



Application layout

1	<b>Contents</b>	
2	Requirements . . . . .	2
3	Static files . . . . .	2
4	Variable files . . . . .	3
5	Upgrade, rollback, reset and uninstall . . . . .	4
6	System extensions . . . . .	6
7	Security and privacy considerations . . . . .	7
8	Miscellaneous . . . . .	7
9	Provisional recommendations . . . . .	8
10	Writing application bundles . . . . .	8
11	Implementation . . . . .	12
12	Permissions and ownership . . . . .	14
13	Physical layout . . . . .	15
14	Installation and upgrading . . . . .	16
15	Uninstallation . . . . .	18
16	AppArmor profiles . . . . .	19
17	Unresolved design questions . . . . .	19
18	Are downloads rolled back? . . . . .	19
19	Does data reset uninstall apps? . . . . .	20
20	Are inactive themes visible to all? . . . . .	20
21	Are built-in bundles visible to all? . . . . .	20
22	Standard icon sizes? . . . . .	20
23	How do bundles discover the per-user, bundle-independent loca-	
24	tion? . . . . .	20
25	Is <code>g_get_home_dir()</code> bundle-independent? . . . . .	20
26	Is <code>g_get_temp_dir()</code> bundle-independent? . . . . .	21
27	Is <code>PICTURES</code> per-user? . . . . .	21
28	What is the scope of <code>DESKTOP</code> , <code>DOCUMENTS</code> , <code>TEMPLATES</code> ? . . . . .	21
29	Unresolved implementation questions . . . . .	21
30	Can we prevent symlink attacks in shared directories? . . . . .	21
31	Should <code>LD_LIBRARY_PATH</code> be set? . . . . .	21
32	Alternative designs . . . . .	22
33	Merge static and variable files for store applications . . . . .	22
34	Add a third subvolume per app-bundle for cache . . . . .	22
35	Each user’s files under their <code>\$HOME</code> . . . . .	22
36	System integration links for services . . . . .	23
37	System services in app-bundles . . . . .	23
38	Appendix: application layout in Apertis 15.09 . . . . .	24
39	Appendix: comparison with other systems . . . . .	26
40	Desktop Linux (packaged apps) . . . . .	26
41	Flatpak . . . . .	27
42	Android . . . . .	27
43	systemd “revisiting Linux systems” proposal . . . . .	28
44	References . . . . .	28

45 Application bundles in the Apertis system may require several categories of

46 storage, and to be able to write correct AppArmor profiles, we need to be able  
47 to restrict each of those categories of storage to a known directory.

48 This document is intended to update and partially supersede discussions of  
49 storage locations in the [applications](#)<sup>1</sup> and [system updates and rollback](#)<sup>2</sup> design  
50 documents.

51 The [Apertis Application Bundle Specification](#)<sup>3</sup> describes the files that can ap-  
52 pear in an application bundle and are expected to remain supported long-term.  
53 This document provides rationale for those categories of files, suggested future  
54 directions, and details of functionality that is not necessarily long-term stable.

## 55 Requirements

### 56 Static files

- 57 • Most application bundles will contain one or more executable [programs](#)<sup>4</sup>,  
58 in the form of either compiled machine code or scripts. These are read-  
59 only and executable, and are updated when the bundle is updated (and  
60 at no other time).
  - 61 – Some of these programs are designed to be run directly by a user.  
62 These are traditionally installed in `/usr/bin` on Unix systems. Other  
63 programs are *supporting programs*, designed to be run internally  
64 by programs or libraries. These are traditionally installed in  
65 `/usr/libexec` (or sometimes `/usr/lib`) on Unix systems. Apertis  
66 does not require a technical distinction between these categories of  
67 program, but it would be convenient for them to be installed in a  
68 layout similar to the traditional one.
- 69 • Application bundles that contain compiled executables may contain *pri-  
70 vate shared libraries*, in addition to those provided by the [platform](#)<sup>5</sup>, to  
71 support the executable. These are read-only ELF shared libraries, and are  
72 updated when the bundle is updated.
  - 73 – For example, [Frampton](#)<sup>6</sup> has a private shared library `libframptona-  
74 gentiface`<sup>7</sup> containing GDBus interfaces.
- 75 • Application bundles may contain dynamically-loaded *plugins* (also known  
76 as loadable modules). These are also read-only ELF shared libraries.
- 77 • Application bundles may contain static *resource files* such as `.gresource`

---

<sup>1</sup><https://martyn.pages.apertis.org/apertis-website/concepts/applications/>

<sup>2</sup><https://martyn.pages.apertis.org/apertis-website/concepts/system-updates-and-rollback/>

<sup>3</sup><https://martyn.pages.apertis.org/apertis-website/architecture/bundle-spec/>

<sup>4</sup><https://martyn.pages.apertis.org/apertis-website/glossary/#program>

<sup>5</sup><https://martyn.pages.apertis.org/apertis-website/glossary/#platform>

<sup>6</sup><https://gitlab.apertis.org/appfw/frampton>

<sup>7</sup><https://gitlab.apertis.org/appfw/frampton/tree/master/src/interface>

78 resource bundles, icons, fonts, or sample content. These are read-only, and  
79 are updated when the bundle is updated.

- 80 – Where possible, application bundles should [embed resources in the](#)  
81 [executable or library using GResource](#)<sup>8</sup>. However, there are some  
82 situations in which this might not be possible, which will result in  
83 storing resource files in the filesystem:
  - 84 \* if the application will load the resource via an API that is not  
85 compatible with GResource, but requires a real file
  - 86 \* if the resource is extremely large
  - 87 \* if the resource will be read by other programs, such as the icon  
88 that will be used by the app-launcher, the .desktop file describ-  
89 ing an entry point (used by Canterbury, Didcot etc.), or D-Bus  
90 service files (used by dbus-daemon)
- 91 – If a separate `.gresource` file is used, for example for programs written  
92 in JavaScript or Python, then that file needs to be stored somewhere.
- 93 • The AppArmor profile for an application bundle must allow that applica-  
94 tion bundle to read, mmap and execute its own static files.
- 95 • The AppArmor profile for an application bundle must not allow that ap-  
96 plication bundle to *write* its own static files, because they are meant to be  
97 static. In particular, the AppArmor profile itself must not be modifiable.

## 98 Variable files

- 99 • The programs in application bundles may save variable data (configura-  
100 tion, state and/or cached files) for each [user](#)<sup>9</sup> ([Applications design - Data](#)  
101 [Storage](#)<sup>10</sup>).
  - 102 – *Configuration* is any setting or preference for which there is a reason-  
103 able default value. If configuration is deleted, the expected result is  
104 that the user is annoyed by the preference being reset, but nothing  
105 important has been lost.
  - 106 – *Cached files* are files that have a canonical version stored elsewhere,  
107 and so can be deleted at any time without any effect, other than  
108 performance, resource usage, or limited functionality in the absence of  
109 an Internet connection. For example, a client for “tile map” services  
110 like Google Maps or OpenStreetMap should store map tiles in its  
111 cache directory. If cached files are deleted, the expected result is that  
112 the system is slower or less featureful until an automated process can  
113 refill the cache.
  - 114 – Non-configuration, non-cache data includes documents written by  
115 the user, database-like content such as a contact list or address  
116 book, license keys, and other unrecoverable data. It is usually con-

---

<sup>8</sup><https://developer.gnome.org/gio/stable/GResource.html>

<sup>9</sup><https://martyn.pages.apertis.org/apertis-website/glossary/#user>

<sup>10</sup><https://martyn.pages.apertis.org/apertis-website/concepts/applications/#data-storage>

117 considered valuable to the user and should not be deleted, except on the  
118 user's request. If non-configuration, non-cache data is unintention-  
119 ally deleted, the expected result is that the user will try to restore it  
120 from a backup.

- 121 • The programs in application bundles may save variable data (configura-  
122 tion, state and/or cached files) that are shared between all [users](#)<sup>11</sup> ([Appli-  
123 cations design - Data storage](#)<sup>12</sup>).
- 124 • Newport needs to be able to write downloaded files to a designated direc-  
125 tory owned by the application bundle.
  - 126 – Because Newport is a platform service, its AppArmor profile will  
127 need to be allowed to write to *all* apps' directories.
  - 128 – Because downloads might contain private information, Newport must  
129 download to a user- and bundle-specific location.
- 130 • The AppArmor profile for an application bundle must allow that applica-  
131 tion bundle to read and write its own variable files.
- 132 • The AppArmor profile for an application bundle should not allow that  
133 application bundle to execute its own variable files (“write xor execute”),  
134 making a broad class of arbitrary-code-execution vulnerabilities consider-  
135 ably more difficult to exploit.
- 136 • Large media files such as music and videos should normally be shared  
137 between all [users](#)<sup>13</sup> and all multimedia application bundles. ([Multi-user  
138 design - Requirements](#)<sup>14</sup>)

## 139 Upgrade, rollback, reset and uninstall

### 140 Store applications

141 Suppose we have a [store application bundle](#)<sup>15</sup>, Shopping List version 23, which  
142 stores each user's grocery list in a flat file. A new version 24 becomes available;  
143 this version stores each user's grocery list in a SQLite database.

