Supported API

Contents

The goal of this document is to explain the relevant issues around API (Application Programming Interface) and ABI (Application Binary Interface) stability and to make explicit the APIs and ABIs that can be and will be guaranteed to be available in the platform for application development.

It will be explained as well how we are going to deal with situations where certain components break their API/ABI.

New releases and API stability

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

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

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

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

By libraries keeping API stability in stable branches and by libraries and applications depending on stable versions of libraries, breaks are greatly reduced to manageable levels.

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

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

API and ABI stability strategies

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

The Android approach

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

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

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

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

The iOS approach

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

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

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

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

The Apertis/OpenSource approach

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

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

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

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

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

Some of the core libraries that are parts of the graphics stack are also likely to change, taking advantage of the API stability break imposed by the move to a new graphics infrastructure to also perform some changes to their core and APIs. Some projects may also decide to break their stability promises from time 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

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

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

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

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

The role of limiting the supported API surface

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

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

> See < http://www.appdevmag.com/10-ios-libraries-to-make-your-life-easier/%3E[3] for more examples of missing APIs and replacements that can be embedded

Android has similar limitations. Android devices are not guaranteed to have APIs for Google services, and although add-ons exist to bolt on those APIs, 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/%3E

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

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

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

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

The GTK+ upgrade

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

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

When a core library breaks

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

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

The same happens with GStreamer plugins. If a library is used by both a GStreamer plugin and an application, and that library changes the same problem described for browser plugins would happen.

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

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

When a "leaf"library breaks ABI

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

ABI is not just library symbols

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

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

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

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

The move to Wayland

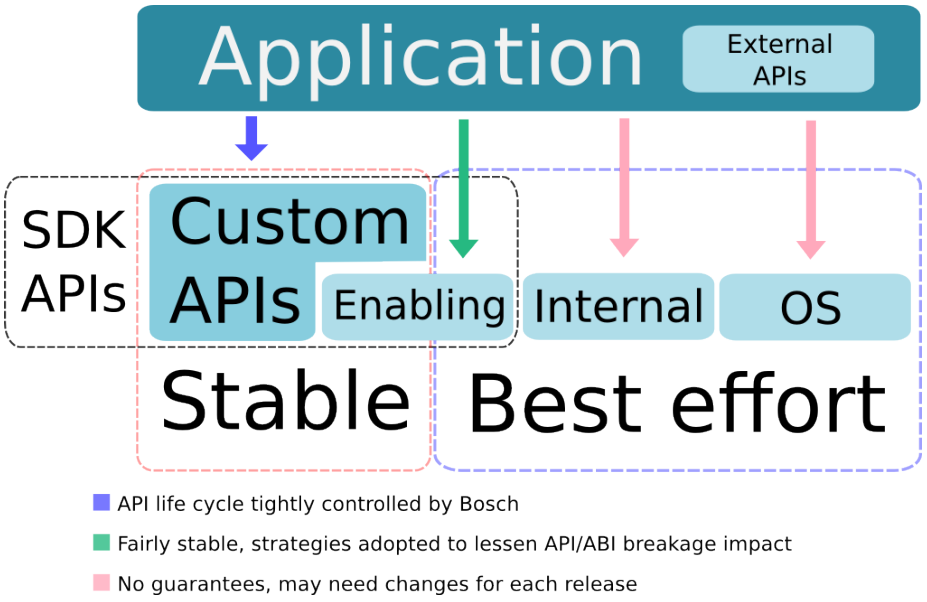
Moving to Wayland was a big change, but the impact on application compatibility may not be that big. Since most applications are built using frameworks such as GTK-3, which hide the different compositor approaches, they just work. Only applications that make use of specific APIs for X need to be ported.

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

API Support levels

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

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



Custom APIs

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

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

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

Enabling APIs

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

As discussed in The GTK upgrade, there are ways to deal with ABI/API breakage in these libraries. Keeping both versions installed for a while is one of them.

OS APIs

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

Internal APIs

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

External APIs

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

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

Differing stability levels

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

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

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

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

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

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

Maintaining API stability

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

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

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

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

Components

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

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

This list shows components available in Apertis in general, some of them might 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/htm
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

Conclusion

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

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