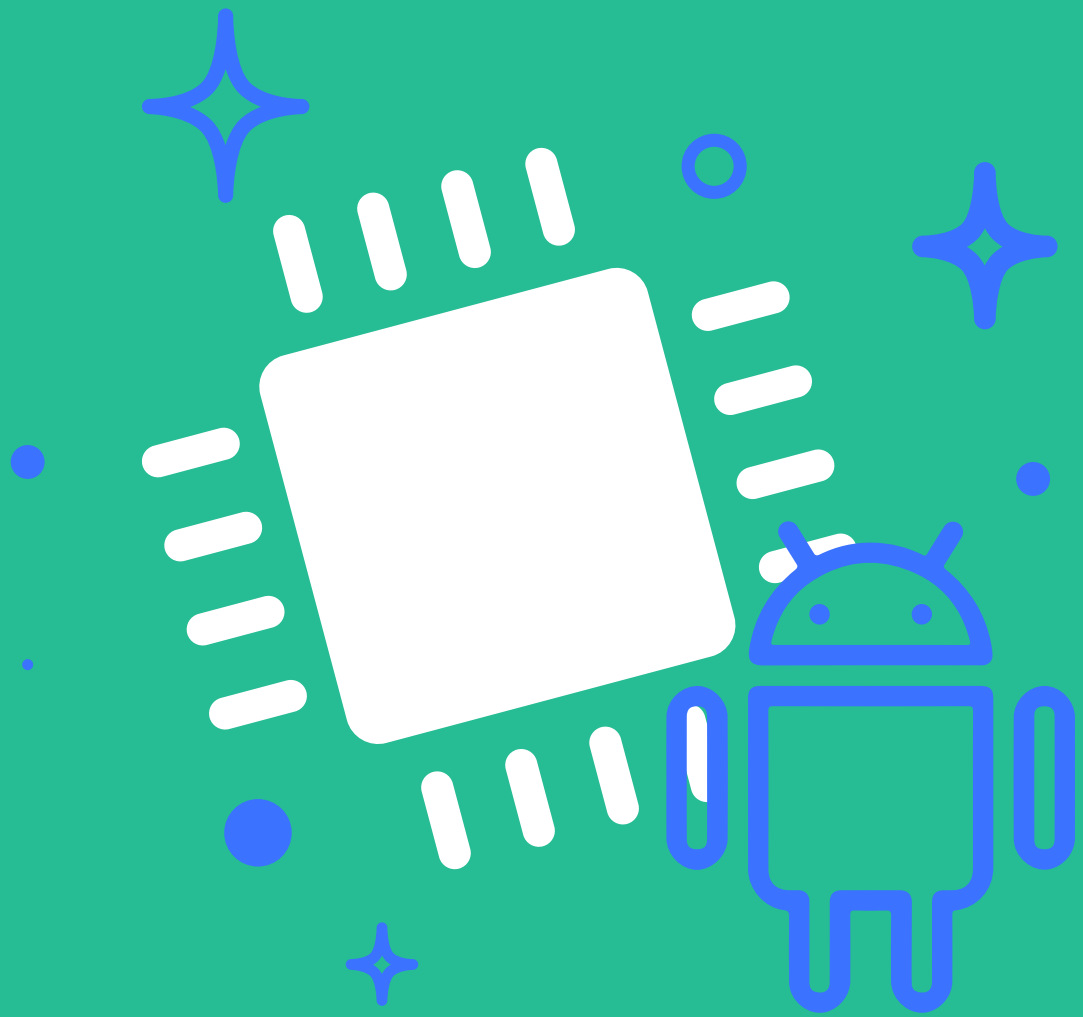
ANDROID POUR L'EMBARQUÉ

Le meilleur des solutions open source



SMILE

I.T IS OPEN

PRÉAMBULE

SMILE

Smile est une **société d'ingénieurs experts** dans la mise en œuvre de **solutions open source** et l'intégration de systèmes appuyés sur l'open source. Smile est membre de l'**APRIL**, l'association pour la promotion et la défense du logiciel libre, du **PLOSS** – le réseau des entreprises du Logiciel Libre en Ile-de-France et du **CNLL – le conseil national du logiciel libre**.

Smile compte plus de 1200 collaborateurs dans le monde ce qui en fait le premier intégrateur français et européen de solutions open source.

Depuis 2000, environ, **Smile mène une action active de veille technologique** qui lui permet de découvrir les produits les plus prometteurs de l'open source, de les qualifier et de les évaluer, de manière à proposer à ses clients les produits les plus aboutis, les plus robustes et les plus pérennes. Cette démarche a donné lieu à **toute une gamme de *livres blancs*** couvrant différents domaines d'application. La gestion de contenus (2004), les portails (2005), la business intelligence (2006), la virtualisation (2007), la gestion électronique de documents (2008), les PGIs/ERPs (2008), les VPN open source (2009), les Firewall et Contrôle de flux (2009), les Middleware orientés messages (2009), l'ecommerce et les Réseaux Sociaux d'Entreprise (2010) et plus récemment, le Guide de l'open source et NoSQL (2011). Chacun de **ces ouvrages présente une sélection des meilleures solutions open source** dans le domaine considéré, leurs qualités respectives, ainsi que des retours d'expérience opérationnels.

Au fur et à mesure que des solutions open source solides gagnent de nouveaux domaines, Smile sera présent pour proposer à ses clients d'en bénéficier sans risque. Smile apparaît dans le paysage informatique français comme **le prestataire intégrateur de choix** pour **accompagner** les plus grandes entreprises dans l'adoption des meilleures solutions open source.

Ces dernières années, Smile a également étendu la gamme des services proposés. Depuis 2005, un département consulting accompagne nos clients, tant dans les phases d'avant-projet, en recherche de solutions, qu'en accompagnement de projet. Depuis 2000, Smile dispose d'un studio graphique, devenu en 2007 Smile Digital – agence interactive, proposant outre la création graphique, une expertise e-marketing, éditoriale, et interfaces riches. Smile dispose aussi d'une agence spécialisée dans la TMA (support et l'exploitation des applications) et d'un centre de formation complet, Smile Training. **Enfin, Smile est implanté à Paris, Lille, Lyon, Grenoble, Nantes, Bordeaux, Marseille, Toulouse et Montpellier. Et présent également en Suisse, au Benelux, en Ukraine et au Maroc**.

Quelques références

SMILE est fier d'avoir contribué, au fil des années, aux plus grandes réalisations Web françaises et européennes. Vous trouverez ci-dessous quelques clients nous ayant adressé leur confiance.

Sites Internet

EMI Music, Salon de l'Agriculture, Mazars, Areva, Société Générale, Gîtes de France, Patrice Pichet, Groupama, Eco-Emballage, CFnews, CEA, Prisma Pub, Véolia, NRJ, JCDecaux, 01 Informatique, Spie, PSA, Boiron, Larousse, Dassault Systèmes, Action Contre la Faim, BNP Paribas, Air Pays de Loire, Forum des Images, IFP, BHV, ZeMedical, Gallimard, Cheval Mag, Afssaps, Benetaux, Carrefour, AG2R La Mondiale, Groupe Bayard, Association de la Prévention Routière, Secours Catholique, Canson, Veolia, Bouygues Telecom, CNIL…

Portails, Intranets et Systèmes d'Information

HEC, Bouygues Telecom, Prisma, Veolia, Arjowiggins, INA, Primagaz, Croix Rouge, Eurosport, Invivo, Faceo, Château de Versailles, Eurosport, Ipsos, VSC Technologies, Sanef, Explorimmo, Bureau Veritas, Région Centre, Dassault Systèmes, Fondation d'Auteuil, INRA, Gaz Electricité de Grenoble, Ville de Niort, Ministère de la Culture, PagesJaunes Annonces…

E-Commerce

Krys, La Halle, Gibert Joseph, De Dietrich, Adenclassifieds, Macif, Furet du Nord, Gîtes de France, Camif Collectivité, GPdis, Projectif, ETS, Bain & Spa, Yves Rocher, Bouygues Immobilier, Nestlé, Stanhome, AVF Périmédical, CCI, Pompiers de France, Commissariat à l'Energie Atomique, Snowleader, Darjeeling…

ERP et Décisionnel

Veolia, La Poste, Christian Louboutin, Eveha, Sun'R, Home Ciné Solutions, Pub Audit, Effia, France 24, Publicis, iCasque, Nomadvantage, Gets, Nouvelles Frontières, Anevia, Jus de Fruits de Mooréa, Espace Loggia, Bureau Veritas, Skyrock, Lafarge, Cadremploi, Meilleurmobile.com, Groupe Vinci, IEDOM (Banque de France), Carrefour, Jardiland, Trésorerie Générale du Maroc, Ville de Genève, ESCP, Sofia, Faiveley Transport, INRA, Deloitte, Yves Rocher, ETS, DGAC, Generalitat de Catalunya, Gilbert Joseph, Perouse Médical…

Gestion documentaire

Primagaz, UCFF, Apave, Géoservices, Renault F1 Team, INRIA, CIDJ, SNCD, Ecureuil Gestion, CS informatique, Serimax, Véolia Propreté, NetasQ, Corep, Packetis, Alstom Power Services, Mazars…

Infrastructure et Hébergement

Agence Nationale pour les Chèques Vacances, Pierre Audoin Consultants, Rexel, Motor Presse, OSEO, Sport24, Eco-Emballage, Institut Mutualiste Montsouris, ETS, Ionis, Osmoz, SIDEL, Atel Hotels, Cadremploi, SETRAG, Institut Français du Pétrole, Mutualité Française…

Systèmes industriels et embarqués

Groupe THALES, NEXTER, Airbus, Sagemcom, Sagem Défense, Stago, ST Microeletronics, Intel, Dassault Aviation, RATP, Coyote, Coved, Groupe PSA, Sigfox, Stago

Consultez nos références, en ligne, à l’adresse : http://www.smile.fr/References.

Ce livre blanc

Ce livre blanc est consacré à l’utilisation du système Android pour des applications industrielles et embarquées. Il ne traite donc pas du développement d’applications Android (il existe de nombreuses publications pour cela) mais plutôt de l’adaptation d’Android à un nouveau matériel (carte mère mais également interfaces industrielles et/ou protocoles de communication associés).

Après un bref historique et quelques rappels sur les principes d'un système embarqué nous décrirons dans quels cas Android peut être utilisé en remplacement de « Linux embarqué » qui avait fait l’objet d’un livre blanc publié par Smile au printemps 2016. Une partie plus technique traitera ensuite de l’architecture d’Android et des techniques d’adaptation des sources fournies par Google sous le nom *AOSP* pour *Android Open Source Project*. Nous terminerons par un chapitre traitant des marchés actuels et à venir pour l’utilisation d’Android dans l’industrie et l'internet des objets (IoT).

Ce livre blanc a été rédigé par Pierre Ficheux, Directeur technique de Smile-ECS (Embedded & Connected Systems). Le dernier chapitre traitant des marchés actuels et à venir a été rédigé par Cédric Ravalec, Responsable de l'offre « Embarqué et IoT » chez Smile.