- 144 • Shopping List can be installed and upgraded. This must be relatively  
145 rapid.
- 146 • Before upgrade from version 23 to version 24, the system should make  
147 version 23 save its state and exit, terminating it forcibly if necessary,  
148 so that processes from version 23 do not observe version 24 files or any  
149 intermediate state, which would be likely to break their assumptions and  
150 cause a crash.
  - 151 – This matches the user experience seen on Android: graphical and  
152 background processes from an upgraded `.apk` are terminated during  
153 upgrade.

---

<sup>11</sup><https://martyn.pages.apertis.org/apertis-website/glossary/#user>

<sup>12</sup><https://martyn.pages.apertis.org/apertis-website/concepts/applications/#data-storage>

<sup>13</sup><https://martyn.pages.apertis.org/apertis-website/glossary/#user>

<sup>14</sup><https://martyn.pages.apertis.org/apertis-website/concepts/multiuser/#requirements>

<sup>15</sup><https://martyn.pages.apertis.org/apertis-website/glossary/#store-application-bundle>

- 154 • Before upgrade from version 23 to version 24, the system must take a copy  
155 (snapshot) of each user’s data for this application bundle.
- 156 • After upgrade from version 23 to version 24, the current data will still be  
157 in the version 23 format (a flat file).
- 158 • When a user runs version 24, the application bundle may convert the data  
159 to version 24 format if desired. This is the application author’s choice.
- 160 • If a user rolls back Shopping List from version 24 to version 23, the system  
161 must restore the saved data from version 23 for each user. ([Applications  
162 design](#)<sup>16</sup> §4.1.5, “Store Applications — Roll-back”)
  - 163 – This is because the application author might have chosen to use an  
164 incompatible format for version 24, as we have assumed here.
  - 165 – For simplicity, we do not require a way for application authors to  
166 avoid the data being rolled back.
- 167 • Shopping List can be uninstalled. This must be relatively rapid. ([Appli-  
168 cations design](#)<sup>17</sup> §4.1.4, “Store Applications — Removal”)
- 169 • When Shopping List is uninstalled from the system, the system must  
170 remove all associated data, for all users.
  - 171 – If a multi-user system emulates a per-user choice of apps by hiding  
172 or showing apps separately on a per-user basis, it should delete user  
173 data at the expected time: if user 1 “uninstalls” Shopping List, but  
174 user 2 still wants it installed, the system may delete user 1’s data  
175 immediately.
- 176 • To save space, *cache files* (defined to mean files that can easily be re-  
177 created, for example by downloading them) should not be included in  
178 snapshots. Instead of being rolled back, these files should be deleted during  
179 a rollback. ([System Update and Rollback design](#)<sup>18</sup> §6.3, “Update and  
180 Rollback Procedure”)
- 181 • **Unresolved:** *Are downloads rolled back?*

## 182 Built-in applications

183 By definition, [built-in application bundles](#)<sup>19</sup> are part of the same filesystem  
184 image as the platform. They are upgraded and/or rolled back with the platform.  
185 Suppose platform version 2 has a built-in application bundle, Browser version  
186 17. A new platform version 3 becomes available, containing Browser version 18.

<sup>16</sup><https://martyn.pages.apertis.org/apertis-website/concepts/applications/>

<sup>17</sup><https://martyn.pages.apertis.org/apertis-website/concepts/applications/>

<sup>18</sup><https://martyn.pages.apertis.org/apertis-website/concepts/system-updates-and-rollback/>

<sup>19</sup><https://martyn.pages.apertis.org/apertis-website/glossary/#built-in-application-bundle>

- 187 • The platform can be upgraded. This does not need to be particularly
- 188 rapid: a platform upgrade is a major operation which requires rebooting,
- 189 etc. anyway.
- 190 • Before upgrade from version 2 to version 3, the system must take a copy
- 191 (snapshot) of each user’s data for each built-in application bundle.
- 192 • Immediately after upgrade, the data is still in the format used by Browser
- 193 version 17.
- 194 • If the platform is rolled back from version 3 to version 2, the system must
- 195 restore the saved data from platform version 2 for every built-in applica-
- 196 tion, across all users. ([Applications design](#)<sup>20</sup> §4.2.4, “Built-in Applications
- 197 — Rollback”; [System Update and Rollback design](#)<sup>21</sup> §6.3, “Update and
- 198 Rollback Procedure”)
- 199 • Uninstalling a built-in application bundle is not possible ([Applications](#)
- 200 [design](#)<sup>22</sup> §4.2.3, “Built-in Applications — Removal”) but it should be possible
- 201 to delete all of its variable data, with the same practical result as if an
- 202 equivalent store application bundle had been uninstalled and immediately
- 203 reinstalled.
- 204 • Cache files for built-in applications are treated the same as cache files for
- 205 [Store applications](#), above.

## 206 Global operations

207 User accounts can be created and/or deleted.

- 208 • Deleting a user account does not need to be as rapid as uninstalling an
- 209 application bundle. It should delete that user’s per-user data in all appli-
- 210 cation bundles.

211 A “data reset” operation affects the entire system. It clears everything.

- 212 • A “data reset” does not need to be as rapid as uninstalling an application
- 213 bundle. It should delete all variable data in each application bundle, and
- 214 all variable data that is shared by application bundles.

215 **Unresolved:** [Does data reset uninstall apps?](#)

## 216 System extensions

217 Bundles with sufficient store curator approval and permissions flags may install

218 *system extensions* which will be loaded automatically by platform components.

219 The required permissions may vary according to the type of system extension.

220 For example, a privileged system-wide systemd unit should be a “red flag” which

221 is normally only allowed in built-in applications, whereas a `.desktop` file for a

222 [menu entry](#)<sup>23</sup> should normally be allowed in store bundles, provided that its

<sup>20</sup><https://martyn.pages.apertis.org/apertis-website/concepts/applications/>

<sup>21</sup><https://martyn.pages.apertis.org/apertis-website/concepts/system-updates-and-rollback/>

<sup>22</sup><https://martyn.pages.apertis.org/apertis-website/concepts/applications/>

<sup>23</sup><https://martyn.pages.apertis.org/apertis-website/concepts/application-entry-points/>

223 name matches the relevant ISV’s reversed domain name.

## 224 **Public system extensions**

225 Depending on the type of system extension, an extension might also be intended  
226 to be loaded directly by store applications. For example, every store application  
227 should normally load the current user interface theme, and the set of icons as-  
228 sociated with that theme (although each store application bundle may augment  
229 these with its own private theming and icon data if desired). We refer to exten-  
230 sions of this type as *public system extensions*, analogous to the *public interfaces*  
231 defined by the [Interface discovery](#)<sup>24</sup> design.

## 232 **Security and privacy considerations**

- 233 • Given an AppArmor profile name, it must be easy to determine (for exam-  
234 ple via a library API provided by Canterbury) whether the program  
235 is part of a built-in application bundle, a store application bundle, or the  
236 platform. For application bundles, it must be easy to determine the bun-  
237 dle ID. This is because the uid and the AppArmor profile name are the  
238 only information available to services like Newport that receive requests  
239 via D-Bus.
- 240 • Similarly, given a bundle ID and whether the program is part of a built-in  
241 or store application, it must be easy to determine where it may write.  
242 Again, this is for services like Newport.
- 243 • If existing open source software is included in an application bundle, it  
244 may read configuration from `$prefix/etc` with the assumption that this  
245 path is trusted. Accordingly, we should not normally allow writing to  
246 `$prefix/etc`.
- 247 • The set of installed store application bundles is considered to be confiden-  
248 tial, therefore typical application bundles (with no special permissions)  
249 must not be able to enumerate the entry points, systemd units, D-Bus  
250 services, icons etc. provided by store application bundles. A permission  
251 flag could be provided to make an exception to this rule, for example for  
252 an application-launcher application like Android’s Trebuchet.
  - 253 – **Unresolved:** *Are inactive themes visible to all?*
- 254 • **Unresolved:** *Are built-in bundles visible to all?*

## 255 **Miscellaneous**

- 256 • Directory names should be namespaced by [reversed domain names](#)<sup>25</sup>, so  
257 that it is not a problem if two different vendors produce an app-bundle  
258 with a generic name like “Navigation”.
- 259 • Because we recommend the GNU Autotools (autoconf, automake, libtool),  
260 the desired layout should be easy to arrange by using configure options

---

<sup>24</sup>[https://martyn.pages.apertis.org/apertis-website/concepts/interface\\_discovery/](https://martyn.pages.apertis.org/apertis-website/concepts/interface_discovery/)

<sup>25</sup><https://martyn.pages.apertis.org/apertis-website/glossary/#reversed-domain-name>



261 such as `--prefix`, in a way that can be standardized by build and packaging  
262 tools.

