breaks ABI, all of
244 these would have to be migrated to the new library at once, otherwise symbol
245 clashes like the ones described above would happen. In GNOME 3 glib has
246 not broken compatibility, but it is expected to break it at some point in the
247 (medium term) future.

248 As discussed in the previous section, ensuring forward compatibility after such
249 a break in the ABI of glib would only be possible with a very significant effort,
250 and might prove to not be viable. Apertis would recommend that turning points
251 like this be treated as major upgrades to the platform, requiring applications to
252 be reworked. Such upgrades can be delayed by a few releases to allow enough
253 time for the applications to be updated, though.

254 **When a “leaf”library breaks ABI**

255 When a core library such as glib breaks, the impact will be felt throughout the
256 platform, but when a library that is used only by a few components breaks there
257 is more room for adjustment. It’s unlikely that both libraries and applications
258 would link to libstartup-notification, for instance. In such cases the new version
259 of the library can be shipped along with the old one, and the old one can be
260 maintained for as long as necessary.

261 **ABI is not just library symbols**

262 A leaf library may end up causing more issues, though, if it breaks. GNOME
263 3 has provided us with an example of that: the GNOME keyring is GNOME’s
264 password storage. It’s made up of a daemon (that among other things provides
265 a D-Bus service), and a client library for applications to use. GNOME keyring
266 has undergone a change in the protocol, and both the library and the daemon
267 were updated. The library was parallel installable with the old one, but the new
268 daemon completely replaced the old one.

269 But the old client library and the new daemon did not know how to talk to each
270 other, so even though applications would not crash because of a missing library
271 or missing symbols, they were not able to store or obtain passwords from the
272 keyring. That is also what would happen in case a D-Bus service changes its
273 interface.

274 In case something like this happens it is possible to work around the issue by
275 adding code to the daemon to keep supporting the old protocol/interface, but
276 this increases the maintenance burden and the cost/benefits ratio needs to be
277 properly assessed, since it may be significant.

278 Similarly, the GIR interface for a library forms part of its public API. The GIR
279 interface is a high-level, language-agnostic API which maps directly to the C
280 API, and can be used by multiple language bindings to automatically allow the
281 library to be used from those languages. Its stability depends on the stability of
282 the underlying C library, plus the stability of the GIR generation, implemented
283 by g-ir-scanner.

284 **The move to Wayland**

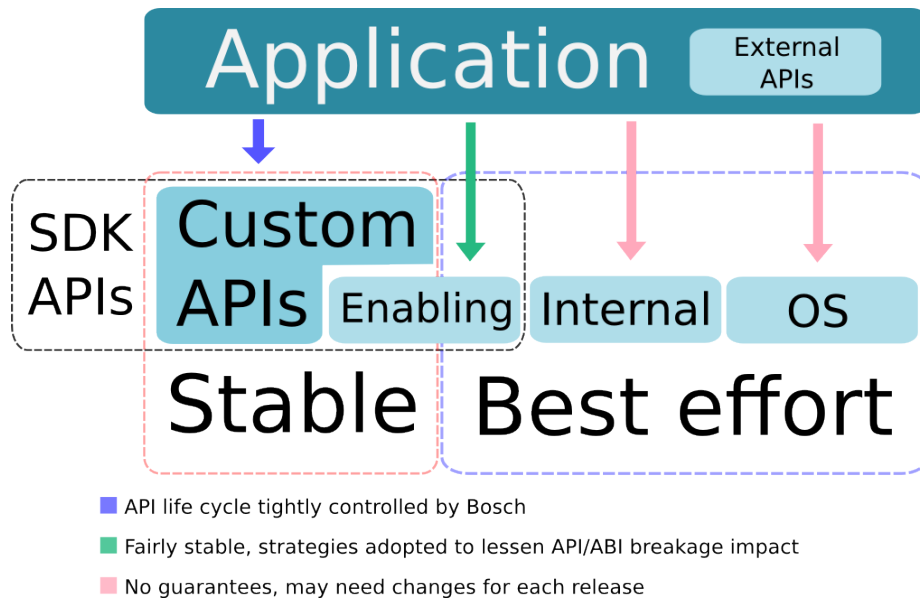
285 Moving to Wayland was a big change, but the impact on application compati-
286 bility may not be that big. Since most applications are built using frameworks
287 such as GTK-3, which hide the different compositor approaches, they just work.
288 Only applications that make use of specific APIs for X need to be ported.

289 That is a good reason for making those APIs part of the unsupported set, and
290 if necessary provide APIs as part of the higher level toolkit to accommodate
291 application needs.

292 **API Support levels**

293 A number of API support levels has been indicated recognizing that some bits
294 of the platform are more prone to change than others, and given the strategy
295 of building higher level custom APIs. The custom and enabling APIs make up
296 what is often called the SDK APIs. They are the ones with better promises,
297 and for which Apertis will try to provide smooth upgrade paths when changes
298 come about, while the APIs on the lower levels will not get as much work, and
299 application developers will be made aware that using them means the app might
300 need to be updated for a platform upgrade.