Sommaire

Introduction aux systèmes embarqués

Par définition, un système embarqué est l'association de matériel (un calculateur) et de logiciel. Contrairement à l'informatique classique (poste de travail ou serveur), le système est dédié à un ensemble fini de fonctionnalités et il en est de même pour le logiciel. Historiquement les premiers domaines d'application furent limités à l'armement et au spatial pour lesquels les coûts des programmes sont très élevés. Une des difficultés importantes de l'époque était l'absence de microprocesseur car le 4004 - premier microprocesseur disponible commercialement - fut créé par Intel (seulement) en 1971. Dans le domaine spatial on s'accorde à dire que le premier système embarqué fut l'*Apollo Guidance Computer* [1] créé en 1967 par le MIT pour la mission lunaire Apollo. Ce dernier disposait de 36K mots de ROM, 2K mots de RAM et fonctionnait à la fréquence de 2 MHz. De nos jours on peut simuler le comportement de ce calculateur grâce à une page web animée par du langage Javascript ! Dans le cas des applications militaires, on évoque le calculateur D-17B, système de guidage pour le missile LGM-30 datant du début des années 60 [2].

Il n'était pas question à l'époque d'évoquer la notion de *système d'exploitation embarqué* et l'on parlait simplement de *logiciel embarqué* écrit en langage machine. Par contre, le programme Apollo utilisait de nombreux calculateurs IBM/360 au sol et le logiciel le plus complexe de la mission occupait 6 Mo. Sur cet IBM/360, la NASA utilisait une version modifiée (temps réel) de l'OS/360 nommée RTOS/360 [3].

REMARQUE

Encore de nos jours, même si l'on utilise désormais des langages évolués comme C ou Ada, certains systèmes embarqués critiques sont toujours dépourvus d'OS. On parle alors de logiciel *bare metal*.

A partir des années 80, l'évolution technologique permet à de nombreux systèmes d'exploitation temps réel (RTOS pour Real Time Operating System) de fonctionner sur des CPU du commerce, la plupart des RTOS étant commercialisés par des éditeurs spécialisés. Nous pouvons citer VRTX (1981) édité par Mentor Graphics et célèbre pour être utilisé dans le télescope Hubble, VxWorks (1987) édité par Wind River et LynxOS (1986) édité par Lynx Software. VxWorks est toujours très utilisé dans les industries sensibles comme le spatial. Il fut entre autres choisi pour la sonde spatiale *Pathfinder* (1996) ainsi que pour la mission *Curiosity* lancée en 2011 et toujours active. Dans le domaine du logiciel libre, *RTEMS* est également fréquemment choisi par la NASA ou l'agence européenne (ESA). Ce système diffusé sous licence GPL est utilisé pour le système de communication de Curiosity et de nombreuses autres missions [4].

A partir des années 2000, l'utilisation des systèmes embarqués ne se limite plus aux applications industrielles ni donc aux RTOS ce qui permet l'utilisation d'un système comme GNU/Linux (qui par défaut n'a pas de capacités temps réel) dans ce domaine d'activité. GNU/Linux est désormais présent dans 95 % des boîtiers d'accès à Internet, décodeur TV (set-top box), sans compter le « noyau » du système Android qui est le sujet de ce livre blanc. De part ses capacités graphiques, Android est désormais utilisé dans plusieurs boîtiers d'accès à la télévision numérique basés sur Android TV [9] qui sera évoqué à la fin du livre.

Plus récemment des alliances d'acteurs de l'industrie automobile ont permis la mise en place de projets comme GENIVI [5] ou Automotive Grade Linux [6] (basés sur un GNU/Linux) dont le but est de créer des composants logiciels compatibles pour les applications *IVI* (In Vehicle Infotainment). Android n'est pas en reste dans le domaine après la création en janvier 2014 de l'Open Automotive Alliance [7] puis la sortie quelques mois plus tard du système Android Auto [8] dédié à l'automobile et d'ores et déjà disponible sur plusieurs modèles des membres de l'alliance.

Historique Android

L'origine d'Android est la société Danger Inc. Fondée en 1999 par Andy Rubin, ancien employé d'Apple. En 2002 la société dévoile un téléphone mobile très innovant pour l'époque nommé *Hiptop* et connu également sous le nom *T-Mobile Sidekick*. Malgré ses capacités (celles d'un PDA), ce téléphone est un échec commercial et Andy Rubin est évincé de la société. Il fonde alors la société Android Inc. en 2004 dans le but de créer une plate-forme ouverte pour la téléphonie (Open Handset Alliance).

La société Android Inc. est acquise par Google en 2005 et l'alliance est créée en 2007 en rassemblant des fabricants de matériels, de composants électroniques, des éditeurs de logiciel et des opérateurs.

Figure 1. Le téléphone Hiptop

Android est avant tout un système d'exploitation (partiellement) ouvert basé sur un noyau Linux mais dont la partie visible (le framework) est écrite en langage Java. La première version commerciale d'Android est publiée en septembre 2008 et fonctionne sur un téléphone HTC Dream. Une version 1.1 est publiée en 2009 puis la version 1.5 (Cupcake) la même année. Dès lors chaque version porte le nom d'un dessert (Donut, Eclair, Froyo, etc.) jusqu'à la dernière version à ce jour (version 7, Nougat). Pour chaque version publiée, Google fournit les sources au travers du projet AOSP (Android Open Source Project) [10]. AOSP représente la version « officielle » d'Android fonctionnant sur les cibles de référence de Google (téléphones et tablettes Nexus et aujourd'hui la gamme Pixel). Les autres fabricants adaptent les sources d'AOSP afin de créer une version alternative (un *fork*) dédiée à leur matériel.

Il faut cependant noter que des composants binaires et/ou propriétaires sont également nécessaires au fonctionnement réel d'Android, en particulier des pilotes graphiques accélérés, les pilotes d'accès au composant radio (RIL pour *Radio Interface Layer*) ou bien l'accès aux GMS (Google Mobile Services) comme Gmail ou Google Maps.

A ce jour, Android est une réussite sans précédent sur son marché initial de la téléphonie puisqu'à fin 2016 on estime à 88 % sa part de marché contre 12 % pour iOS (mais il fonctionne uniquement sur iPhone donc il faut relativiser!) [14].

En plus du système initial, Google fournit la version « Wear » destinée – entre autres – aux montres connectées dont la version 2.0 est juste disponible ainsi que l'OS « Things » (en version *developer preview 2*) qui remplace « Brillo » pour les objets connectés. Dans les deux cas les dernières versions – au moment de la rédaction de ce livre - datent de février 2017. Un paragraphe sera consacré à Android Things dans la partie technique ainsi que dans le dernier chapitre concernant les marchés.

Android pour les systèmes embarqués

Nous avons vu dans l'introduction une description de ce qu'est un système embarqué puis un « OS » embarqué (VxWorks, LynxOS, RTEMS, etc.) ces derniers étant très éloignés des fonctionnalités d'Android en particuliers sur les capacités temps réel (absente sur Android), l'empreinte mémoire utilisée sur la cible (plusieurs centaines de Mo, voire plus de 1 Go pour Android) sans compter l'espace utilisé pour les sources AOSP (plusieurs dizaines de Go). Plus encore que GNU/Linux, Android est donc réservé à des équipements dotés de bonnes capacités matérielles (CPU 32 ou 64 bits, 1 Go ou plus de RAM et plusieurs dizaines de Go de mémoire flash). Ces caractéristiques sont celles disponibles sur les téléphones et tablettes actuelles.

Dans le cas d'un système autre qu'un téléphone ou une tablette – voire un système industriel – l'utilisation d'Android est donc justifiée lorsque l'application nécessite des fonctionnalités proches de celles des cibles initiales, en l'occurrence une interface graphique évoluée ainsi qu'une connectivité réseau (Wi-Fi, Bluetooth, etc.). L'utilisation de la version standard d'Android dans le cas d'un système sans interface graphique (parfois nommé « headless Android ») nous paraît totalement dénuée de sens même si elle est parfois évoquée (et Things comblera probablement ce vide).

Comparaison avec GNU/Linux

La comparaison avec GNU/Linux s'impose car Android est également basé sur un noyau Linux, bien que les sources de ce noyau ne soient pas directement fournies avec AOSP. De plus, GNU/Linux est le plus souvent utilisable pour des applications similaires à celles d'Android car il nécessite également un matériel puissant et une empreinte mémoire importante (même si elle est inférieure à celle utilisée par Android). En dehors de GNU/Linux, les environnements Microsoft de type Windows CE (sans parler du malheureux Windows Phone) sont en passe d’être totalement marginalisés par l’arrivée d’Android (qui peut prétendre proposer une meilleure application GPS que celle de Waze, acquise par Google en 2013 ?).

Dans de nombreux cas, le choix du système cible se fera entre GNU/Linux « embarqué » et Android ce qui justifie cette comparaison sur plusieurs critères.

Modèle de développement et licences

Au niveau du modèle de développement, AOSP n’est pas un modèle collaboratif comme peut l’être le noyau Linux ou l’environnement de construction Yocto [15] (utilisé sur les projets GENIVI et AGL cités dans l’introduction). Les développements sont en effet réalisés en interne par Google pour AOSP qui est ensuite adapté par les constructeurs de matériel ou des prestataires spécialisés.

Comme nous l’avons vu dans l’introduction, Android utilise des composants binaires (certains pilotes propriétaires). Il n’est donc pas totalement open source et c'est encore moins du logiciel libre comme l'explique la FSF (Free Software Foundation) [11]. En effet, mis à part pour le noyau Linux, la licence Android (utilisée dans AOSP) est celle du projet Apache (ASL version

2.0) beaucoup moins contraignante que la licence GPL. Le fait n'est pas très étonnant sachant que le marché de la téléphonie est peu enclin au partage et dispose d'une forte culture propriétaire (aucun téléphone totalement « libre » n'a pour l'instant pu s'imposer). Le responsable de la gouvernance open source de Google affirma il y a quelques années lors d'une conférence dans les locaux de l'entreprise à Paris que « Google is not an open source company » (et on le croit assez facilement). Le fait de dépendre de composants propriétaires provoqua la fureur de Jean-Baptiste Queru (dit JBQ) célèbre leader de l'équipe AOSP qui quitta le projet en 2013 (puis Google quelques mois plus tard) [16].

« Well, I see that people have figured out why I'm quitting AOSP.

There's no point being the maintainer of an Operating System that can't boot to the home screen on its flagship device for lack of GPU support, especially when I'm getting the blame for something that I don't have authority to fix myself and that I had anticipated and escalated more than 6 months ahead. »

Architecture et HAL

Android n'est cependant pas une « distribution Linux » même si les couches basses du système (celles qui ne sont pas écrites en Java) ressemblent fortement à celles de GNU/Linux. Il en est de même pour la procédure de démarrage du système. L'architecture d'Android est cependant plus complexe que celle de GNU/Linux de part l'utilisation du langage Java pour le framework et donc les applications. La figure ci-après met en évidence les différentes couches du système et les différences avec GNU/Linux. La principale différence est l'existence d'une couche d'abstraction (HAL). La partie supérieure correspond à du code Java, la partie intermédiaire à du code C/C++ et la dernière à l'espace du noyau Linux. Si l'on prend l'exemple simple d'une application accédant à une led, on peut énoncer les principes suivants :

- L'application est écrite en langage Java (LedDemo.java sur le schéma)
- L'application utilise les ressources (Wi-Fi, Bluetooth, luminosité écran, etc.) via des « services » Android (LedService.java sur le schéma)
- Les services utilisent des bibliothèques (en C/C++) indépendantes de la cible (libled_runtime.so)
- Pour chaque cible matérielle on met en place une implémentation dépendante de la cible (led.default.so, led.<cible 1>.so, led.<cible 2>.so, etc.). Cette bibliothèque réalise des accès au noyau Linux qui assure l'interface avec le matériel.