- 263 • Where possible, functions in standard open-source libraries in our stack,  
264 such as GLib, Gtk, Clutter should “do the right thing”. For example,  
265 `g_get_cache_dir()` should continue to be the correct function to call to get  
266 a parent directory for an application’s cache.
- 267 • Where possible, functions in other standard open-source libraries such as  
268 Qt and SDL should generally also behave as we would want. This can  
269 be achieved by making use of common Linux conventions such as the  
270 [XDG Base Directory specification](#)<sup>26</sup> where possible. However, these other  
271 libraries are likely to have less strong integration with the Apertis platform  
272 in general, so there may be pragmatic exceptions to this principle: full  
273 compatibility with these libraries is a low priority.

## 274 Provisional recommendations

275 The overall structure of these recommendations is believed to be valid, but the  
276 exact paths used may be subject to change, depending on the answers to the  
277 [Unresolved design questions](#) and comparison with containerization technologies  
278 such as [Flatpak](#).

## 279 Writing application bundles

280 Application bundle authors should refer to the [Apertis Application  
281 Bundle Specification](#)<sup>27</sup> instead of this section. This section might  
282 describe functionality that is outdated or has not yet been imple-  
283 mented.

## 284 Static data

285 For system-wide static data, programs in application bundles should:

- 286 • link against private shared libraries in the Automake `$libdir` or `$pkglibdir`  
287 via the `DT_RPATH` (libtool will do this automatically)
- 288 • link against public shared libraries provided by the platform in the com-  
289 piler’s default search path, without a `DT_RPATH` (again, libtool will do this  
290 automatically)
- 291 • run executables from the platform, if required, using the normal `$PATH`  
292 search
- 293 • run other executables from the same bundle using paths in the Automake  
294 `$bindir`, `$libexecdir` or `$pkglibexecdir`
- 295 • load static data from the Automake `$datadir`, `$pkgdatadir`, `$libdir` and/or  
296 `$pkglibdir` (using the data directories for architecture-independent data,  
297 and the library directories for data that may be architecture-specific)

---

<sup>26</sup><http://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html>

<sup>27</sup><https://martyn.pages.apertis.org/apertis-website/architecture/bundle-spec/>

- 298           – where possible, resource files should be embedded in the executable or
- 299           library using GResource; if that is not possible, they can be included
- 300           in a `.gresource` resource bundle in the `$datadir` or `$pkgdatadir`; if that
- 301           is not possible either, they can be ordinary files in the `$datadir` or
- 302           `$pkgdatadir`
- 303           – load plugins from the Automake `$pkglibdir` or a subdirectory
- 304      • install system extensions to the appropriate subdirectories of `$datadir` and
- 305        `$prefix/lib`, if used:
  - 306           – `.desktop` files describing entry points (applications and agents) in
  - 307           `$datadir/applications`
  - 308           – D-Bus session services in `$datadir/dbus-1/services`
  - 309           – D-Bus system services in `$datadir/dbus-1/system-services`
  - 310           – systemd user units in `$prefix/lib/systemd/user`
  - 311           – systemd system units in `$prefix/lib/systemd/system`
  - 312           – icons in subdirectories of `$datadir/icons` according to the [freedesk-](https://standards.freedesktop.org/icon-theme-spec/icon-theme-spec-latest.html)
  - 313           [top.org Icon Theme Specification](https://standards.freedesktop.org/icon-theme-spec/icon-theme-spec-latest.html)<sup>28</sup>

314 All of these paths will be part of the application bundle. For store applications,  
 315 they will be installed, upgraded, rolled back and removed as a unit. For built-in  
 316 applications, all of these paths will be part of the platform image.

## 317 Icons and themes

318 *This section might be split out into a separate design document as more require-*  
 319 *ments become available.*

320 Icons should be installed according to the [freedesktop.org Icon Theme specifica-](https://standards.freedesktop.org/icon-theme-spec/icon-theme-spec-latest.html)  
 321 [tion](https://standards.freedesktop.org/icon-theme-spec/icon-theme-spec-latest.html)<sup>29</sup>.

322 If an application bundle installs a general-purpose icon that should represent an  
 323 included application throughout the Apertis system, it should be installed in the  
 324 `hicolor` fallback theme, i.e. `$datadir/icons/hicolor/$size/apps/$app_id.$format`,  
 325 where `$size` is either a pixel-size or `scalable`, and `$format` is `png` or `svg`.

326           The reserved icon theme name `hicolor` is used as the fallback when-  
 327           ever a specific theme does not have the required icon, as specified in  
 328           the [freedesktop.org Icon Theme specification](https://standards.freedesktop.org/icon-theme-spec/icon-theme-spec-latest.html)<sup>30</sup>. The name `hicolor`  
 329           was chosen for historical reasons.

330 If an application author knows about specific icon themes and wishes to in-  
 331 stall additional icons styled to coordinate with those themes, they may create  
 332 `$datadir/icons/$theme_name/$size/apps/$app_id.$format` for that purpose. This  
 333 should not be done for themes where the desired icon is simply a copy of the  
 334 `hicolor` icon.

<sup>28</sup><http://standards.freedesktop.org/icon-theme-spec/icon-theme-spec-latest.html>

<sup>29</sup><http://standards.freedesktop.org/icon-theme-spec/icon-theme-spec-latest.html>

<sup>30</sup><http://standards.freedesktop.org/icon-theme-spec/icon-theme-spec-latest.html>

335 *Rationale:* Suppose there is a popular theme named `org.example.metallic`, and a  
336 popular application named `com.example.ShoppingList`. If the author of Shopping  
337 List has designed an icon that matches the metallic theme, we would like the  
338 application launcher to use that icon. If not, the author of the metallic theme  
339 might have included an icon in their theme that matches this popular applica-  
340 tion; we would like to use that icon as our second preference. Finally, if there  
341 is no metallic-styled icon available, the launcher should use the application’s  
342 theme-agnostic icon from the `hicolor` fallback directory. We can achieve this  
343 result by placing icons from each app bundle’s `$datadir` in an early position in  
344 the launcher’s `XDG_DATA_DIRS`, and placing icons from the theme itself in a later  
345 position in `XDG_DATA_DIRS`: the freedesktop Icon Theme lookup algorithm will  
346 look for a metallic icon in all the directories listed in `XDG_DATA_DIRS` before it  
347 falls back to the `hicolor` theme.

348 The application may install additional icons representing actions, file types,  
349 emoticons, status indications and so on into its `$datadir/icons`. For example,  
350 a web browser might require an icon representing “incognito mode”, which is  
351 probably not present in all icon themes. Similar to the application icon, the  
352 browser may install variants of that icon for themes other than `hicolor`, if its  
353 author is aware of particular themes and intends the icon to coordinate with  
354 those themes.

355 **Unresolved:** [Standard icon sizes?](#)

## 356 Per-user, per-bundle data

357 For *cached files* that are specific to the application and also specific to a user,  
358 programs in application bundles may read and write the directory given by  
359 `g_get_user_cache_dir()` or by the environment variable `XDG_CACHE_HOME`. This lo-  
360 cation is kept intact during upgrades, but is not included in the snapshot made  
361 during upgrade, so it is effectively emptied during rollback. It is also removed  
362 by uninstallation or a data reset.

363 For *configuration* that is specific to the application and also specific to a user, the  
364 preferred API is the `GSettings` abstraction described in the [Preferences and Per-  
365 sistence design document](#)<sup>31</sup>. As an alternative to that API, programs in applica-  
366 tion bundles may read and write the directory given by `g_get_user_config_dir()`,  
367 or equivalently by the environment variable `XDG_CONFIG_HOME`. This locations is  
368 kept intact and also backed up during upgrades, restored to its old contents  
369 during a rollback, and removed by uninstallation of the bundle, deletion of the  
370 user account, or a data reset.

371 For other variable data that is specific to the application and also specific  
372 to a user, programs in application bundles may read and write the directory  
373 given by `g_get_user_data_dir()`, or equivalently by the environment variable  
374 `XDG_DATA_HOME`. This location has the same upgrade, rollback and removal be-

---

<sup>31</sup><https://martyn.pages.apertis.org/apertis-website/concepts/preferences-and-persistence/>

375 behaviours as `g_get_user_config_dir()`. Applications may distinguish between con-  
376 figuration and other variable data, but we do not anticipate that this will be  
377 necessary in Apertis.