301 The overall strategy being considered right now to assign APIs to each of these
302 support levels is to start with the minimum set of libraries required to run the
303 Apertis system being part of the image with all libraries assigned to the Internal
304 APIs support level, and gradually promote them as development progresses and
305 decisions are made. The following sections describe the support levels.



306

307 Custom APIs

308 The Custom APIs are high level APIs built on top of the middleware provided by
 309 Apertis. These APIs do not expose objects, types or data from the underlying
 310 libraries, thus providing easier and abstract ways of working with the system.

311 Examples of such APIs are the share functionality, and a number of UI compo-
 312 nents that have been designed and built for the platform. Apertis has had only
 313 limited information about these components, so an assessment of how effectively
 314 they shield store applications from lower support level libraries is currently not
 315 possible.

316 For these components to deliver on their promise of abstracting the lower level
 317 APIs it is imperative that they expose no objects, data types, functions and so
 318 on from other libraries to the application developer.

319 Enabling APIs

320 These APIs are not guaranteed to be stable between platform upgrades, but
 321 work may be done on a case-by-case basis to provide a smooth migration path,
 322 with old versions coexisting with newer ones when possible. Most existing open
 323 source APIs related to core functionality fall in this support level.

324 As discussed in [The GTK upgrade](#), there are ways to deal with ABI/API break-
 325 age in these libraries. Keeping both versions installed for a while is one of
 326 them.

327 **OS APIs**

328 The OS APIs include low level libraries such as glib and its siblings gio, gdbus,
329 as well as system services such as PulseAudio, glibc and the kernel. Applications
330 reaching down to these components would, as is the case for enabling APIs, not
331 necessarily work without changes after a platform upgrade.

332 **Internal APIs**

333 These are APIs used to build the Apertis system itself but not exposed to store
334 applications. A library might get assigned to this support level if it is required
335 to implement system features, but its API is too unstable to expose to from-
336 store applications. Some libraries that fit this support level might also be in the
337 External APIs one.

338 **External APIs**

339 Some libraries are not core enough that they warrant being shipped along with
340 the main system or are not very stable API-wise. One such example is poppler,
341 which changes API and ABI fairly often and is not really required for most
342 applications –it will certainly be used on the main PDF viewing application,
343 and most other applications will simply yield to the system viewer when faced
344 with a PDF file.

345 That means poppler is a good candidate for bundling with the applications that
346 need it instead of being part of the core supported APIs.

347 **Differing stability levels**

348 While the Enabling, Custom, External, Internal and OS categories separate
349 APIs based on the level of control and direct involvement we have over them,
350 a separate dimension is needed to track the stability of APIs, with four levels:
351 private, unstable, stable, and deprecated. An API starts as private, and can
352 transition to any of the other levels. Transitions between stable and deprecated
353 are possible, but an API can never change or go back to being unstable or
354 private once it is stable –this is one of the stability guarantees.

355 It may be possible to move a library from the unstable level to the stable level
356 piecewise, for example by initially exposing a limited set of core functions as
357 stable, while marking the rest of the API as ‘currently unstable’. Old API could
358 later be marked as deprecated. Further, it may be desirable to expose the same
359 API at different levels for different languages. For example, a library might be
360 stable for the C language, but unstable when used from JavaScript, pending
361 further testing and documentation work to mark it as stable.

362 This approach allows a phased introduction of stable APIs, giving sufficient
363 time for them to be thoroughly reviewed and tested before committing to their
364 stability.

365 This could be implemented in the GIR files for an API, with annotations ex-
366 tracted from the gtk-doc comments of the API's C source code —gtk-doc cur-
367 rently supports a 'Stability' annotation. As an XML format, GIR is extensible,
368 and custom attributes could be used to annotate each function and type in an
369 API with its stability, extracted from the gtk-doc comments. Separate docu-
370 mentation manuals could then be generated for the different stability levels, by
371 making small modifications to the documentation generation utilities in gtk-doc.

372 Restricting less stable or deprecated parts of an API from being used by an
373 app written in C is technically complex, and would likely involve compiling two
374 versions of each library. It is suggested that less stable functions and types are
375 always exposed, with the understanding that app developers use them at their
376 own risk of having to keep up with API-incompatible changes between Apertis
377 versions. Their existence would not be obvious, as they would not be included
378 in the documentation for the stable API.

379 By contrast, restricting the use of such APIs from high-level languages is simpler:
380 as all language bindings use GIR, only the GIR files and the infrastructure
381 which handles them needs modifying to support varying the visibility of APIs
382 according to their stability level. The bindings infrastructure already supports
383 'skipping'specific APIs, but this is not currently hooked up to their advertised
384 stability. A small amount of work would be needed to enable that.