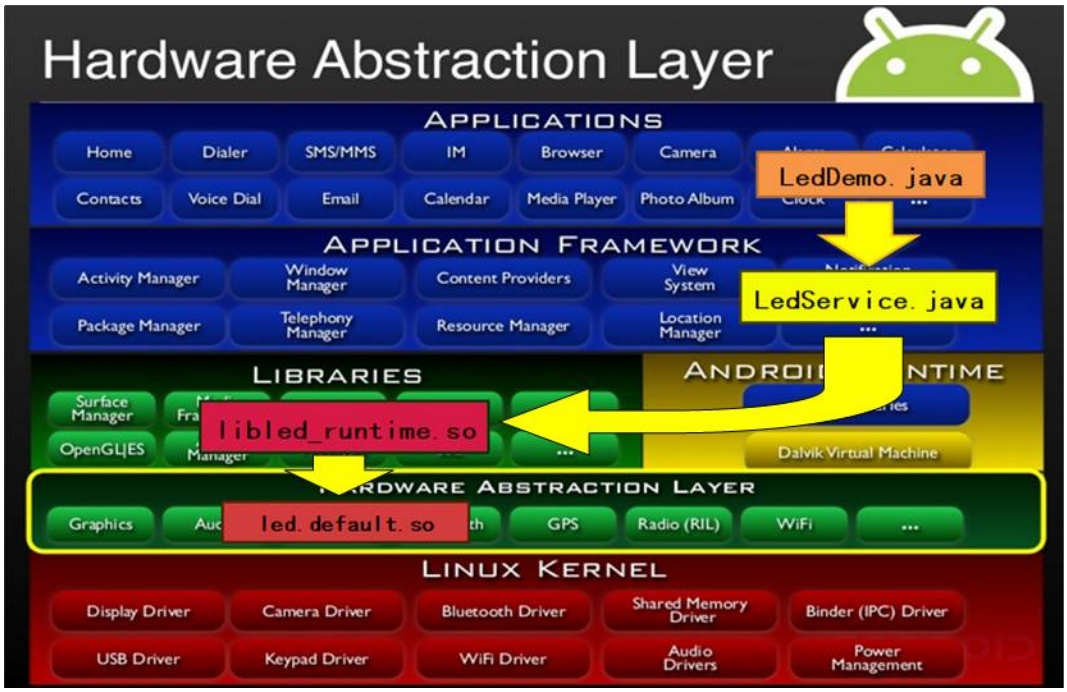


Figure 2. Architecture Android

Le choix d'utilisation de la HAL permet également de limiter les problèmes de licences puisque la propriété intellectuelle liée à l'utilisation d'un composant matériel est également basée sur une bibliothèque située en espace utilisateur et non exclusivement sur un pilote en espace noyau (qui utilise exclusivement la licence GPL).

Fonctionnalités graphiques

Concernant les fonctionnalités graphiques, GNU/Linux est dérivé d'UNIX qui fut conçu à la fin des années 60 et donc fonctionnait en mode texte. Il existe bien entendu quelques environnements graphiques pour UNIX et GNU/Linux mais ils ne sont pas aussi simples à aborder que dans le cas d'Android pour lequel le framework graphique est totalement intégré de part son utilisation native dans la téléphonie. Une application Android est (presque) toujours graphique et passe le plus souvent par l'utilisation de l'environnement de développement *Android Studio*.

Coté GNU/Linux, Qt est l'environnement graphique le plus connu et utilise le langage C++ ou plus récemment le QML (Qt Meta Language) [12]. Il a l'avantage d'être multiplate-forme et dispose désormais d'une cible Android – et iOS. Dans le cas d'une cible plus contrainte en mémoire, on peut également utiliser EFL (Enlightenment Foundation Library) [13] mais cette bibliothèque ne dispose pas d'un outil de construction d'interface (comme il en existe pour Qt ou Android). La disponibilité d'outils et d'un langage de programmation relativement simple (Java) n'est pas le seul avantage d'Android. Un autre point important est le grand nombre d'applications et de bibliothèques « métier » disponibles alors qu'elles ne le sont pas forcément pour GNU/Linux. La facilité et rapidité de développement pour cette plate-forme y est certainement pour beaucoup. Nous avons eu récemment l'exemple d'un projet industriel

initialement prévu pour GNU/Linux mais qui a finalement été développé sur Android car certains composants métier existaient déjà, alors que ce n'était pas le cas pour GNU/Linux. Le système étant graphique par nature, on comprend tout à fait qu'il puisse concurrencer GNU/Linux sur le terrain des applications de type set-top box ou environnements IVI dédiées à l'automobile (GENIVI et AGL pour GNU/Linux).

Construction du système

Le système étant conçu pour la téléphonie (et développé uniquement par Google), la technique de construction – et donc de modification - du système est donc radicalement différente de celle utilisée pour GNU/Linux. Le système est fourni sous la forme d'une image (on parle de « ROM ») et la mise à jour s'effectue en remplaçant l'intégralité de cette image. Par contre les applications Android disposent d'un système de gestion de paquets que l'on peut assimiler à ceux dont on dispose sur GNU/Linux (RPM sur Fedora, DEB sur Debian/Ubuntu). A cela il faut ajouter le controversé « Google Play Services » qui permet à Google d'effectuer des modifications et mises à jour sur les quelques milliards d'appareils connectés. Nous reviendrons sur ce point plus tard.

La construction du système (la « ROM ») s'effectue grâce un outil de construction dédié à AOSP. Cet outil assez rustique est basé sur l'utilitaire GNU Make ainsi que sur l'interpréteur de commande BASH de GNU/Linux. L'adaptation de la configuration finale du système (en particulier les applications intégrées par défaut) n'est pas simple et passe par l'édition de fichiers de configuration et l'utilisation de variables d'environnement qui ne sont pas toujours documentées. Ce système devrait être remplacé pour les futures versions d'Android. Un exemple de construction d'une image pour carte de développement *BeagleBone Black* (BBB) [19] sera explicité dans la suite du livre.

Limitations, inconvénients et pièges à connaître

L'utilisation d'Android pour une solution industrielle n'est bien évidemment pas sans risques ni difficultés à la fois techniques et commerciales. Le but de ce paragraphe est de répertorier les quelques écueils que l'utilisateur pourrait rencontrer lors du choix de l'utilisation d'Android.

Google Mobile / Play Services et certification

Ce point est probablement le plus crucial à prendre en compte si l'on envisage d'utiliser Android pour un nouvel appareil. Le modèle économique de Google est – entre autres – basé sur la diffusion d'applications mobiles (GMS pour Google Mobile Services). Ces applications (Gmail, Google Maps, etc.) sont diffusées sous licence propriétaire dans la quasi-totalité des cas (tout comme pour iOS d'Apple) et le support d'accès à Google Play (permettant de télécharger les applications) n'est pas fourni avec AOSP car il n'est pas open source, loin s'en faut. L'utilisation sur une plate-forme spéciale nécessite un agrément avec Google ainsi que la « certification » de la plate-forme. Un article du journal anglais The Guardian [48] évoque les « coûts cachés de la création d'un appareil sous Android ». Ils évoquent des coûts de $40000 pour 30000 appareils, $75000 pour 100000. L'article cite plusieurs exemples d'entreprises n'ayant pas respecté la licence de diffusion de GMS et contraintes à l'arrêt de la diffusion de

leur produit. Bien entendu, AOSP peut fonctionner sans ces applications mais le choix de l’utilisation d’Android sur un produit peut être motivé – par exemple – par l’utilisation de Google Maps. Dans le cas de produits industriels ciblés et à faible diffusion le cas est un peu différent, citons l’article du Gardian :

"Smaller OEMs [device manufacturers] don't register on Google's radar, and Google tends to turn a blind eye. Retailers get pressured by legal OEMs to make sure illegal installs of GMS are weeded out. It's almost like crowdsourcing."

La généralisation de Google Play Services est également un point à prendre en compte. Un article de FrAndroid [49] évoque cette fonctionnalité comme un « cheval de Troie » permettant à Google d’avoir le contrôle des milliards d’appareils fonctionnant sous Android. Cette fonctionnalité (non incluse dans AOSP) permet à Google de mettre à jour des fonctionnalités sans passer par une mise à jour majeure du système (complexe et angoissante pour un utilisateur moyen). Ainsi, la quasi-totalité des appareils utilisent la dernière version de Google Play Services même si ils n’ont pas la toute dernière version d’Android. Tout comme GMS, le fonctionnement sous AOSP d’un appareil dédié ne nécessite pas Google Play Services mais on peut raisonnablement se demander si cela peut perdurer.

A ce jour il est cependant possible (et fortement conseillé) de s’assurer de la compatibilité de l’appareil (et du portage AOSP) en utilisant la suite CTS (Compatibility Test Suite) qui est gratuite [50]. Concernant la certification complète, elle passe par plusieurs étapes :

- Respect du CDD (Compatibility Definition Document) [51] par le produit
- Validation de la totalité des tests CTS par le constructeur
- Envoi de plusieurs produits chez Google pour des tests complémentaires
- Certification par Google.

Notons que la certification n’est pas obligatoire sauf si l’on désire utiliser l’écosystème Android (GMS). Le passage par les étapes pré-citées ne garantit pas l’obtention effective de la certification malgré le paiement de sommes rondelettes. Google est une entreprise commerciale (et quelque part hégémonique) qui fait passer ses intérêts avant tout. Ce point explique certainement pourquoi des industries comme l’automobile ou l’aéronautique ont quelques réticences à généraliser l’utilisation d’Android ce qui explique la présence d’alliances comme GENIVI ou AGL (Automotive Grade Linux) fonctionnant sous Linux.

Empreinte mémoire

L'espace utilisé par Android sur la mémoire flash n'a aucune commune mesure avec ce que l'on connaît pour la plupart des systèmes embarqués. En effet, si l'on observe la taille de l'image principale de la version Android 7.1 pour un téléphone Google Pixel (version binaire fournie par Google), celle-ci flirte allègrement avec les 1,5 Go [21]. Il est bien entendu possible de la réduire si l'on optimise la distribution AOSP (en retirant des applications inutiles) mais il ne sera jamais possible de descendre en dessous de plusieurs centaines de Mo pour des versions plus anciennes. Dans le cas d'Android Wear (destiné à des cibles beaucoup plus

légères comme des montres ou de lunettes) l’empreinte mémoire reste autour de 300 Mo [22]. L’empreinte est à peine plus faible pour les premières versions d’Android Things disponibles.

A titre de comparaison on peut créer grâce à Buildroot ou Yocto des distributions GNU/Linux occupant au minimum 20 Mo d'empreinte mémoire !

Qualité du BSP (AOSP)

Nous avons déjà évoqué le problème des licences et des composants propriétaires. L’ambiguïté d'Android vis à vis du modèle du logiciel libre est un facteur de confusion qui peut facilement conduire à des relations difficiles avec un fournisseur. Si l'on choisit un matériel déjà disponible, par exemple un tablette plus ou moins adaptable aux besoins d'un projet, la fourniture de l'environnement AOSP (permettant d'adapter l'image Android) n'est pas forcément évidente car elle n'est pas obligatoire au regard de la licence utilisée (Apache et licences propriétaires). Dans le cas de GNU/Linux, une majorité de BSP sont actuellement disponibles sous forme de « couches » Yocto, le projet étant farouchement libre, collaboratif et très bien documenté. Même si l'arbre AOSP est fourni, la qualité de ce dernier peut être très variable et plus ou moins éloignée de celui de Google, qui reste la référence.