378 For downloads, programs in application bundles may read and write the result of  
379 `g_get_user_special_dir (G_USER_DIRECTORY_DOWNLOADS)`. Each application bundle  
380 may assume that it has a download directory per user, shared by all separate  
381 from other users and other application bundles. The download service, Newport,  
382 may also write to this location. Uninstalling the application bundle or removing  
383 the user account causes the download directory to be deleted.

384 **Unresolved:** [Are downloads rolled back?](#)

### 385 **Per-user, bundle-independent data**

386 For variable data that is shared between all applications but specific to a user,  
387 programs in application bundles may read and write locations in the user's sub-  
388 directory of `/home` if they have appropriate permissions flags for their AppArmor  
389 profiles to allow it. We should restrict this capability, because it may affect the  
390 behaviour of other applications.

391 These locations should not be what is returned by `g_get_config_home()`, because  
392 we want the default to be that app bundles are self-contained. We could po-  
393 tentially provide a way to arrange for specific directories to be symlinked or  
394 bind-mounted into the normally-app-specific `g_get_user_config_dir()` and so on.

395 These locations are not subject to upgrade or rollback, and are never cleared or  
396 removed by uninstalling an app-bundle. They are cleared when the user account  
397 is deleted, or when a data-reset is performed on the entire device.

398 **Unresolved:** [How do bundles discover the per-user, bundle-independent loca-  
399 tion?](#)

400 **Unresolved:** [Is `g\_get\_home\_dir\(\)` bundle-independent?](#)

### 401 **User-independent, per-bundle data**

402 As of Apertis 16.12, this feature has not yet been implemented.

403 For variable data that is specific to the application but shared be-  
404 tween all users, programs in application bundles may read and write  
405 `/var/Applications/$bundle_id/cache,` `/var/Applications/$bundle_id/config`  
406 and/or `/var/Applications/$bundle_id/data`. Convenience APIs to construct  
407 these paths should be provided in `libcantebury`. Ribchester should create and  
408 `chmod` these directories if and only if the app has a permissions flag saying it  
409 uses them, so that the system will deny access otherwise.

410 These locations have the same upgrade and rollback behaviour as the per-user,  
411 per-bundle data areas. They are deleted by a whole-device data reset, but are  
412 not deleted if an individual user account is removed.

### 413 Shared data

414 For media files, programs in application bundles may read and write the result of  
415 `g_get_user_special_dir (G_USER_DIRECTORY_MUSIC)` and/or `g_get_user_special_dir`  
416 `(G_USER_DIRECTORY_VIDEOS)`. These locations are shared between users and be-  
417 tween bundles. The platform may deny access to these locations to bundles  
418 that do not have a special permissions flag.

419 For other variable data that is shared between all applications and all  
420 users, programs in application bundles may read and write the result of  
421 `g_get_user_special_dir (G_USER_DIRECTORY_PUBLIC_SHARE)`. The platform may  
422 deny access to this location to bundles that do not have a special permissions  
423 flag. This location is shared between users and between bundles.

424 These locations are unaffected by upgrade or rollback, but will be cleared by a  
425 data reset.

### 426 Other well-known directories

427 **Unresolved:** Is `PICTURES` per-user?

428 **Unresolved:** What is the scope of `DESKTOP`, `DOCUMENTS`, `TEMPLATES`?

### 429 Implementation

430 Application bundles should be installed according to the [Apertis Application](#)  
431 [Bundle Specification](#)<sup>32</sup>. This document does not duplicate the information pro-  
432 vided in that specification, but only gives rationale.

433 The split between `/Applications` OR `/usr/Applications` for static data, and  
434 `/var/Applications` for variable data, makes it easy for developers and AppAr-  
435 mor profiles to distinguish between static and variable data. It also results in  
436 the two different algorithms used during upgrade for store apps being applied  
437 to different directories.

438 The additional split between `/Applications` for store application bundles, and  
439 `/usr/Applications` for built-in application bundles, serves two purposes:

- 440 • `/usr` is part of the *system partition*, which is read-only at runtime (for  
441 robustness), contains the platform and built-in application bundles, and  
442 has a limited storage quota because the safe upgrade/rollback mechanism  
443 means it appears on-disk twice. `/Applications` is part of the *general storage*  
444 *partition*, which has a more generous storage quota and is read/write at  
445 runtime.
- 446 • Using a distinctive prefix for built-in application bundles makes it trivial  
447 to identify built-in applications from their AppArmor profile names, which  
448 are conventionally linked to the programs' filenames.

---

<sup>32</sup><https://martyn.pages.apertis.org/apertis-website/architecture/bundle-spec/>

449 The specified layout was chosen so that the static files in `share/` and  
450 `lib/` could be organised in the way that would be conventional for  
451 Automake installation with a `--prefix=/Applications/$bundle_id` or `--`  
452 `prefix=/usr/Applications/$bundle_id` option. For example, because the  
453 app icon in a store app bundle is named something like `/Applica-`  
454 `tions/$bundle_id/share/icons/hicolor/$size/apps/$entry_point_id.png`, it  
455 can be installed to `${datadir}/icons/hicolor/$size/apps/$entry_point_id.png` in  
456 the usual way.

457 If there are any non-Automake-based application bundles, they should be con-  
458 figured to install in the same GNU-style directory hierarchy that we would use  
459 with Automake, with the analogous parameter corresponding to `${prefix}`. We  
460 do not recommend distributing non-Automake-based application bundles.

461 The top-level `config`, `cache`, `data` directories within the bundle's variable data  
462 should only be created if the application bundle has special permissions flags.  
463 `config`, `cache`, `data` should be considered to be a minor “red flag” by app-store  
464 curators: because they share data across user boundaries, they come with some  
465 risk.

## 466 System integration links for built-in applications

467 The `.deb` package for built-in applications should also include symbolic links for  
468 the following system integration files:

- 469 • Entry points: `link /usr/share/applications/*.service` points to  
470 `/usr/Applications/$bundle_id/share/applications/*.service`
- 471 • Icons: `/usr/share/icons/*` → `/usr/Applications/$bundle_id/share/icons/*`
- 472 • Other theme files: `/usr/share/themes/*` → `/usr/Applications/$bundle_id/share/themes/*`

473 Store applications must not contain these links: similar links are created at  
474 install-time instead. See [Store application system integration links](#) for details.

## 475 Special directory configuration

476 Programs in store application bundles should be run with these environment  
477 variables, so that they automatically use appropriate directories:

- 478 • `XDG_DATA_HOME=/var/Applications/$bundle_id/users/$uid/data` (used by  
479 `g_get_user_data_dir`)
- 480 • `XDG_DATA_DIRS=/Applications/$bundle_id/share:/var/lib/apertis_extensions/public:/usr/share`  
481 (used by `g_get_system_data_dirs`)  
482 – See [Store application system integration links](#) for the rationale for  
483 `/var/lib/apertis_extensions/public`
- 484 • `XDG_CONFIG_HOME=/var/Applications/$bundle_id/users/$uid/config` (used by  
485 `g_get_user_config_dir`)
- 486 • `XDG_CONFIG_DIRS=/var/Applications/$bundle_id/etc/xdg:/Applications/$bundle_id/etc/xdg:/etc/xdg`  
487 (used by `g_get_system_config_dirs`)

- 488 • XDG\_CACHE\_HOME=/var/Applications/\$bundle\_id/users/\$uid/cache (used by
- 489 g\_get\_user\_cache\_dir)
- 490 • PATH=/Applications/\$bundle\_id/bin:/usr/bin:/bin (used when executing
- 491 programs)
- 492 • XDG\_RUNTIME\_DIR=/run/user/\$uid (used by g\_get\_user\_runtime\_dir and pro-
- 493 vided automatically by systemd; access is subject to a “whitelist”)

494 **Unresolved: Should LD\_LIBRARY\_PATH be set?**

495 This is automatically done by canterbury-exec in Apertis 16.06 or later, unless  
 496 the entry point’s bundle ID cannot be determined from its .desktop file. For  
 497 backwards compatibility, Canterbury in Apertis 16.09 still attempts to run entry  
 498 points whose bundle ID cannot be determined, but this should be prevented in  
 499 future.

500 Built-in application bundles should be given the same environment variables,  
 501 but with /usr/Applications replacing /Applications.