385 **Maintaining API stability**

386 It is easy to accidentally break API or ABI stability between releases of a library,
387 and once a release has been made with an API break, that break cannot be
388 undone.

389 The Debian project has some tooling to detect API and ABI changes between
390 releases of a library, though this is invoked at packaging time, which is after the
391 library has been officially released and hence after the damage is done.

392 This tooling could be leveraged to perform the ABI checks before making a
393 library release.

394 While such tools exist for C APIs, no equivalents exist for GIR and D-Bus
395 APIs; the stability of these must currently be checked manually for each release.
396 As both APIs are described using XML formats, developing tools for checking
397 stability of such APIs would not be difficult, and may be a prudent investment.

398 **Components**

399 The following table has a list of libraries that are likely to be on Apertis images
400 or fit into one of the supported levels discussed before. The table has links
401 to documentation and comments on API/ABI stability promises made by each
402 project for reference. As discussed before, fitting components into one of the

403 supported levels will be an iterative process throughout development, so this
 404 table should not be seen as a canonical list of supported APIs.

405 This list shows components available in Apertis in general, some of them might
 406 have been deprecated in the latest release but are still available on older ones.

Name	Version	API reference
GLibc	2.14	http://www.gnu.org/software/libc/manual/html_node/index.html
OpenGL ES	2.0	http://www.khronos.org/opengles/sdk/docs/man/
EGL	1.4	http://www.khronos.org/registry/egl/specs/eglspec.1.4.20110406.pdf
GLib	2.32	https://docs.gtk.org/glib/
Cairo	1.10	http://cairographics.org/documentation/
Pango	1.29	https://docs.gtk.org/Pango/
Cogl	1.10	https://mutter.gnome.org/cogl/
Clutter	1.10	https://mutter.gnome.org/clutter/
Mx	1.4	http://docs.clutter-project.org/docs/mx/stable/
GStreamer	1.0	http://gstreamer.freedesktop.org/data/doc/gstreamer/head/gstreamer/html/
Clutter-GStreamer	1.6	http://docs.clutter-project.org/docs/clutter-gst/stable/
GeoClue	0.12	https://gitlab.freedesktop.org/geoclue/geoclue/-/wikis/home
LibXML2	2.7	https://gnome.pages.gitlab.gnome.org/libxml2/devhelp/general.html
libsoup	2.4	https://libsoup.org/
librest	0.7	https://gnome.pages.gitlab.gnome.org/librest/
libchamplain	0.14.x	https://gnome.pages.gitlab.gnome.org/libchamplain/champlain/
Mutter	3.3	
ConnMan	0.78	http://git.kernel.org/?p=network/connman/connman.git;a=tree;f=doc;hb=
Telepathy-GLib	0.18	http://telepathy.freedesktop.org/doc/telepathy-glib/
Telepathy-Logger	0.2	http://telepathy.freedesktop.org/doc/telepathy-glib/
Folks	0.6	http://telepathy.freedesktop.org/doc/folks/c/
PulseAudio	1.1	http://freedesktop.org/software/pulseaudio/doxygen/
Bluez	4.98	http://git.kernel.org/?p=bluetooth/bluez.git;a=tree;f=doc
libstartup-notification	0.12	See Notes
libecal	3.3	http://developer.gnome.org/libecal/3.3/
SyncEvolution	1.2	http://api.syncevolution.org/
GUPnP	0.18	http://gupnp.org/docs
libGData	0.11	http://developer.gnome.org/gdata/unstable/
Poppler	0.18	There is minimal inline API documentation
libsocialweb	0.26	GLib-based API has no documentation
Grilo	0.1	API docs in sources
Ofono	1.0	http://git.kernel.org/?p=network/ofono/ofono.git;a=tree;f=doc
WebKit-Clutter	1.8.0	
libexif	0.6.20	http://libexif.sourceforge.net/api/
TagLib	1.7	https://taglib.org/api/index.html

407 **Conclusion**

408 Open Source has been chosen in order to be able to reuse code that is freely
409 available and for its customization potential. It is also desired to keep the plat-
410 form up-to-date with fresh new open source releases as they come about. While
411 choosing to leverage Open Source software does lower cost and the required
412 investment significantly, it does bring with it some challenges when compared
413 to building everything and controlling the whole platform, especially when it
414 comes to the tension between stability and novelty.

415 Those challenges will have to be met and worked upon on a case-by-case basis,
416 and trade-offs will have to be made. Like other distributors of open source
417 software have done over the years, delaying adoption of a particular technology
418 or newer versions of a core package goes a long way in ensuring platform stability
419 and providing safe and manageable upgrade paths, so it is certainly an option
420 that must be considered. Other solutions should of course be considered and
421 planned for, including shipping more versions of the same library in parallel.
422 Limiting the API that is considered supported and requiring that some libraries
423 be statically linked or be shipped along with the program are also tools that
424 should be used where necessary.