La procédure de production de l'image peut être différente de celle définie par Google et l'utilisation d'outils spécifiques (et là aussi propriétaires) peut apparaître au gré du choix du constructeur. Même dans le cas d'une version AOSP de bonne qualité, la documentation officielle relative au fonctionnement interne d'Android est très pauvre car Google communique très peu sur le sujet alors qu'il existe pléthore de documentation (et d'ouvrages) pour le développement d'applications. L'ouvrage « Embedded Android » (par Karim Yaghmour) [17] reste encore et toujours l'ouvrage de référence même si il date un peu. De nombreuses présentations - du même Karim Yaghmour - et documentations officieuses sont cependant disponibles sur Internet (et pour cela Google est votre ami !).

Adaptation de code « legacy »

De nombreux industriels sont tentés par les capacités graphiques d'Android et la facilité avec laquelle on peut écrire des applications élégantes et ergonomiques. Cependant certains secteurs d'activité (aéronautique, militaire, automobile, etc.) utilisent des cycles de vie très longs (plusieurs années voire dizaines d'années) qui n'ont rien à voir avec ceux de la téléphonie mobile. Un banc de test d'équipement aéronautique peut utiliser du code source écrit il y a plus de 15 ans et souvent en langage C/C++ pour une cible UNIX (POSIX) de l'époque. Android utilise en majorité le langage Java et le C/C++ est réservé aux couches « systèmes ». La compatibilité de l'environnement de développement d'Android avec la norme POSIX est partielle et la tâche d'intégration n'est donc pas simple même si des techniques existent et seront introduites dans la suite de ce livre.

Interfaces matérielles

Android est dédié à la téléphonie (voire au multimédia) alors que l'industrie utilise de nombreuses interfaces parfois datées (les standards RS-232 ou RS-485) mais toujours utiles. Un téléphone ou une tablette Android dispose de connexions sans fil (Wi-Fi, Bluetooth, NFC), de différents capteurs (GPS, accéléromètre, caméra, etc.) et d'une connectique USB. L'environnement de développement permet d'accéder à ces capteurs [18] mais l'ajout d'un nouveau type ou pire d'un bus de communication non prévu (SPI, I²C) n'est pas une tâche aisée même si cela reste réalisable de part le support de ces bus dans le noyau Linux. Plus généralement, l'ajout de pilotes étant toujours envisageable dans le noyau Linux, Android peut également être adapté pour les utiliser. Le tout nouveau « Android Things » dédié à l'Internet des objets prend en compte ces bus de manière native mais n'est pas encore officiellement disponible.

Version du noyau Linux

Android utilise depuis toujours le noyau Linux et on ne sait pas si c'est par choix ou par nécessité (probablement un mélange des deux). Le menu *Paramètres* puis *A propos de la tablette* permet d'afficher la version du noyau Linux utilisée. Les relations entre Google et la communauté du noyau Linux furent très tumultueuses de part – entre autres – la méfiance de Google vis à vis de la licence GPL . Comme nous l'avons expliqué, la majorité des fonctionnalités Android se trouvent dans l'espace utilisateur et ce afin d'éviter l'usage de la licence GPL. Les quelques pilotes ajoutés par Google représentent de ce fait un très faible volume de code (quelques milliers de lignes) dans le répertoire drivers/staging/android. Le terme *staging* désigne des pilotes qui ne sont pas encore officiellement intégrés au noyau. En 2009, les pilotes furent d'ailleurs retirés du noyau officiel (ou *mainline*).

Staging: android: delete android drivers

These drivers are no longer being developed and the original authors seem to have abandonded them and hence, do not want them in the mainline kernel tree.

So sad :(

Cc: Brian Swetland <swetland@google.com>

Signed-off-by: Greg Kroah-Hartman <gregkh@suse.de>

Les pilotes furent cependant réintégrés pour la version 3.3 du noyau officiel et même si il existe encore quelques tensions, Andrew Morton (contributeur majeur au noyau Linux) est aujourd'hui salarié de Google. Alors qu'il y a quelques années la version Google du noyau Linux était un peu éloignée, on peut désormais considérer qu'elle est assez proche de la

version officielle si l’on omet les modifications pour s’adapter aux cibles spécifiques (qui ne sont pas forcément *mainline*, mais ce point n'a rien à voir avec Android). Concrètement il est aujourd'hui possible d’utiliser le même binaire de noyau Linux pour une distribution GNU/Linux ou pour Android. Nous verrons le cas de la carte BBB dans un prochain paragraphe.

A l’heure actuelle, le noyau utilisé par Android 7.1 est au mieux la version 4.4 pour les cibles officielles d’AOSP (Google) mais cela varie en fonction de la cible [20]. Pour les autres cibles, le noyau est fourni avec le BSP.

Compiler et adapter Android (AOSP)

L'adaptation – et même la compilation – d'AOSP nécessite quelques compétences en système (GNU/Linux ou UNIX) et un environnement adapté. Il faut noter qu'un système GNU/Linux est nécessaire car Google ne fournit pas de procédure pour compiler AOSP sous Windows. La partie pré-requis du système est cependant bien documentée par Google [25].

Le choix de la version de Java à installer peut se révéler complexe car une version donnée d'AOSP n'est pas compatible avec toutes les versions de Java (sans compter l'alternative OpenJDK). Il semblerait que les choses aient évolué dans le bon sens, la branche courante d'AOSP étant désormais compatible avec OpenJDK.

Dans cette partie plus technique, nous allons donc évoquer la compilation et le test d'AOSP dans le cas le plus simple de l'émulateur (ARM) intégré. Nous traiterons ensuite le cas d'une carte *open hardware* très utilisée, en l'occurrence la fameuse BeagleBone Black (BBB). Nous réaliserons une adaptation simple d'AOSP en permettant au système de fonctionner sur un écran tactile et non sur la sortie HDMI. Enfin nous verrons comment permettre à Android d'accéder à des interfaces plus industrielles comme des ports d'entrées/sorties (GPIO) ou des capteurs (SPI, I²C, etc.).

Cas de l'émulateur

L'émulateur est la cible la plus simple et rapide à construire et nous allons passer par cette première étape d'initiation. Même si l'émulateur ne permet pas d'accéder à des interfaces matérielles, il permet de tester bon nombre d'applications simples fournies par défaut dans AOSP.

De part le volume des sources d'AOSP, il n'est pas possible d'utiliser un unique arbre Git [26] et les sources sont constituées d'un ensemble d'arbres Git que l'on manipule avec la commande repo [27] fournie par Google (un script en Python basé sur la commande git). Le volume des sources est sans comparaison avec celui d'autres projets comme le noyau Linux. De plus, il est en forte augmentation car la version 4.3 occupait environ 17 Go alors que l'on frise les 50 Go avec la version 6.0 !

La liste des arbres est définie dans un fichier manifest. Suite à cela, on peut synchroniser le répertoire local avec le dépôt des sources. Pour une cible standard d'AOSP on télécharge les sources en utilisant les commandes suivantes.

```
$ mkdir work && cd work
$ repo init -u https://android.googlesource.com/platform/manifest -b <branche>
$ repo sync
```

A ce jour le système de construction d'AOSP est basé sur des fichiers au format GNU Make et des scripts shell (au format BASH). Ce point devrait évoluer pour les futures versions mais nous traiterons uniquement le cas présent dans ce document. Il faut également noter que les versions AOSP disponibles pour les cartes industrielles ne sont pas forcément les plus récentes vu qu'un temps d'adaptation est nécessaire. C'est d'ailleurs également le cas pour des téléphones ou tablettes autres que ceux fournis par Google.

La compilation d'AOSP nécessite tout d'abord de sélectionner la cible et l'on utilise ensuite la commande make pour produire la distribution binaire. L'adaptation d'AOSP permettra d'ajouter de nouvelles cibles utilisables avec la commande lunch.

```
$ source build/envsetup.sh
$ lunch <nom-de-cible>
$ make -j <nombre-de-coeurs>
```

Si l'on utilise la commande lunch sans paramètre, la liste des cibles disponibles est affichée sur le terminal et il suffit de sélectionner le numéro de la cible désirée.

```
$ lunch

You're building on Linux

Lunch menu... pick a combo:
    1. aosp_arm-eng
    2. aosp_x86-eng
    3. aosp_mips-eng
    4. vbox_x86-eng
    5. aosp_deb-userdebug
    6. aosp_flo-userdebug
    ...
```

Dans le cas de l'utilisation de l'émulateur, le nom de la cible est *arm_aosp-eng* (soit la première de la liste). Le suffixe *eng* correspond à engineering, soit l'activation de toutes les options et outils de mise au point. Le suffixe *user* correspond à une version de production et *userdebug* à une configuration intermédiaire permettant d'accéder au système en tant qu'administrateur (root) et activer l'option de mise au point à distance. Le résultat de la compilation (plus d'une heure sur une machine décente) se trouve dans le répertoire out/target/product/generic qui occupe également un espace important. Il suffit ensuite d'utiliser la commande suivante pour tester Android.

```
$ emulator &
```

L'utilisation de ccache [40] permet d'améliorer le temps de compilation dans le cas fréquent d'effacement puis re-compilation par make clean puis make.

Cas de la carte BBB

Si l'on considère une autre cible que les produits Google (Nexus, Pixel) ainsi que la carte de référence HiKey, il est nécessaire d'apporter des modifications à AOSP pour utiliser Android sur cette cible. Outre le bootloader et le noyau Linux qui doivent être disponibles pour la cible

(avec les pilotes nécessaires), la partie système doit être enrichie d'un certain nombre de fichiers de configuration mais également des bibliothèques (.so) afin d'être compatible avec la HAL (voir figure 2).

Il existe parfois plusieurs projets fournissant la version AOSP modifiée (fournisseur de matériel, communauté, etc.) comme c'est fréquemment le cas dans le monde de l'open source. Le cas est identique pour le noyau Linux et si nous prenons l'exemple de la carte BBB, il en existe *trois* (noyau officiel ou *mainline*, noyau fourni par Texas Instruments et celui de la communauté Beagleboard !). Concernant la BBB il existe deux portages d'AOSP, soit [23] et [24]. Nous utiliserons le deuxième – développée par la société 2net – car il est très bien documentée. Le projet fournit une adaptation pour plusieurs versions d'AOSP (4.3, 4.4.4, 5.1.1 et 6.0.1) basées sur des versions officielles de l'arbre AOSP de Google (android-4.3_r2.1, android-4.4.4_r2, android-5.1.1, android-6.0) en utilisant la procédure de construction préconisée par Google et utilisée pour le cas de l'émulateur (ce qui n'est pas toujours le cas de tous les BSP Android !). Dans notre cas nous choisirons une version 4.3 (Jelly Bean) [28] car c'est la plus rapide à construire et exécuter sur la cible de part son encombrement relativement raisonnable (la BBB disposant uniquement de 512 Mo de RAM). L'utilisation d'une version plus récente ne change pas le principe de construction et d'installation, que ce soit sur la BBB ou bien une autre cible.

L'adaptation d'AOSP pour une cible donnée nécessite donc :

- Les sources AOSP officielles fournies par Google (voir paragraphe précédent)
- Les modifications liées au portage, rassemblées dans le répertoire device/<constructeur>/<cible>
- Une version de bootloader (souvent U-Boot) adaptée à la cible matérielle et compatible avec le protocole « Fastboot » de Google (utilisé pour programmer la mémoire flash de la cible)
- Une version du noyau Linux adaptée à la cible matérielle. Il faut noter que désormais le même noyau Linux peut être utilisé pour GNU/Linux ou pour Android ce qui est une avancée notoire !