502 **Unresolved: Is g\_get\_home\_dir() bundle-independent?**

503 **Unresolved: Is g\_get\_temp\_dir() bundle-independent?**

504 In addition, the XDG special directories should be configured as follows for both  
 505 built-in and store application bundles:

- 506 • g\_get\_user\_special\_dir (G\_USER\_DIRECTORY\_DESKTOP): **Unresolved: What**  
 507 **is the scope of DESKTOP, DOCUMENTS, TEMPLATES?**
- 508 • g\_get\_user\_special\_dir (G\_USER\_DIRECTORY\_DOCUMENTS): **Unresolved:**  
 509 **What is the scope of DESKTOP, DOCUMENTS, TEMPLATES?**
- 510 • g\_get\_user\_special\_dir (G\_USER\_DIRECTORY\_DOWNLOAD): /var/Applications/\$bundle\_id/users/\$uid/downloads
- 511 • g\_get\_user\_special\_dir (G\_USER\_DIRECTORY\_MUSIC): /home/shared/Music
- 512 • g\_get\_user\_special\_dir (G\_USER\_DIRECTORY\_PICTURES): **Unresolved: Is PIC-**  
 513 **TURES per-user?**
- 514 • g\_get\_user\_special\_dir (G\_USER\_DIRECTORY\_PUBLIC\_SHARE): /home/shared
- 515 • g\_get\_user\_special\_dir (G\_USER\_DIRECTORY\_TEMPLATES): **Unresolved:**  
 516 **What is the scope of DESKTOP, DOCUMENTS, TEMPLATES?**
- 517 • g\_get\_user\_special\_dir (G\_USER\_DIRECTORY\_VIDEOS): /home/shared/Videos

518 Again, this is automatically done by canterbury-exec in Apertis 16.06 or later.

## 519 Permissions and ownership

520 All files under /usr/Applications and /Applications should be owned by root,  
 521 with the standard system permissions (u=rwx,og=rX — that is, root may write,  
 522 and all users may read all files, execute programs that are marked executable  
 523 and enter directories).

524 /var/Applications, /var/Applications/\$bundle\_id and /var/Applications/\$bundle\_id/users/  
 525 are also owned by root, with the standard system permissions.

526 If they exist, `/var/Applications/$bundle_id/{config,data,cache}/` are owned by  
527 root, with permissions `a=rwx`. If they are not required and allowed by a permis-  
528 sions flag, they must not exist.

529 **Unresolved:** `Can we prevent symlink attacks in shared directories?`

530 `/var/Applications/$bundle_id/users/$uid/` and all of its subdirectories are owned  
531 by `$uid`, with permissions `u=rwx,og-rwx` for privacy (in other words, only acces-  
532 sible by the owner or by root).

### 533 Physical layout

534 The application-visible directories in `/var/Applications` and `/Applications` are  
535 only mount points. Applications' real storage is situated on the general storage  
536 volume, in the following layout:

```
537 <general storage volume>
538 └─app-bundles/
539   │ └─com.example.MyApp/ (store app-bundle)
540   │ │ └─current → version-1.2.2-1 (symbolic link)
541   │ │ └─rollback → version-1.0.8-2 (symbolic link)
542   │ │ └─version-1.0.8-2/
543   │ │ │ └─static/ (subvolume)
544   │ │ │ │ └─bin/
545   │ │ │ │ └─share/ (etc.)
546   │ │ │ └─variable/ (subvolume)
547   │ │ │ │ └─users/
548   │ │ │ │ │ └─1001/
549   │ │ │ │ │ └─cache/
550   │ │ │ │ │ └─config/
551   │ │ │ │ └─data/ (etc.)
552   │ │ └─version-1.2.2-1/
553   │ │ │ └─static/ (subvolume)
554   │ │ │ └─variable/ (subvolume)
555   │ └─org.apertis.Frampton/ (store app-bundle)
556   │ │ └─current → version-2.5.1-1 (symbolic link)
557   │ │ └─version-2.5.1-1/
558   │ │ │ └─variable/ (subvolume)
559   ... <other directories subvolumes unrelated to application bundles>
```

560 The `static` and `variable` directories are `btrfs` subvolumes so that they can be  
561 copied using snapshots, while the other directories shown may be either subvol-  
562 umes or ordinary directories. The `current` and `rollback` symbolic links indicate  
563 the currently active version, and the version to which a rollback would move,  
564 respectively.

565 Built-in application bundles do not have a `static` subvolume, because their static  
566 files are part of `/usr` on the read-only operating system volume.



567 All other filenames in this hierarchy are reserved for the application manager,  
568 which may create temporary directories and symbolic links during its operation.  
569 It must create these in such a way that it can recover from abrupt power loss  
570 at any point, for example by making careful use of POSIX atomic filesystem  
571 operations to implement “transactions”.

572 During normal operation, the subvolumes would be mounted as follows:

```
573 com.example.MyApp/current/static      → /Applications/com.example.MyApp
574 com.example.MyApp/current/variable   → /var/Applications/com.example.MyApp
575 org.apertis.Frampton/current/variable → /var/Applications/org.apertis.Frampton
```

576 so that the expected paths such as `/var/Applications/com.example.MyApp/users/1001/config/`  
577 are made available.

578 Only one subvolume per application is mounted – under normal circumstances,  
579 this will be the one with the highest version. After a system rollback it might  
580 be an older version if the most recent is unlaunchable.

## 581 Installation and upgrading

582 Suppose we are installing `com.example.MyApp` version 2, or upgrading it from  
583 version 1 to version 2. An optimal implementation would look something like  
584 this:

- 585 • If it was already installed:
  - 586 – Instruct any running processes belonging to that bundle to exit
  - 587 – Wait for the processes to save their state and exit; if a timeout is  
588 reached, kill the processes
  - 589 – Unmount the `com.example.MyApp/version-1/static` subvolume from  
590 `/Applications/com.example.MyApp`
  - 591 – Unmount the `com.example.MyApp/version-1/variable` subvolume from  
592 `/var/Applications/com.example.MyApp`
  - 593 – Create a snapshot of `com.example.MyApp/version-1/static` named  
594 `com.example.MyApp/version-2/static`
  - 595 – Create a new snapshot of `com.example.MyApp/version-1/variable`,  
596 named `com.example.MyApp/version-2/variable`
  - 597 – Recursively delete the `cache` and `users/*/cache` directories from  
598 `com.example.MyApp/version-1/variable`
- 599 • If it was not already installed, instead:
  - 600 – Create a new, empty subvolume `com.example.MyApp/version-`  
601 `2/variable` to be mounted at `/var/Applications/com.example.MyApp`
  - 602 – Create a new, empty subvolume `com.example.MyApp/version-2/static`  
603 to be mounted at `/Applications/com.example.MyApp`
- 604 • For each existing static file in `com.example.MyApp/version-2/static` that was  
605 carried over from `com.example.MyApp/version-1/static`:
  - 606 – If there is no corresponding file in version 2, delete it

- 607       – If its contents do not match the corresponding file in version 2, delete
- 608       it
- 609       – If its metadata do not match the one in version 2, update the meta-
- 610       data
- 611     • For each static file in version 2:
  - 612       – If there is no corresponding file in `com.example.MyApp/version-`
  - 613       `2/static`, the file is either new or changed. Unpack the new
  - 614       version.
- 615     • *(Optional, if support for this feature is required)* Copy any files required
- 616       from `share/factory/{etc,var}` to `{etc,var}`, overwriting files retained from
- 617       previous versions if and only if the retained version matches what is
- 618       in version 1's `share/factory/{etc,var}` but does not match version 2's
- 619       `share/factory/{etc,var}`

620 A simpler procedure would be to create the `com.example.MyApp/version-2/static`  
 621 subvolume as empty, and then unpack all of the static files from the new version.  
 622 However, that procedure would not provide de-duplication between consecutive  
 623 versions if a file has not changed. As of Apertis 16.09, only this simpler proce-  
 624 dure has been implemented.

625 Ribchester (and perhaps Canterbury) must be modified to create the per-user  
 626 directories `/var/Applications/$bundle_id/users/$uid`. This was implemented in  
 627 Apertis 16.06.

## 628 Store application system integration links

629 Application installation for store applications may set up symbolic links in  
 630 `/var/lib/apertis_extensions` for the categories of system integration files de-  
 631 scribed in [System integration links for built-in applications](#), but the files and  
 632 their contents must be restricted unless the bundle has special permissions flags.  
 633 In particular, all entry points (agents and applications) in a bundle must be in  
 634 the relevant ISV<sup>33</sup>'s namespace.

635 For example, an application bundle containing a user interface and an agent  
 636 could be linked like this:

- 637     • `/var/lib/apertis_extensions/applications/com.example.MyApp.UI.desktop`
- 638       → `/Applications/com.example.MyApp/share/applications/com.example.MyApp.UI.desktop`
- 639     • `/var/lib/apertis_extensions/applications/com.example.MyApp.Agent.desktop`
- 640       → `/Applications/com.example.MyApp/share/applications/com.example.MyApp.Agent.desktop`

641 The designers of Apertis can introduce new system integration points in future  
 642 versions if required.

643 The platform components that need to support loading “extension” compo-  
 644 nents from store application bundles will be modified or configured to look  
 645 in `/var/lib/apertis_extensions`. For example, Canterbury could be run with

<sup>33</sup><https://martyn.pages.apertis.org/apertis-website/glossary/#isv>

646 XDG\_DATA\_DIRS=/var/lib/apertis\_extensions:/usr/share so that it will pick up ac-  
647 tivatable services from /var/lib/apertis\_extensions/dbus-1/services.