En premier lieu il convient d'obtenir les sources (non modifiées) à partir du dépôt Google en précisant la branche utilisée, soit la commande :

```
$ mkdir work_43 && cd work_43
```

```
$ repo init -u https://android.googlesource.com/platform/manifest -b android-4.3_r2.1
$ repo sync
```

Dans la source AOSP, la partie dépendante du matériel est localisée dans le répertoire device.

```
$ ls -l device
total 36
drwxr-xr-x  7 pierre pierre 4096 août   8  2013 asus
```

```
drwxr-xr-x  4 pierre pierre 4096 août   8  2013 common
drwxr-xr-x 10 pierre pierre 4096 août   8  2013 generic
drwxr-xr-x  3 pierre pierre 4096 août   8  2013 google
drwxr-xr-x  4 pierre pierre 4096 août   8  2013 lge
drwxr-xr-x 10 pierre pierre 4096 août   8  2013 sample
drwxr-xr-x  7 pierre pierre 4096 août   8  2013 samsung
drwxr-xr-x  3 pierre pierre 4096 août   8  2013 samsung_slsi
drwxr-xr-x  4 pierre pierre 4096 janv. 10 18:25 ti
```

Par défaut, seule la carte Pandaboard est (ou était) prise en charge dans le répertoire ti.

```
$ ls -l ti/
total 4
drwxr-xr-x 9 pierre pierre 4096 août   8  2013 panda
```

On doit donc ajouter le répertoire beagleboneblack contenant les fichiers spécifiques à la carte BBB. Nous utilisons la branche distante *jb4.3-fastboot* car l'organisation des partitions est adaptée à Fastboot.

```
$ cd work_43/device/ti
$ git clone https://github.com/csimmonds/bbb-android-device-files.git beagleboneblack
$ cd beagleboneblack
$ git fetch
$ git checkout jb4.3-fastboot
```

Nous avons vu au paragraphe précédent que la sélection de la cible s'effectuait par la séquence suivante à partir du répertoire des sources.

```
$ source build/envsetup.sh
$ lunch beagleboneblack-eng
```

On constate l'ajout de la cible *beagleboneblack* par rapport à la version standard. Il suffit alors de compiler AOSP par :

```
$ make -j <nombre-de-coeurs>
```

Le répertoire beagleboneblack ajouté contient un certain nombre de fichiers dont le script vendorsetup.sh qui ajoute l'entrée *beagleboneblack-eng* .

```
$ cat device/ti/beagleboneblack/vendorsetup.sh
add_lunch_combo beagleboneblack-eng
```

Les autres fichiers constituent la configuration spécifique à la BBB.

```
$ ls
Android.mk         fstab.am335xevm       README.md
AndroidProducts.mk  gpio-keys.kl         sgx
audio_policy.conf   init.am335xevm.rc     ti-tsc.idc
beagleboneblack.mk  init.am335xevm.usb.rc  uEnv.txt
BoardConfig.mk      liblights             ueventd.am335xevm.rc
CleanSpec.mk        media_codecs.xml      vendorsetup.sh
device.mk           media_profiles.xml    vold.fstab
device-sgx.mk       mixer_paths.xml
egl.cfg             overlay
```

Ce point est très peu documenté par Google, mais on peut se référer à l'ouvrage cité en [17] ainsi qu'à la conférence sur AOSP en [28]. Outre le manque de documentation officielle, il existe hélas très peu d'ouvrages et de références sur le domaine si l'on compare avec le

développement d'application. Les fichiers .mk sont au format GNU Make et utilisent des macros du type :

```
$(call inherit-product, device/ti/beagleboneblack/device.mk)
```

Ces macros permettent d' « hériter » des paramètres d'un autre fichier .mk. En premier lieu, le fichier AndroidProducts.mk définit la (ou les) cible(s) :

```
PRODUCT_MAKEFILES := $(LOCAL_DIR)/beagleboneblack.mk
```

Le fichier beagleboneblack.mk utilise des variables propres à AOSP afin de définir un certain nombre de paramètres comme le nom de la cible, le constructeur, etc.

```
PRODUCT_NAME := beagleboneblack
PRODUCT_DEVICE := beagleboneblack
PRODUCT_BRAND := Android
PRODUCT_MODEL := BEAGLEBONEBLACK
PRODUCT_MANUFACTURER := Texas_Instruments_Inc
```

Le fichier device.mk permet de définir d'autres paramètres comme des fichiers ajoutés à l'image AOSP produite, dont le noyau Linux que nous devons compiler séparément. Ce dernier doit être copié sur le répertoire sous le nom kernel.

```
ifeq ($(TARGET_PREBUILT_KERNEL),)
LOCAL_KERNEL := device/ti/beagleboneblack/kernel
else
LOCAL_KERNEL := $(TARGET_PREBUILT_KERNEL)
endif
```

La variable PRODUCT_COPY_FILE permet tout simplement de copier des fichiers vers l'image AOSP en utilisant le séparateur « deux points ».

```
PRODUCT_COPY_FILES := \
    $(LOCAL_KERNEL):kernel \
    device/ti/beagleboneblack/init.am335xevm.rc:root/init.am335xevm.rc \
    device/ti/beagleboneblack/init.am335xevm.usb.rc:root/init.am335xevm.usb.rc \
...
```

De même, la variable PRODUCT_PROPERTY_OVERRIDE permet de définir (ou modifier) une propriété Android (qui correspond à peu près à une « variable d'environnement » sous UNIX ou GNU/Linux).

```
PRODUCT_PROPERTY_OVERRIDES += \
    persist.sys.strictmode.visual=0 \
    persist.sys.strictmode.disable=1
```

La variable DEVICE_PACKAGE_OVERLAYS permet de remplacer une arborescence complète par une nouvelle fournie dans la définition de la carte, soit dans notre cas le répertoire overlay.

```
DEVICE_PACKAGE_OVERLAYS := \
  device/ti/beagleboneblack/overlay
```

Cette adaptation d'Android utilise la sortie HDMI de la BBB pour l'affichage (et donc un écran externe). Dans la suite de la présentation nous allons voir quelles modifications apporter afin d'utiliser un écran tactile compatible avec la BBB visible sur la figure ci-dessous [29]. Cet écran est bon marché et peut être utilisé facilement avec Android.

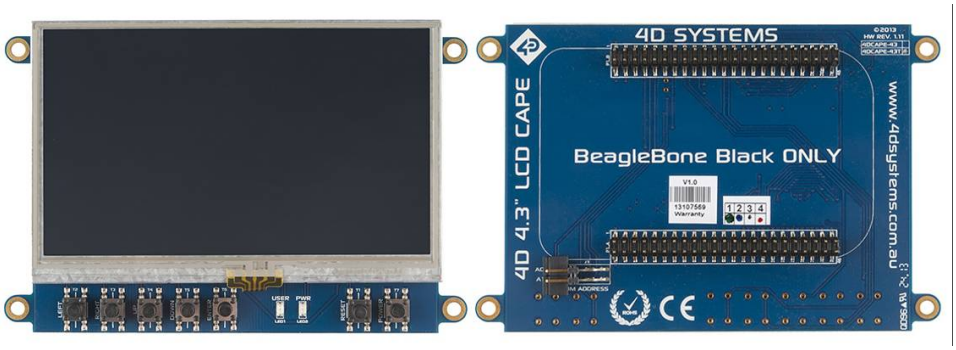


Figure 3. Écran tactile LCD Cape

En l'absence d'accélération graphique, le système utilise le *framebuffer* du noyau Linux comme dans le cas de l'émulateur Android (QEMU). Le fichier BoardConfig.mk décrit la configuration matérielle de la cible et en particulier les options utilisées lors du démarrage du noyau. Dans notre cas, nous définissons le nom de la cible à la valeur *am335xevm* et surtout l'option qemu=1 qui force l'utilisation du framebuffer.

```
# CAPE touchscreen config (qemu=1) => no hw acceleration
BOARD_KERNEL_CMDLINE := console=ttyO0,115200n8 androidboot.console=ttyO0
androidboot.hardware=am335xevm rootwait ro init=/init ip=off qemu=1 vt.global_cursor_default=0
```

Le nom de la machine est important dans le cas d'Android puisque cela permet au processus *init* de démarrer le script d'initialisation init.am335xevm.rc.

Le deuxième point concerne le noyau Linux. L'écran tactile ne fonctionne pas avec le noyau 3.2 utilisé par le portage initial et nous devons donc utiliser une version plus récente, soit la 3.8.13. Le noyau BBB utilise la technique du *device tree* dont la description dépasse largement les limites de ce livre. En résumé, cette technique permet de définir la configuration matérielle de la cible (la liste des contrôleurs matériels disponibles) dans un fichier séparé (.dtb) chargé par le bootloader et traité par le noyau. Une présentation du *device tree* est disponible en [30].

Le bootloader adapté à Android n'est pas compatible avec le *device tree* et il convient d'ajouter le fichier am335x-boneblack.dtb à la suite du noyau zImage afin de créer le fichier kernel utilisé par Android en ayant pris soin d'activer l'option noyau nécessaire.

```
$ cat arch/arm/boot/zImage  arch/arm/boot/dts/am335x-boneblack.dtb >
<path>/work_43/device/ti/beaglebonblack/kernel
```

Le dernier point concerne les touches Android (Back, Home, etc.), car l'écran tactile utilisé est trop petit pour pouvoir utiliser les boutons virtuels d'une tablette standard. Nous allons donc associer les boutons physiques de l'écran (LEFT, RIGHT, etc.) à des « actions » Android. L'action BACK est la plus importante, car elle permet de quitter l'application courante. Les codes des touches ainsi que le nom du périphérique (écran tactile) sont obtenus grâce à la commande getevent, ce qui permet ensuite de définir le nom du fichier de « mapping », soit gpio_keys_13.kl.

```
$ cat device/ti/beagleboneblack/gpio_keys_13.kl
# Beaglebone LCD Cape  GPIO KEYPAD keylayout
key 105   DPAD_LEFT      VIRTUAL
```

```
key 106   DPAD_RIGHT     VIRTUAL
key 103   DPAD_UP        VIRTUAL
key 108   DPAD_DOWN      VIRTUAL
key 28    BACK           VIRTUAL
```

L'utilisation de getevent est décrite ci-dessous :

```
# getevent
could not get driver version for /dev/input/mice, Not a typewriter
could not get driver version for /dev/input/mouse0, Not a typewriter
add device 1: /dev/input/event1
  name:     "ti-tsc"
could not get driver version for /dev/input/mouse1, Not a typewriter
add device 2: /dev/input/event3
  name:     "PS/2+USB Mouse"
add device 3: /dev/input/event2
  name:     "gpio_keys.13"
add device 4: /dev/input/event0
  name:     "tps65217_pwr_but"
/dev/input/event2: 0001 0069 00000001
/dev/input/event2: 0000 0000 00000000
/dev/input/event2: 0001 0069 00000000
/dev/input/event2: 0000 0000 00000000
```

Une fois la compilation terminée, on peut passer à l'installation de l'image AOSP sur la cible. Android utilise le protocole *Fastboot* pour l'installation sur la cible des images produites. Là encore, la société 2net fournit en [31] une documentation détaillée pour l'installation d'une version modifiée de U-Boot supportant Fastboot puis l'installation des images .img produites lors de la compilation d'AOSP dans le répertoire out/target/product/beagleboneblack. On remarque l'empreinte mémoire utilisée par les images, largement supérieure à ce que l'on peut faire avec GNU/Linux (embarqué) !

```
$ ls -lh out/target/product/beagleboneblack/*.img
-rw-r--r-- 1 pierre pierre 5.1M Mar  3 11:54 out/target/product/beagleboneblack/boot.img
-rw-r--r-- 1 pierre pierre 256M Mar  2 17:54 out/target/product/beagleboneblack/cache.img
-rw-rw-r-- 1 pierre pierre 241K Mar  3 11:54 out/target/product/beagleboneblack/ramdisk.img
-rw-r--r-- 1 pierre pierre 256M Mar  3 11:54 out/target/product/beagleboneblack/system.img
-rw-r--r-- 1 pierre pierre 256M Mar  2 18:00 out/target/product/beagleboneblack/userdata.img
```

Après avoir formaté la mémoire flash (de type eMMC), on installe le firmware MLO puis l'image U-Boot depuis une carte Micro-SD.

```
$ fastboot oem format
$ fastboot flash spl MLO
$ fastboot flash bootloader u-boot.img
```

Après redémarrage sur la eMMC, on peut alors installer les images Android.

```
$ fastboot flash userdata
$ fastboot flash cache
$ fastboot flashall
```

Le démarrage de la carte à partir d'U-Boot donne les traces suivantes sur la console.

```
U-Boot 2013.01.01-gc0dd2af-dirty (Aug 30 2014 - 20:36:55)

I2C:   ready
DRAM:  512 MiB
WARNING: Caches not enabled
NAND:  No NAND device found!!!
```

```
0 MiB
MMC:   OMAP SD/MMC: 0, OMAP SD/MMC: 1
*** Warning - readenv() failed, using default environment

musb-hdrc: ConfigData=0xde (UTMI-8, dyn FIFOs, bulk combine, bulk split, HB-ISO Rx, HB-ISO Tx, SoftConn)
musb-hdrc: MHDRC RTL version 2.0
musb-hdrc: setup fifo_mode 4
musb-hdrc: 28/31 max ep, 16384/16384 memory
USB Peripheral mode controller at 47401000 using PIO, IRQ 0
musb-hdrc: ConfigData=0xde (UTMI-8, dyn FIFOs, bulk combine, bulk split, HB-ISO Rx, HB-ISO Tx, SoftConn)
musb-hdrc: MHDRC RTL version 2.0
musb-hdrc: setup fifo_mode 4
musb-hdrc: 28/31 max ep, 16384/16384 memory
USB Host mode controller at 47401800 using PIO, IRQ 0
Net:   <ethaddr> not set. Validating first E-fuse MAC
cpsw
Hit any key to stop autoboot:  0
mmc_send_cmd : timeout: No status update
mmc1(part 0) is current device
mmc_send_cmd : timeout: No status update
Loading efi partition table:
    256    128K spl
    512    512K bootloader
   1536    128K misc
   2048      8M recovery
  18432      8M boot
  34816    256M system
 559104    256M cache
1083392    256M userdata
1607680   1047M media
Loaded eMMC partition table
SELECT MMC DEV: 1
mmc1(part 0) is current device
...
Cleanup before kernel ...

Starting kernel, theKernel ...

Uncompressing Linux... done, booting the kernel.
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Linux version 3.8.13 (pierre@XPS-pf) (gcc version 4.7.3 (Sourcery CodeBench Lite 2013.05-24) ) #8
SMP Thu Jan 15 12:47:24 CET 2015
[    0.000000] CPU: ARMv7 Processor [413fc082] revision 2 (ARMv7), cr=50c5387d
[    0.000000] CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
[    0.000000] Machine: Generic AM33XX (Flattened Device Tree), model: TI AM335x BeagleBone
...
```

Le démarrage est assez long (surtout pour la partie graphique) mais on obtient finalement l'image suivante sur l'écran .

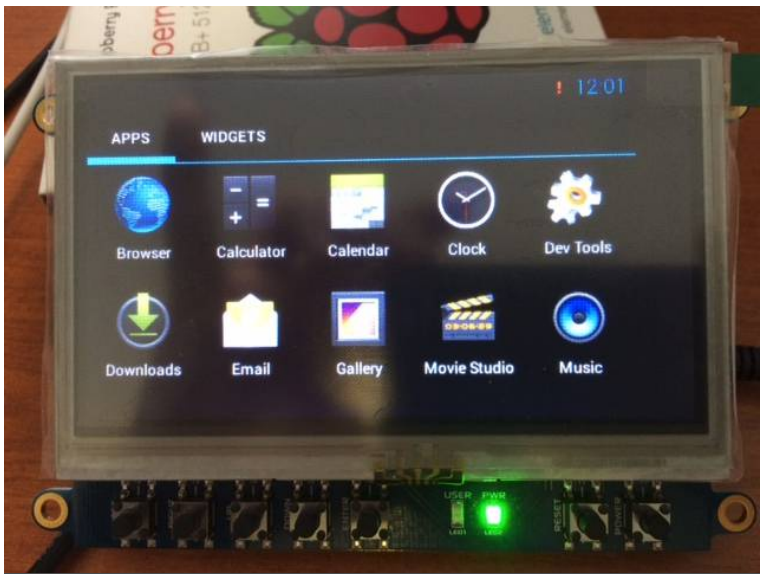


Figure 4. Android sur BBB et écran tactile

Utilisation d'interfaces « industrielles »

De nombreux projets industriels utilisent des interfaces matérielles spécifiques, assez éloignées du monde de la téléphonie sur lequel Android est largement majoritaire. Nous pouvons citer l'interface réseau Ethernet (et dérivés), les bus RS-232, RS-485, I²C et SPI, les entrées/sorties « tout ou rien » ou GPIO (pour *General Purpose Input Output*). Le support Ethernet est souvent présent dans les versions AOSP fournies par les constructeurs (c'est le cas pour la BBB). Certaines interfaces sont désormais prises en compte par « Android Things », système d'exploitation proche d'Android et adapté aux objets connectés. A titre d'exemple, le SDK de Things fournit une interface pour le bus SPI, le bus I²C, les GPIO et autres PWM [32]. Dans le cas d'AOSP, les choses ne sont pas si simples si l'on sort des interfaces habituelles de la téléphonie. L'interface USB – très répandue – est également accessible depuis le SDK Android [33]. Quelques exemples sont disponibles dont le pilotage du célèbre lance-missile USB [34]. L'utilisation d'interfaces non prévues dans le SDK Android peut s'effectuer de plusieurs manières :

- En utilisant une application en mode « texte », sachant qu'Android utilise un noyau Linux et qu'il est possible de compiler des exécutables.

- En utilisant JNI pour interfacer une application Java avec un périphérique matériel via une bibliothèque partagée (.so). Cette bibliothèque peut elle-même faire appel à un pilote Linux en ouvrant un fichier dans /dev .

- En créant un « service » et un « manager » Android associés à la nouvelle interface matérielle ainsi qu'une extension de l'API de programmation. Cette méthode est la plus aboutie car elle est conforme au fonctionnement du système mais l'ajout d'un service est relativement complexe et nécessite des modifications à plusieurs niveaux (Framework Java, HAL, SDK). Les services standards (Wi-Fi, téléphonie, etc.) utilisent ce principe ainsi que le schéma de la figure 2. Nous reviendrons sur ce sujet et un exemple simple est disponible en référence [35].

- En utilisant des fichiers « virtuels » (exemple /sys/class/leds ou /sys/class/gpio) l'accès étant possible depuis une application Java. Bien entendu cette solution concerne des périphériques pilotables par de simples lectures/écritures.

Une présentation évoquant ces solutions est disponible en référence [38].

Compilation d'application en mode « texte »

De nombreuses applications sous GNU/Linux fonctionnent en mode texte. Il est ensuite possible d'interfacer – si nécessaire - le programme de pilotage avec une application graphique. Comme nous l'avons vu au début du livre, la compilation de code natif C/C++ n'est pas forcément évidente car ce n'est pas l'approche proposée par Android qui par défaut favorise le développement d'application en Java en utilisant le framework fourni (et le SDK). Un NDK (pour *Native Development Kit*) est cependant disponible auprès de Google [36] . Ce dernier est constitué d'un ensemble de chaînes de compilation croisées pour les architectures compatibles avec Android (ARM, x86, MIPS en 32 ou 64 bits). Quelques exemples sont également disponibles en [37].

Il est important de remarquer que le NDK n'est pas réellement conçu pour compiler les applications génériques C/C++ que l'on peut rencontrer sur GNU/Linux (ou des systèmes d'exploitation POSIX en général) mais plutôt des composants intégrés à Android et liés à des applications Java (bibliothèque JNI). Android n'étant pas un système d'exploitation POSIX, il utilise une libC simplifiée (Bionic) qui n'a rien à voir avec la GNU-libC (ou Glibc) utilisée par défaut sous GNU/Linux. Dans une des documentations de Bionic, on peut d'ailleurs lire :

> The core idea behind Bionic's design is: KEEP IT REALLY SIMPLE.
>
> This implies that the C library should only provide lightweight wrappers
>
> around kernel facilities and not try to be too smart to deal with edge cases.

Nous allons cependant voir comment compiler un programme simple en utilisant le script ndk-build fourni avec le NDK. Android n'utilise pas le principe habituel du fichier Makefile bien connu sous UNIX ou GNU/Linux mais un fichier Android.mk qui est placé dans le répertoire des sources et contient les lignes suivantes :

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_SRC_FILES = helloworld.c

LOCAL_MODULE = HelloWorld
LOCAL_MODULE_TAGS = optional
```

```
include $(BUILD_EXECUTABLE)
```

Le répertoire contenant les sources ainsi que le fichier Android.mk doit être nommé jni par défaut.

```
$ tree .
.
└── jni
    ├── Android.mk
    └── helloworld.c
```

La compilation utilise le script ndk-build et produit les différents fichiers binaires à installer sur la cible correspondante.

```
$ ~/android-ndk-r14b/ndk-build
[arm64-v8a] Compile       : HelloWorld <= helloworld.c
[arm64-v8a] Executable    : HelloWorld
[arm64-v8a] Install       : HelloWorld => libs/arm64-v8a/HelloWorld
[x86_64] Compile       : HelloWorld <= helloworld.c
[x86_64] Executable    : HelloWorld
[x86_64] Install       : HelloWorld => libs/x86_64/HelloWorld
[mips64] Compile       : HelloWorld <= helloworld.c
[mips64] Executable    : HelloWorld
[mips64] Install       : HelloWorld => libs/mips64/HelloWorld
[armeabi-v7a] Compile thumb  : HelloWorld <= helloworld.c
[armeabi-v7a] Executable     : HelloWorld
[armeabi-v7a] Install       : HelloWorld => libs/armeabi-v7a/HelloWorld
[armeabi] Compile thumb  : HelloWorld <= helloworld.c
[armeabi] Executable     : HelloWorld
[armeabi] Install       : HelloWorld => libs/armeabi/HelloWorld
[x86] Compile       : HelloWorld <= helloworld.c
[x86] Executable    : HelloWorld
[x86] Install       : HelloWorld => libs/x86/HelloWorld
[mips] Compile       : HelloWorld <= helloworld.c
[mips] Executable    : HelloWorld
[mips] Install       : HelloWorld => libs/mips/HelloWorld
```

Pour un test rapide d’installation et d’exécution on peut utiliser ADB.