## 648 System integration links for public extensions

649 /var/lib/apertis\_extensions should *not* be included in the XDG\_DATA\_DIRS for  
650 store applications, so that store applications do not automatically attempt to  
651 read these restricted directories and receive AppArmor denials. However, a few  
652 types of system extension should be loaded by all programs, not just privileged  
653 platform components. For example, GUI themes would typically provide icons  
654 in \$datadir/icons and other related files in \$datadir/themes, which are intended  
655 to be loaded by arbitrary applications (so that those applications coordinate  
656 with the theme).

657 We recommend that the system bind-mounts or copies these files into the cor-  
658 responding subdirectory of /var/lib/apertis\_extensions/public. In conjunction  
659 with the environment variables described above, this means that libraries and  
660 applications that follow the [XDG Base Directory specification](#)<sup>34</sup>, for example  
661 Gtk's theme support, will load them automatically.

662 Please note that symbolic links are *not* suitable for public extensions,  
663 because AppArmor access-control is based on the result of dereferenc-  
664 ing the symbolic link: if a store application com.example.ShoppingList  
665 renders widgets using the org.example.metallic theme, it would not  
666 be allowed to read through a symbolic link that points into /Applica-  
667 tions/org.example.metallic/share/themes/org.example.metallic/, but it can be  
668 allowed to read the same directory indirectly by bind-mounting that directory  
669 onto /var/lib/apertis\_extensions/public/themes/org.example.metallic/.

## 670 Uninstallation

- 671 • Uninstalling a store application bundle consists of removing /Applica-  
672 tions/\$bundle\_id, /var/Applications/\$bundle\_id and the corresponding sub-  
673 volumes.
- 674 • Uninstalling a built-in application bundle is not possible, but it can be  
675 reset (equivalent to uninstallation and reinstallation) by deleting and re-  
676 creating /var/Applications/\$bundle\_id and its corresponding subvolumes.
- 677 • Deleting a user should delete every directory matching /var/Applications/\*/users/\$uid,  
678 in addition to the user's home directory.
- 679 • A “data reset” consists of:
  - 680 – deleting and re-creating /var/Applications/\$bundle\_id for every appli-  
681 cation bundle
  - 682 – *(optional, if a data reset is intended to uninstall store app bundles)*  
683 clearing /Applications
  - 684 – *(optional, if this feature is required)* populating {etc,var} from  
685 share/factory/{etc,var} as if for initial installation

---

<sup>34</sup><http://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html>

## 686 AppArmor profiles

687 Every application bundle should have rules similar to these in its AppArmor  
688 profile:

- 689 • #include <abstractions/chaiwala-base> (normal “safe” functionality)
- 690 • /{usr/,}Applications/\$bundle\_id/{bin,lib,libexec}/\*\* mr (map libraries  
691 and the executable described by the profile; read arch-dependent static  
692 files)
- 693 • /{usr/,}Applications/\$bundle\_id/{bin,libexec}/\*\* pix (run other executables  
694 from the same bundle under their own profile, or inherit current profile  
695 if they do not have their own)
- 696 • /{usr/,}Applications/\$bundle\_id/share/\*\* r (read arch-independent static  
697 files)
- 698 • owner /var/Applications/\$bundle\_id/users/\*\* rwk (read, write and lock per-  
699 app, per-user files for the user running the app)

700 Note that a write is only allowed if it is allowed by both AppArmor and file  
701 permissions, so user A is normally prevented from accessing user B’s files by file  
702 permissions. The last rule is given the `owner` keyword only for completeness.

703 Application bundles that require them may additionally have rules similar to  
704 these:

- 705 • /var/Applications/\$bundle\_id/{config,data,cache}/\*\* rwk (read, write,  
706 lock per-bundle, cross-user variable files)
- 707 • /home/shared/{Music,Videos} rwk (read, write, lock cross-bundle, cross-user  
708 media files)
- 709 • /home/shared/{, \*} rwk (read, write, lock all cross-bundle, cross-user files)
- 710 • owner /home/\*/\$something rwk (read, write, lock selected cross-bundle, per-  
711 user files for the user running the app)

712 <abstractions/chaiwala-base> should be modified to include

- 713 • /var/lib/apertis\_extensions/public/\*\* r

714 to support public extensions.

## 715 Unresolved design questions

### 716 Are downloads rolled back?

717 Newport stores downloaded files in a directory per (bundle ID, user) pair. When  
718 an app is rolled back, are those files treated like a cache (deleted), or treated  
719 like user data (also rolled back), or left as they are?

### 720 Does data reset uninstall apps?

721 Does a data reset leave the installed store apps installed, or does it uninstall  
722 them all? (In other words, does it leave store apps’ static files intact, or does it

723 delete them?)

724 **Are inactive themes visible to all?**

725 Suppose the system-wide theme is “blue”, and the user has installed but not acti-  
726 vated “red” and “green” themes from the app store. Is it OK for an unprivileged  
727 app-bundle to be able to see that the “red” and “green” themes exist?

- 728 • The same applies to any other **Public system extensions**.
- 729 • For simplicity, we recommend the answer “yes, this is acceptable” unless  
730 there is a reason to do otherwise.

731 **Are built-in bundles visible to all?**

732 We know that unprivileged app-bundles are not allowed to enumerate the store  
733 application bundles that are installed. Is it OK for an unprivileged app-bundle  
734 to be allowed to enumerate the built-in application bundles?

- 735 • For simplicity, we recommend the answer “yes, this is acceptable” unless  
736 there is a reason to do otherwise.

737 **Standard icon sizes?**

738 Are there specific icon sizes that we want to require every app to supply? As of  
739 November 2015, the “Mildenhall” reference HMI uses 36x36 icons. Launchers  
740 should be prepared to scale icons as a fallback, but scaled icons at small pixel  
741 sizes tend to look blurry and low-quality, so icons of exactly the size required  
742 for the HMI should be preferred.

743 **How do bundles discover the per-user, bundle-independent location?**

744 The precise location to be used for per-user, bundle-independent data, and the  
745 API to get it, has not been decided.

746 **Is `g_get_home_dir()` bundle-independent?**

747 It is undecided whether the HOME environment variable and `g_get_home_dir()`  
748 should point to `/home/$user`, or to a per-user, per-bundle location. If those point  
749 to a per-user, per-bundle location, then a separate API will need to be provided  
750 by libcanterbury with which a program can access per-user, bundle-independent  
751 data.

752 **Is `g_get_temp_dir()` bundle-independent?**

753 It is undecided whether the TMPDIR environment variable and `g_get_temp_dir()`  
754 should point to `/tmp` as they normally do, or to a per-user, per-bundle location.

755 **Is PICTURES per-user?**

756 Should `G_USER_DIRECTORY_PICTURES` be shared between users and between bundles  
757 like `G_USER_DIRECTORY_MUSIC` and `G_USER_DIRECTORY_VIDEOS`, or should it be per-user  
758 like `$HOME`, or should it be per-user per-bundle like `g_get_user_cache_dir()`?

759 As of Apertis 16.06, it has been implemented as shared, like `G_USER_DIRECTORY_MUSIC`.

760 **What is the scope of DESKTOP, DOCUMENTS, TEMPLATES?**

761 What should the scope of `G_USER_DIRECTORY_DESKTOP`, `G_USER_DIRECTORY_DOCUMENTS`,  
762 `G_USER_DIRECTORY_TEMPLATES` be? Or should we declare these to be unsupported  
763 on Apertis, and set them to the same place as `$HOME` as documented by their  
764 specification?

765 As of Apertis 16.06, these were marked as unsupported and set to be the same  
766 as `$HOME`.

767 **Unresolved implementation questions**

768 **Can we prevent symlink attacks in shared directories?**

769 Can we use AppArmor to prevent the creation of symbolic links in directories  
770 that are shared between users or between bundles, so that applications do not  
771 need to take precautions to avoid writing through a symbolic link, which could  
772 allow one trust domain to make another trust domain overwrite a chosen file  
773 if the writing application is insufficiently careful? We probably cannot use `+`  
774 `permissions` (the “sticky bit”, which activates restricted deletion and symlink  
775 protection), because that would prevent one user from deleting a file created by  
776 another user, which is undesired here.

777 **Should LD\_LIBRARY\_PATH be set?**

778 The Autotools build system (`autoconf`, `automake` and `libtool`) will automatically  
779 configure executables to load libraries built from the same source tree in their  
780 installed locations, using the `DT_RPATH` ELF header, so it is unnecessary to set  
781 `LD_LIBRARY_PATH`.