```
$ file libs/armeabi-v7a/HelloWorld
libs/armeabi-v7a/HelloWorld: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked,
interpreter /system/bin/linker, BuildID[sha1]=6ad3885a96e5a8f311a35436d89aa4d2eae57dd7, stripped

$ adb push libs/armeabi-v7a/HelloWorld /data
$ adb shell /data/HelloWorld
Hello World !
```

Dans le cas d’une application plus complexe nécessitant une meilleure compatibilité POSIX il est également possible d’utiliser un compilateur croisé non fourni par Android mais disponible dans l’environnement GNU/Linux (embarqué) et basé sur la Glibc. De ce fait on pourra faire cohabiter des applications Android avec des exécutables compatibles avec Linux. Cette configuration est possible car le noyau utilisé par Android est compatible avec celui de GNU/Linux (au niveau des appels systèmes). Le problème est la communication entre les deux mondes puisqu’ Android utilise *Binder* comme système d’IPC, contrairement à GNU/Linux qui utilise les IPC SystemV. On peut cependant imaginer l’architecture suivante en supposant que l’on utilise un support de communication commun entre les deux mondes (par exemple des *sockets*).

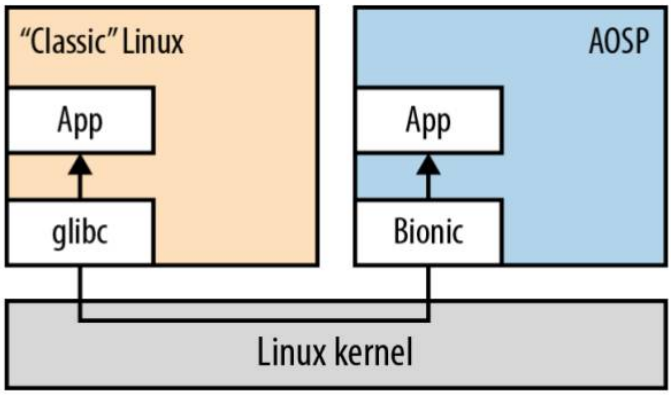


Figure 5. Cohabitation Bionic / Glibc

Utilisation de JNI

L'utilisation de JNI est plus fréquente que le cas précédent. Nous rappelons qu'il s'agit d'écrire une bibliothèque partagée (.so) permettant d'accéder à des couches C/C++ depuis l'application Java, la bibliothèque étant compilée avec le NDK. Il existe quelques exemples disponibles dont la référence [39] décrivant un accès I²C depuis Android. Dans le cas présent nous allons décrire un exemple simple basé sur l'accès à un périphérique simulé par une entrée /dev/temper0 créée par l'insertion d'un module noyau temper.ko. Nous ne décrirons pas ici la création du pilote ni sa compilation.

La lecture du fichier retourne une valeur correspondant à une température sur deux octets. Le calcul de la température réelle (en °C) sera réalisé dans l'application. Le pilote modifie automatiquement la valeur retournée lors de chaque lecture en ajoutant 500.

```
# insmod temper.ko
# cat /dev/temper0
5000
# cat /dev/temper0
5500
```

L'accès au fichier /dev/temper0 sera réalisé par une fonction getTemp() définie dans une bibliothèque JNI (libtemper.so) intégrée au paquet (.apk) de l'application Android. Le paquet devra contenir les binaires utilisables sur les différentes cibles (x86, ARM, etc.) puisque le code est compilé. Le code source de la fonction – très simple – est reproduit ci-dessous :

```
#include <string.h>
#include <stdio.h>
#include <jni.h>

#define FILE_PATH "/dev/temper0"

/*  Java_<package name>_<class name>_<method name> */

jstring
Java_com_example_temper_MainActivity_getTemp( JNIEnv* env,
                                              jobject thiz )
{
  char buf[256];
  FILE *fp = fopen (FILE_PATH, "r");
  if (!fp)
```

```
    return (*env)->NewStringUTF(env, "No data file !");

  memset (buf, 0, sizeof(buf));
  fread (buf, 1, sizeof(buf), fp);
  fclose (fp);

  return (*env)->NewStringUTF(env, buf);
}
```

Le fichier Android.mk associé est proche du précédent mis à part que l'on produit une *bibliothèque partagée* et non un exécutable.

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE    := temper
LOCAL_SRC_FILES := temper.c

include $(BUILD_SHARED_LIBRARY)
```

Du coté de l'application Java, la bibliothèque est chargée dans la classe *MainActivity*

```
static {
    System.loadLibrary("temper");
}
```

La fonction update_temp() effectue la mise à jour de la température lue depuis /dev/temper0 via la bibliothèque JNI et l'affiche dans un objet graphique *TextView*.

```
public native String getTemp();
...
void update_temp()
{
  tv = (TextView)this.findViewById(R.id.textView2);

  Float f = Float.parseFloat(getTemp());  // get temperature from device

  // Convert driver data to °C
  f = (f * 125) / 32000;

  String s = String.format("%.1f °C", f);

  tv.setText(s); // update temperature
}
```

L'application (fonctionnant ici dans l'émulateur Android) indique la température. L'action sur le bouton *Update* permet de mettre à jour la valeur.

Figure 6. Application Android

Méthode complète (HAL, service et « manager »)

Nous ne décrirons pas entièrement cette méthode car elle nécessite un développement beaucoup plus lourd. Elle est cependant utilisée pour les périphériques standards d’Android (Wi-Fi, téléphonie, rétro-éclairage, etc.). Le principe général d’Android est de ne jamais permettre l’accès « direct » à une ressource matérielle mais de passer par un *manager* (service) dédié. Un exemple déjà cité est disponible en référence [35]. La commande suivante permet d’afficher les services disponibles.

```
$ service list
Found 69 services:
0	phone: [com.android.internal.telephony.ITelephony]
1	iphonesubinfo: [com.android.internal.telephony.IPhoneSubInfo]
2	simphonebook: [com.android.internal.telephony.IIccPhoneBook]
3	isms: [com.android.internal.telephony.Isms]
...
31	 statusbar: [com.android.internal.statusbar.IstatusBarService]
...
```

De plus, l'utilisation de la HAL (voir figure 2) permet de garantir le fonctionnement du service sur toutes les plate-formes. Pour notre exemple simple de mesure de température nous devrions en théorie modifier AOSP à plusieurs niveaux.

- Définir une nouvelle interface HAL pour le nouveau service et ajouter les bibliothèques – en C/C++ - permettant d'utiliser le service sur les différentes cibles (émulateur, BBB, etc.)
- Ajouter le code JNI nécessaire au chargement du module HAL (en C/C++)
- Ajouter un nouveau service (en Java)
- Ajouter un *manager* et exposer le nouveau service au framework
- Modifier le SDK afin que la nouvelle classe soit utilisable pour le développement des applications

Une fois les opérations accomplies, un nouveau service est visible dans le système.

```
$ service list | grep temper
7	temper: [android.os.ItemperService]
```

Coté application on dispose à présent d'une classe correspondant à notre nouveau matériel et on déclare une variable de type *TemperManager*.

```
import android.os.TemperManager; // My new class !!

public class HelloTemperInternalActivity extends Activity {
   ...
   TemperManager mTemperManager;
}
```

La nouvelle version de la fonction update_temp() obtient la température en interrogeant le service et donc le périphérique via la HAL.

```
void update_temp() {
 Float f = 0.0f;
 String s;
 tv = (TextView)this.findViewById(R.id.textView2);

 try {
 f = Float.parseFloat (mTemperManager.read (20)); // Get temp from service

  f = (f * 125) / 32000; // Convert to °C
  s = String.format ("%.1f °C", f);
  tv.setText(s); // update TextView
 } catch (Exception e) { … }
}
```

Le cas d'Android Things

Android Things est le dernier né de la famille Google et nous avons peu d'éléments à son sujet sachant qu'il est pour l'instant disponible en version *preview* [41] et que les sources ne sont pas encore publiées. Cet événement justifie cependant quelques mots car le marché de l'IoT pourrait être impacté par ce nouvel OS. Pour refaire un peu l'histoire, Google avait proposé il y a quelques années l'OS *Brillo* qui correspondait peu ou prou à Android auquel on aurait retiré la partie graphique (Java). Ce système n'a jamais pu s'imposer et on pria donc les ingénieurs de Google de réfléchir à une nouvelle approche moins radicale. L'interface graphique – même si elle est optionnelle - est donc de retour dans Android Things. Point fondamental, les interfaces matérielles évoquées précédemment (I^2C, SPI, GPIO, etc.) sont nativement supportées par Things. Autre point intéressant, le développement d'application passe par l'outil Android Studio. Android Things semble donc être l'OS parfait pour le développement embarqué et IoT dans l'environnement Android.

La version actuelle est pour l'instant disponible pour un nombre limité de plate-formes (4 cibles dont la Raspberry Pi 3). Comme pour Android, cette liste concerne des cibles 32 bits ou plus et il y a donc peu de chances de trouver Things sur un capteur léger qui reste la chasse gardée du *bare metal* ou des OS comme Contiki [42] ou RIOT [43] qui tout deux peuvent fonctionner sur des cibles 16 bits (voire 8 bits) et des empreintes mémoire de quelques dizaines de Ko ! A contrario l'image de test – compressée – de Things pour Raspberry Pi 3 tourne autour de 250 Mo sans aucune application fournie (à part l'affichage de la liste des ports détectés).

```
$ ls -lh ~/Téléchargements/androidthings_rpi3_devpreview_3.zip
-rw-rw-r-- 1 pierre pierre 250M May  6 09:23
/home/pierre/Téléchargements/androidthings_rpi3_devpreview_3.zip
```

Le choix de Things doit donc être motivé par d'autres facteurs comme l'écosystème Android, une certaine standardisation dans les API et outils de développement (Android Studio) mais aussi l'intégration possible dans un environnement utilisant des technologies comme *Greybus* [44] développé pour « feu » le projet ARA de Google (téléphone ouvert et modulaire). Une excellente présentation technique de Things (par Karim Yaghmour) est disponible en [45]. De même une interview de Karim et Chris Simmonds (auteur du portage Android pour BBB) est disponible en [46]. Une présentation Google I/O 2017 est également disponible en [47]. Là encore il faut noter (et peut-être craindre?) l'intégration complète des objets dans l'écosystème Google (dont le fameux Google Play Services déjà évoqué dans le document).

Le marché actuel et à venir

Depuis son lancement, Android est déployé sur plus de deux milliards de smartphones et tablettes. La stratégie de Google est claire, fournir un système d'exploitation, y adjoindre l'ensemble de ses services et s'appuyer sur une communauté de développeurs pour enrichir la plateforme avec du contenu. Fort de ce succès sur le marché du mobile, c'est lors de la conférence Google I/O en 2014, que le géant californien de l'internet a révélé sa stratégie de conquête en déclinant sa plate-forme Android sur de nouveaux marchés verticaux en s'appuyant sur son écosystème créé pour le Smartphone.

La télévision (Android TV)

Comme son nom le suggère, Android TV se base sur le système d'exploitation Android profitant par la même occasion de l'écosystème de ce dernier et de sa facilité de développement. Pour les constructeurs de TV, cela permet de profiter du marché des télévisions connectées à « moindre coût ». Cette solution offre aux clients la possibilité d'accéder à pléthore d'applications déjà développées pour Android TV alors qu'il serait trop compliqué d'attirer des développeurs sur une autre plateforme.

À l'heure actuelle, des produits sont disponibles chez des constructeurs comme Sony, Sharp et Philips, et Google a signé de nouveaux partenariats. Dans le courant de l'année, Android TV sera disponible sur des appareils signés Arcelik (Turquie), Vestel (Turquie), RCA, Hisense (Chine), TCL (Chine) et Bang & Olufsen (Danemark). Une liste qui s'étend donc, mais sans pour autant inclure les plus grands noms puisque des constructeurs comme Samsung ou LG préfèrent miser sur leurs propres systèmes d'exploitation, à savoir Tizen et WebOS 2.0. En France, nous pouvons citer Bouygues Telecom et Free qui utilisent Android dans les set-top box Miami et mini 4K.

Les « wearables » (Android Wear)

Android Wear a la particularité de reposer sur le système de reconnaissance et de commande vocale Google Now. Android Wear – désormais en version 2.0 - à débuté sa carrière sur les montres connectées de ses partenaires historiques comme Asus, LG et Huawei. Cependant, selon un nouveau rapport de Strategy Analytics [52], l'OS Tizen de Samsung a dépassé Android Wear pour la première fois avec 19% de part de marché global, contre 18% pour Google.

L'Automobile (Android Auto)

Android représente un choix potentiellement intéressant pour les systèmes d'infotainment automobile (IVI). L'objectif de Google est d'inciter les constructeurs à intégrer Android dans l'habitacle des véhicules et donc permettre aux conducteurs de déporter l'affichage du téléphone sur l'écran du véhicule afin de l'utiliser en toute sécurité. Les OEM automobiles voient Android comme moyen de fournir la meilleure expérience multimédia possible et un moyen de tirer parti de la familiarité des consommateurs avec les appareils mobiles. De plus, certaines applications mobiles sont aujourd'hui supérieures en qualité à des fonctionnalités fournies par le constructeur (citons l'application GPS Waze de Google).

La disponibilité de la plate-forme d'infotainment Android arrive à un moment où les équipementiers prennent de plus en plus de contrôle sur l'infrastructure numérique dans les voitures. Le modèle traditionnel de la sous-traitance de l'ensemble du système d'infotainment est modifié par une approche dans laquelle l'OEM choisit le système d'exploitation, l'environnement de développement et l'architecture matérielle. Mais bien que ces avantages soient attrayants pour les OEM, Android pose de nouveau le problème de la dépendance par rapport à Google et donc la méfiance de certains constructeurs. La France n'est pas en reste dans ce secteur avec des acteurs comme Parrot qui fut précurseur avec l'autoradio *Asteroid* (bien que techniquement dépassé dès sa sortie car sous Android 2.3) ou Coyote qui commercialise des produits d'aide à la navigation dont certains fonctionnent sous Android.

L'IoT (Android Things)

L'expansion d'Android sur le marché de l'embarqué a été évidente depuis 2012, mais a progressé plus lentement que certains l'avaient imaginé. Linux continue de dominer les secteurs de la domotique, de la santé et de l'industrie. Alors que l'énorme écosystème d'application d'Android est largement salué comme l'avantage essentiel pour les systèmes embarqués, la réalité du terrain montre que les principaux facteurs de motivation pour passer à Android sont le support du tactile, la réduction du temps de commercialisation des produits et une forte population d'ingénieurs maîtrisant la plateforme.

Google souhaite aborder le marché de l'IoT et de l'embarqué avec la même stratégie que les autres domaines cités. L'IoT est un marché naissant que l'on estime à plusieurs milliards de dollars où se ruent des dizaines de constructeurs sans réelle concertation, avec de nombreux systèmes d'exploitation, protocoles de communication et frameworks d'intégration. L'internet des objets est actuellement quelque peu cacophonique et Google souhaiterait y imposer sa loi avec Android Things. Ce dernier s'est joint début 2017 à la famille Android aux côtés d'Android TV, Android Auto et Android Wear. L'objectif d'Android Things est de rendre le développement de produits embarqués plus abordable car le système d'exploitation utilise les outils de développement de la plate-forme Android (Android Studio). Les entreprises n'ont cependant pas attendu l'arrivée d'Android Things pour utiliser Android dans leurs produits. Des acteurs majeurs comme JC Decaux ont utilisé Android pour des panneaux publicitaires « connectés » [53]. Plus récemment, la société Parkeon a développé une gamme d'horodateurs sous Android intégré à une solution complète utilisant l'écosystème Android. Le fait d'ouvrir la plate-forme

permet aux opérateurs locaux de développer leurs propres spécificités. Une présentation du produit est disponible sur une vidéo en [54].

Android Things n’est qu’une nouvelle étape dans la stratégie de Google pour aborder le marché de l’IoT. Google a également annoncé début mai 2017 le lancement de son offre cloud pour l’IoT. Google n’est cependant pas le seul – loin s’en faut - à proposer un OS pour l’embarqué et les objets connectés. Les plus populaires à ce jour sont Linux, Contiki et autres RIOT qui sont eux entièrement libres (mais fournissent uniquement un OS). D’autres concurrents de taille sont déjà sur la place comme Samsung avec Tizen et tout récemment Tizen RT annoncé lors du Tizen Developer Days en mai 2017. Il faut aussi compter avec ARM et son OS mbed ou des systèmes embarqués libres récemment adaptés à l’IoT comme FreeRTOS.

Conclusion

N’étant affilié à aucun éditeur ni constructeur et en nous appuyant sur notre expérience de plusieurs années dans le domaine de l'embarqué et de l'IoT, nous avons tenté dans ce livre blanc de présenter concrètement et sans complaisance les avantages et inconvénients de l’utilisation d’Android en dehors de ses domaines de prédilection actuels.

Google tente actuellement d’imposer son écosystème construit pour le smartphone et la tablette au marché de l’embarqué (au sens large). L’approche est cependant assez différente car des règles – et des leaders - existent depuis des décennies. Contrairement au marché des smartphones, Android n’est donc pas le « killer OS » comme on pouvait le penser il y a quelques années. Comme nous l’avons démontré dans ce livre, il reste cependant très bien adapté à des solutions graphiques mais reste parfois en deçà d’autres solutions (comme Linux) de part son approche de développement partiellement open source car centralisée par Google. Cependant, si le marché de l’IoT se développe comme prévu, l’arrivée d’Android Things pourrait faciliter l’utilisation d’Android dans un contexte industriel très différent de celui des smartphones.

Nous profiterons de la sortie officielle d’Android Things (d’ici fin 2017) pour réaliser un mini-livre blanc de quelques pages sur ce sujet ! A suivre !

Une question ? Un projet ? Rendez-vous sur Smile.fr rubrique Contacts !

BIBLIOGRAPHIE

[1] Apollo Guidance Computer et son simulateur en Javascript
https://en.wikipedia.org/wiki/Apollo_Guidance_Computer et
http://svtsim.com/moonjs/agc.html

[2] Calculateur D17-B https://en.wikipedia.org/wiki/D-17B

[3] RTOS/360
https://www.computer.org/csdl/proceedings/afips/1969/5073/00/50730015.pdf

[4] Exemples d'utilisations de RTEMS https://www.rtems.org/node/70

[5] Alliance GENIVI http://www.genivi.org/genivi-members

[6] Automotive Grade Linux https://www.automotivelinux.org

[7] Open Automotive Alliance http://www.openautoalliance.net

[8] Android Auto https://www.android.com/auto

[9] Android TV https://www.android.com/intl/fr_fr/tv

[10] AOSP https://source.android.com

[11] Android et la liberté des utilisateurs https://www.gnu.org/philosophy/android-and-users-freedom.fr.html

[12] Qt https://www.qt.io

[13] EFL https://www.enlightenment.org/about-efl

[14] Parts de marché Android fin 2016 https://qz.com/826672/android-goog-just-hit-a-record-88-market-share-of-all-smartphones/

[15] Project Yocto https://www.yoctoproject.org

[16] JBQ quitte le projet AOSP http://phandroid.com/2013/08/07/jbq-quits-aosp-qualcomm-to-blame/

[17] Ouvrage « Embedded Android » par Karim Yaghmour
http://shop.oreilly.com/product/0636920021094.do

[18] Utilisation des capteurs sous Android
https://developer.android.com/guide/topics/sensors/sensors_overview.html

[19] Carte BeagleBone Black https://beagleboard.org/black

[20] Noyau Linux pour AOSP http://source.android.com/source/building-kernels.html

[21] Images binaires Android 7.1 https://developers.google.com/android/images

[22] Images binaire Android Wear 2.0
https://developer.android.com/wear/preview/downloads.html#preview_system_images

[23] Projet BBB Android http://beagleboard.org/project/bbbandroid

[24] Projet Android4Beagle (similaire au précédent)
http://www.2net.co.uk/android4beagle.html

[25] Compilation AOSP http://source.android.com/source/requirements.html

[26] Projet Git https://git-scm.com

[27] Commande repo de Google https://source.android.com/source/using-repo.html

[28] Android 4.3 pour BBB (2net.com) http://2net.co.uk/tutorial/android-4.3-beaglebone

[28] Conférence Opersys « Embedded Android » http://www.opersys.com/blog/ea-videos-lca-2013

[29] Écran tactile LCD CAPE pour BBB http://fr.farnell.com/4d-systems/4dcape-43t/lcd-cape-module-bbone-black-dev/dp/2451217

[30] Présentation du « device tree » par Thomas Petazzoni
https://events.linuxfoundation.org/sites/events/files/slides/petazzoni-device-tree-dummies.pdf

[31] Utilisation U-Boot et Fastboot sur BBB http://www.2net.co.uk/tutorial/fastboot-beaglebone

[32] Interfaces matérielles pour Android « Things »
https://developer.android.com/things/sdk/pio/index.html

[33] Pilotage USB dans le SDK Android
https://developer.android.com/guide/topics/connectivity/usb/host.html

[34] Exemples de pilotage de périphériques USB
https://developer.android.com/samples/index.html

[35] Extension d'Android http://www.opersys.com/blog/extending-android-hal

[36] Native Development Kit (NDK) https://developer.android.com/ndk/index.html

[37] Exemples NDK https://developer.android.com/ndk/samples/index.html

[38] Accessing hardware on Android https://fr.slideshare.net/gibsson/ftf2014-hw

[39] Accès I²C depuis une application Android http://dangerousprototypes.com/blog/2012/10/26/i2c-communications-and-android-applications/

[40] Utilisation de ccache https://source.android.com/source/initializing#setting-up-ccache

[41] SDK preview Android Things https://developer.android.com/things/preview/index.html

[42] Système d'exploitation Contiki http://www.contiki-os.org/

[43] Système d'exploitation RIOT https://www.riot-os.org/

[44] Greybus pour IoT https://openiotelceurope2016.sched.com/event/7rrP/using-greybus-for-iot-alexandre-bailon-baylibre

[45] Android Things (Android for IoT) https://fr.slideshare.net/opersys/android-things-android-for-iot

[46] Interview K Yaghmour et C Simmonds sur Android Things https://www.linkedin.com/feed/update/urn:li:activity:6267693175582781440

[47] Conférence Google I/O 2017 sur Android Things https://www.youtube.com/watch?v=sjpNek_7z-I

[48] Article « the hidden cost of building an Android device » https://www.theguardian.com/technology/2014/jan/23/how-google-controls-androids-open-source

[49] Article « Google Play Services, le cheval de Troie de Google » http://www.frandroid.com/android/developpement/227959_google-play-services-cheval-troie-google

[50] Android compatibility https://source.android.com/compatibility

[51] Compatibility Definition Document https://source.android.com/compatibility/cdd

[52] Android wear vs Tizen https://www.strategyanalytics.com/access-services/devices/wearables/reports/report-detail/global-smartwatch-os-market-share-by-region-q1-2017

[53] JC Decaux et Android http://www.frandroid.com/android/246775_jcdecaux-lassaut-villes-connectees-jcdbox-espace-publicitaire-android

[54] Présentation Parkeon https://www.youtube.com/watch?v=UFX9NIprWG8