782 However, we might wish to set `LD_LIBRARY_PATH=/Applications/${bundle_id}/lib`  
783 (or the obvious `/usr/Applications` equivalent) so that app-bundles built with a  
784 non-Automake build system will “just work”.

785 Similarly, we might wish to set `GI_TYPELIB_PATH=/Applications/${bundle_id}/lib/girepository-`  
786 `1.0` for app-bundles that use GObject-Introspection.

## 787 **Alternative designs**

### 788 **Merge static and variable files for store applications**

789 One option that was considered was to separate the read-only parts of built-in  
790 application bundles (in `/usr/Applications`) from the read/write parts (in `/Ap-`  
791 `plications`), but not separate the read-only parts of store application bundles  
792 (in `/Applications`) from the read/write parts (also in `/Applications`).

793 This reduces the number of subvolumes (one subvolume per store bundle instead  
794 of two), but requires additional complexity in the store bundle installer: it would  
795 have to distinguish between the static data directories (`bin`, `share`, etc.) and the  
796 variable data directories (`cache`, `users`, etc.) by name.

### 797 **Add a third subvolume per app-bundle for cache**

798 Conversely, because cache files are not rolled back, we could consider separat-  
799 ing disposable cache files from the other read/write parts; they would not be  
800 subject to snapshots, and during a rollback, the cache subvolume would simply  
801 be deleted and re-created.

### 802 **Each user's files under their `$HOME`**

803 This strategy is not recommended, and is only mentioned here to document why  
804 we have not taken it.

805 The recommendations above keep all users' variable files for a given application  
806 bundle, and any variable files for that bundle that are shared among all users,  
807 together. An alternative design that we could have used would be to keep all of  
808 a user's variable files, across all bundles, in one place (for example their home  
809 directory, `$HOME`).

810 Because store application bundles can be rolled back independently, each user  
811 would need at least one subvolume per store application bundle plus one sub-  
812 volume for built-in application bundles, so that the chosen store application  
813 bundle's data area could be rolled back without affecting other bundles.

814 The reason that this design was rejected is that it scales poorly in some cases, in-  
815 cluding the one that we expect to be most frequent (store app-bundle installation  
816 and uninstallation). While it does require fewer subvolume manipulations than  
817 the recommended design for some operations, those operations are expected to  
818 be rare. To illustrate this, suppose we have 10 built-in bundles, 20 store bundles  
819 and 5 users.

820 If we install, upgrade or remove the store bundle `com.example.MyApp`, which ad-  
821 ditionally has some variable files that are shared between users. With the rec-  
822 ommended design, we only have to perform  $O(1)$  subvolume operations (two  
823 with the recommended design, one if we [Merge static and variable files for store](#)  
824 [applications](#), or three if we [Add a third subvolume per app-bundle for cache](#)).

825 In this alternative design, we would have to perform  $O(\text{number of users})$  sub-  
826 volume operations, in this case 7: one for the bundle's static files, one for its  
827 variable files shared between users, and one per user.

828 Similarly, when we upgrade the platform and we wish to take a snapshot of each  
829 built-in application's data, the recommended design requires us to take 10 snap-  
830 shots (more generally  $O(1)$ , one per built-in bundle), whereas this alternative  
831 requires 50-60 snapshots (more generally  $O(\text{number of users})$ , one per built-in  
832 bundle per user, and zero or one per built-in bundle for non-user-specific data).

833 If we add or delete a user, in the recommended design we would have to perform  
834 31 subvolume operations, or more generally  $O(\text{number of bundles})$ : one per  
835 store or built-in bundle, plus one extra operation for non-bundle-specific data.  
836 In this alternative we would need a minimum of 22 subvolume operations, or  
837 more generally  $O(\text{number of store bundles})$ : one per store bundle, one for all  
838 built-in bundles together, and one for non-bundle-specific data.

839 If we perform a data reset without uninstalling store app bundles, the recom-  
840 mended design would require at least 30 subvolume deletions (one per applica-  
841 tion bundle), whereas this design would require at least 150 subvolume deletions  
842 (one per bundle per user).

#### 843 **System integration links for services**

844 It would be technically possible to install user-services (services that run as a  
845 particular user, similar to Tracker) in an application bundle, and register them  
846 with the wider system via system integration links ( [System integration links](#)  
847 [for built-in applications](#), [Store application system integration links](#)) pointing to  
848 their systemd user services and D-Bus session services.

849 We recommend that this is not done, because general systemd user services are  
850 powerful and have a global effect. Instead, we recommend that per-app-bundle  
851 user-services (agents) are implemented by having the application manager (Can-  
852 terbury) generate a carefully constrained subset of service file syntax from the  
853 entry point metadata.

#### 854 **System services in app-bundles**

855 It would be technically possible to install system services (services that do not  
856 run as a specific user) in an application bundle, registering them via system  
857 integration links as above.

858 We recommend that this is not done, because system services are extremely  
859 powerful and can have extensive privileges. Instead, system services should be  
860 part of the [platform](#)<sup>35</sup> layer.

---

<sup>35</sup><https://martyn.pages.apertis.org/apertis-website/glossary/#platform>



861 **Appendix: application layout in Apertis 15.09**

862 Sudoku is one example of a store application bundle. Its source code is not  
863 currently public. `xyz` is used here to represent the common prefix for an Apertis  
864 variant. The layout of the store application bundle looks like this:

```
865 /appstore/  
866     store.json  
867     store.sig  
868     xyz-sudoku_config.tar  
869         xyz-sudoku_config/  
870             xyz-sudoku.png  
871             xyz-sudoku_manifest.json  
872 /xyz-sudoku.tar  
873     xyz-sudoku/  
874         bin/  
875             xyz-sudoku  
876         share  
877             glib-2.0  
878                 schemas  
879                     com.app.xyz-sudoku.gschema.xml  
880                     com.app.xyz-sudoku.enums.xml  
881                     gschemas.compiled  
882             background.png  
883             icon_sudoku.png  
884             (more graphics)
```

885 The manifest indicates that `/xyz-sudoku.tar` is expected to be unpacked into `/Ap-`  
886 `plications`, leading to filenames like `/Applications/xyz-sudoku/bin/xyz-sudoku`.

887 [Frampton](#)<sup>36</sup> is an example of a built-in application bundle shipped in 15.09. Its  
888 layout is as follows:

```
889 /usr/  
890     Applications/  
891         frampton/  
892             bin/  
893                 frampton  
894                 frampton-agent  
895                 test-frampton-agent  
896             lib/  
897                 libframptonagentiface.so{,.0,.0.0.0}  
898             share/  
899                 IconBig_Music.png  
900                 icon_albums_inactive.png  
901                 ...  
902                 artist-album-views/
```

---

<sup>36</sup><https://gitlab.apertis.org/appfw/frampton>

```

903         DetailView.json
904         ...
905     glib-2.0/
906         schemas/
907             com.app.frampton-agent.gschema.xml
908         ...
909     locale/
910         de/
911         ...
912 /Applications/
913     Frampton/
914         app-data/
915             Internal/
916                 FramptonAgent.db
917     frampton/
918         app-data/
919             (empty)

```

920 Issues with the application filesystem layout in these examples:

- 921 • There is no “manifest” file with metadata for the built-in application bundle as a whole.
- 922 • The “manifest” files for entry points in both store and built-in applications are GSettings schema XML, which is not how GSettings is designed to be used. They are also incorrectly namespaced: the app developer presumably does not own `app.com`. We should use `org.apertis.*` for Apertis components, `{com,net,org}.example.*` for developer examples, and a vendor’s name elsewhere.
- 923 • There is no separation between users. “user” owns all of `/Applications`.
- 924 • Frampton’s app bundle ID is ambiguous: is it Frampton or frampton? We should choose exactly one ID, and make the AppArmor profile forbid using the other.
- 925 • Frampton’s app bundle ID is not namespaced. The [Applications design document](#)<sup>37</sup> specifies use of a [reversed domain name](#)<sup>38</sup> such as `org.apertis.Frampton`.
- 926 • Similarly, Sudoku’s app bundle ID is not namespaced.
- 927 • There is no well-known location for apps’ icons: Frampton places its icons in `/usr/Applications/frampton/share/`, but other apps use `/usr/Applications/$bundle_id/share/images`, requiring `mildenhall-launcher` to be allowed to read both locations.
- 928 • There is no well-known location into which Newport may download files.

<sup>37</sup><https://martyn.pages.apertis.org/apertis-website/concepts/applications/>

<sup>38</sup><https://martyn.pages.apertis.org/apertis-website/glossary/#reversed-domain-name>

## 942 Appendix: comparison with other systems

### 943 Desktop Linux (packaged apps)

944 There are many possibilities, but a common coding standard looks like this:

- 945 • Main programs are installed in `$bindir` (which is set to `/usr/bin`)
- 946 • Supporting programs are installed in `$libexecdir` (which is set to either  
947 `/usr/libexec` or `/usr/lib`), often in a subdirectory per application package
- 948 • Public shared libraries are installed in `$libdir` (which is set to either  
949 `/usr/lib` or `/usr/lib64` or `/usr/lib/$architecture`)
  - 950 – Plugins are installed in a subdirectory of `$libdir`
  - 951 – Private shared libraries are installed in a subdirectory of `$libdir`
- 952 • `.gresource` resource bundles (and any resource files that cannot use GRe-  
953 source) are installed in `$datadir`, which is set to `/usr/share`
- 954 • System-level configuration is installed in a subdirectory of `$sysconfdir`,  
955 which is set to `/etc`
- 956 • System-level variable data is installed in `$localstatedir/lib/$package` and  
957 `$localstatedir/cache/$package`, with `$localstatedir` set to `/var`
- 958 • There is normally no technical protection between apps, but per-user vari-  
959 able data is stored according to the [XDG Base Directory specification](#)<sup>39</sup>  
960 in:
  - 961 – `$XDG_CONFIG_HOME/$package`, defaulting to `/home/$username/.config/$package`,  
962 where `$username` is the user’s login name and `$package` is the short  
963 name of the application or package
  - 964 – `$XDG_DATA_HOME/$package`, defaulting to `/home/$username/.local/share/$package`
  - 965 – `$XDG_CACHE_HOME/$package`, defaulting to `/home/$username/.cache/$package`
- 966 • The user’s home directory, normally `/home/$username`, is shared between  
967 apps but private to the user
  - 968 – It is usually technically possible for one app to alter another app’s  
969 subdirectories of `$XDG_CONFIG_HOME` etc.
- 970 • There is no standard location that can be read and written by all users,  
971 other than temporary directories which are not intended to be shared

972 [Debian Policy §9.1 “File system hierarchy”](#)<sup>40</sup> describes the policy followed on  
973 Debian and Ubuntu systems for non-user-specific data. It references the [Filesys-  
974 tem Hierarchy Standard, version 2.3](#)<sup>41</sup>.

975 Similar documents:

- 976 • The [Filesystem Hierarchy Standard, version 3.0](#)<sup>42</sup> has not yet been  
977 adopted by Debian Policy.
- 978 • The [GNU Coding Standards](#)<sup>43</sup> use a similar layout by default.

---

<sup>39</sup><http://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html>

<sup>40</sup><https://www.debian.org/doc/debian-policy/ch-opersys.html#s9.1>

<sup>41</sup><http://www.pathname.com/fhs/pub/fhs-2.3.html>

<sup>42</sup>[http://refspecs.linuxfoundation.org/FHS\\_3.0/fhs/index.html](http://refspecs.linuxfoundation.org/FHS_3.0/fhs/index.html)

<sup>43</sup>[https://www.gnu.org/prep/standards/html\\_node/Directory-Variables.html#Directory-Variables](https://www.gnu.org/prep/standards/html_node/Directory-Variables.html#Directory-Variables)

- 979     • [systemd's proposals for file hierarchy](#)<sup>44</sup> have been partially adopted by  
980     Linux distributions.

## 981 **Flatpak**

982 Autoconf/Automake software in a [Flatpak](#)<sup>45</sup> package is built with `--prefix=/app`,  
983 and the static files of the app are mounted at `/app` inside the sandbox. Each  
984 Flatpak has its own private view of the filesystem inside its sandbox, so this  
985 does not lead to conflict over ownership of `/app` as might be expected.

- 986     • Main programs are installed in `$bindir`, which is `/app/bin`
- 987     • Supporting programs are installed in `$libexecdir`, which is `/app/libexec`
- 988     • Private shared libraries are installed in `$libdir`, which is `/app/lib`, or in a  
989     subdirectory
  - 990         – Plugins are installed in a subdirectory of `$libdir`
- 991     • Static resources are embedded using `GResource`, installed in `/app/share` as  
992     a `.gresource` resource bundle, or installed in `/app/share` as plain files
- 993     • System-level configuration is installed in `/app/etc`
- 994     • Per-user variable data is stored in `/home/$username/.var/app/$app_id/{data,config,cache}`,  
995     which are bind-mounted into the app's filesystem namespace, with the  
996     `XDG_{DATA,CONFIG,CACHE}_HOME` environment variables set to point at those  
997     locations
- 998     • Shared variable data is stored in `/var/lib/$app_id`, `/var/cache/$app_id`.  
999     *(How widely shared is this really?)*

1000 Integration files (systemd units, D-Bus services, etc.) are said to be *exported*  
1001 by the Flatpak, and they are linked into `$XDG_DATA_HOME/flatpak/exports` or  
1002 `/var/lib/flatpak/exports` outside the sandbox.

1003 *Runtimes* (sets of libraries) are mounted at `/usr` inside the sandbox.

## 1004 **Android**

- 1005     • System app packages (the equivalent of our [built-in application bundles](#)<sup>46</sup>)  
1006     are stored in `/system/app/$package.apk`
- 1007     • Normal app packages (the equivalent of our [store application bundles](#)<sup>47</sup>)  
1008     are stored in `/data/app/$package.apk`
- 1009     • Private shared libraries and plugins (and, technically, any other supporting  
1010     files) are automatically unpacked into `/data/data/$package/lib/` by the OS
- 1011     • Resource files are loaded from inside the `.apk` file (analogous to `GResource`)  
1012     instead of existing as files in the filesystem
- 1013     • Per-user variable data is stored in `/data/data/$package/` on single-user de-  
1014     vices

---

<sup>44</sup><http://www.freedesktop.org/software/systemd/man/file-hierarchy.html>

<sup>45</sup><http://flatpak.org/>

<sup>46</sup><https://martyn.pages.apertis.org/apertis-website/glossary/#built-in-application-bundle>

<sup>47</sup><https://martyn.pages.apertis.org/apertis-website/glossary/#store-application-bundle>

- 1015 • Per-user variable data is stored in `/data/user/$user/$package/` on multi-
- 1016 user devices
- 1017 • There is no location that is private to an app but shared between users.
- 1018 The closest equivalent is `/sdcard/$package`, which is conventionally only
- 1019 used by the app `$package`, but is technically accessible to all apps.
- 1020 • There is no location that is shared between apps but private to a user.
- 1021 • `/sdcard` is shared between apps and between users. Large data files such
- 1022 as music and videos are normally stored here.

### 1023 **systemd “revisiting Linux systems” proposal**

1024 The [authors of systemd propose a structure like this](#)<sup>48</sup>. At the time of writing,

1025 no implementations of this idea are known.

- 1026 • The static files of application bundles are installed in a subvolume named
- 1027 `app:$bundle_id:$runtime:$architecture:$version`, where:
  - 1028 – `$bundle_id` is a reversed domain name identifying the app bundle itself
  - 1029 – `$runtime` identifies the set of runtime libraries needed by the applica-
  - 1030 tion bundle (in our case it might be `org.apertis.r15_09`)
  - 1031 – `$architecture` represents the CPU architecture
  - 1032 – `$version` represents the version number
- 1033 • That subvolume is mounted at `/opt/$bundle_id` in the app sandbox. The
- 1034 corresponding runtime is mounted at `/usr`.
- 1035 • User-specific variable files are in a subvolume named, for example,
- 1036 `home:user:1000:1000` which is mounted at `/home/user`.
- 1037 • System-level variable files go in `/etc` and `/var` as usual.
- 1038 • There is currently no concrete proposal for a trust boundary between apps:
- 1039 all apps are assumed to have full access to `/home`.
- 1040 • There is no location that is private to an app but shared between users.
- 1041 • There is no location that is shared between apps and between users, other
- 1042 than removable media.

### 1043 **References**

- 1044 • [Applications design document](#)<sup>49</sup> (v0.5.4 used)
- 1045 • [Multimedia design document](#)<sup>50</sup> (v0.5.4 used)
- 1046 • [Security design document](#)<sup>51</sup> (v1.1.3 used)
- 1047 • [System Update and Rollback design document](#)<sup>52</sup> (v1.6.2 used)

---

<sup>48</sup><http://0pointer.net/blog/revisiting-how-we-put-together-linux-systems.html>

<sup>49</sup><https://martyn.pages.apertis.org/apertis-website/concepts/applications/>

<sup>50</sup><https://martyn.pages.apertis.org/apertis-website/concepts/multimedia/>

<sup>51</sup><https://martyn.pages.apertis.org/apertis-website/concepts/security/>

<sup>52</sup><https://martyn.pages.apertis.org/apertis-website/concepts/system-updates-and-rollback/>