Institut Universitaire de Technologie de Caen
Licence Professionnelle Microcircuits, Cartes et Applications
Département Mesures Physiques
Année 2003-2004

Recherche et Développement Bios Libre
Mangrove LinuxBios

par Mathieu Deschamps

Community release version 2.0

Remerciements

Je veux remercier M. Laurent Texier et M. Stéphane Durand pour m'avoir fait confiance et m'avoir confié ce travail. Je remercie toute l'équipe de Mangrove Systems pour son accueil et l'équipe de développement pour l'ambiance sympathique du bureau dans lequel je travaille.

Je tiens à remercier toute la communauté LinuxBios et le LANL pour leur initiative, leur contribution et leur aide qu'ils ont apporté tout au long de mon stage et du projet Mangrove LinuxBios. C'est agréable de retrouver cet esprit de recherche et de penser qu'il est partagé au moyen du réseau.

Enfin, je remercie tous ceux qui, en toutes langues, ont promu et encouragé ce type de travaux et les personnes contribuant à leur développement et à leur réalisation car ils font ce qu'il y a de meilleur dans l'informatique libre.

Table des matières

LA NAISSANCE DU PROJET

C'est dans le cadre dans stage en entreprise que naît le projet Mangrove LinuxBios pour cible embarquée. S'il est clair que l'innovation est un moteur dans ce domaine d'activité, il n'est pas toujours facile de trouver des personnes à charge de la réaliser... Comme il n'est vraiment pas facile de trouver les moyens aidant à cette réalisation.

Aux limites de cette technologie de noyaux de systèmes d'exploitation et de logiciels libres, survit la logique propriétaire. Si ces systèmes d'exploitations tendent vers une ouverture, les codes très hautement liés à l'architecture matérielle qui servent uniquement à les initialiser, les BIOS[1], eux stagnent dans l'opacité. Ce n'est pas un gage d'efficacité, mais les grands fabriquants de ces logiciels coûteux peinent à suivre cette logique d'ouverture. La mutation de l'infrastructure informatique et du marché achève de les y pousser.

Déjà, la communauté scientifique, le monde OpenSource et quelques uns de ces fabriquants travaillaient sur des projets visant à ouvrir les possibilités des matériels, et à l'exploiter d'une façon plus efficace pour des tâches nouvelles. Dans cette optique, plusieurs projets existent depuis quelques années, tels OpenBios, Freebios, LinuxBios.

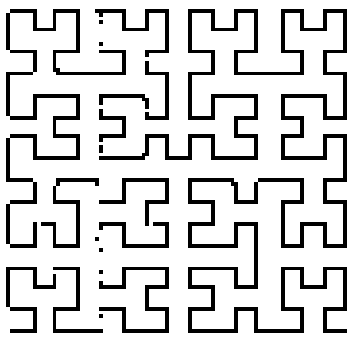
Mangrove voit l'occasion dans ces initiatives OpenSource d'éditer son propre BIOS libre et à terme de doter ses produits d'un nouveau plus, performant, qui conforterait et voire développerait sa présence sur le marché. La maîtrise verticale de l'ensemble logiciel des systèmes informatiques embarqué apporterait à Mangrove et à ses partenaires :

- Une liberté de choix dans les fonctions bios permettant aussi d'alléger le système à embarquer;
- Une réduction à néant du coût des droits d'exploitation des codes BIOS (comme Award ou Phenix) qui devient important au regard du prix de revient d'un client léger;
- Un temps d'initialisation plus rapide, et donc un temps de boot[2] plus court permettant qu'un système soit opérationnel en un instant;
- Une personnalisation dès les premières microsecondes du boot pour spécifier des offres sur mesure;
- Une possibilité de mise à jour commune à distance de l'OS[3] et du BIOS réduisant à peu de chose la maintenance de l'intégralité du système.

[1] BIOS : Basic Input Output System
[2] Boot : Au sens large, procédure d'initialisation du matériel et de chargement d'un système d'exploitation
[3] OS : Operating System, Système d'exploitation

Institut Universitaire de Technologie de Caen Licence Pro. M.C.A. Année 2003-2004
Mathieu Deschamps

L' ETUDE



I. Analyse du projet

a) Présentation

Le but de ce projet de recherche et développement Bios Libre est de faire démarrer un noyau Linux (par exemple LBT) sur un client léger avec une carte mère à base de chipset VIA depuis un Bios libre. Justement, LinuxBios propose de construire un BIOS et de le flasher[4] dans une puce.

LinuxBios est un projet mené par le LANL (Los Alamos Advanced Computing Laboratory) en étroite relation avec d'autre projets comme OpenBIOS et Freebios; en fait ils ont plus ou moins fusionné. Certains constructeurs et fabriquants de matériels y apportent une aide précieuse. Dans l'esprit OpenSource, une communauté d'utilisateurs et de développeurs apportent aussi leur concours en travaillant à but professionnel ou (également ?) à titre de loisir, sur le modèle collaboratif en utilisant le logiciel CVS (Concurrent Versionning System). LinuxBios a ainsi évolué il y a moins d'un an, Il a été complètement repensé en terme de structure. Le version résultante appelée V2 apporte plus de modularité, de portabilité et surtout de clarté.

Pour Mangrove, LinuxBios se situe déjà comme un objectif, les raisons tiennent au rapport potentiel/difficulté de LinuxBios, qui paraît en l'état actuel avantageux. Effectivement, c'est un projet qui présente des intérêts et des risques en différents endroits du développement.

D'ailleurs, depuis quelques années dans le cadre du projet Mangrove LinuxBios, plusieurs personnes ont essayé, dans les bureaux de Mangrove, de mettre en oeuvre LinuxBios. Si ce développement n'a jamais vu le jour, j'ai toutefois pu recueillir quelques documents autour du sujet. Ceci dit, je n'ai pu obtenir aucune implémentation, même pas une amorce. Dans ce contexte, les éléments qui manquèrent aux précédentes personnes et les récentes avancées dans le projet LinuxBios en général, doivent permettre de voir Mangrove LinuxBios sous un jour nouveau et de constater l'étendue de la tâche.

Il est apparu très rapidement que ce projet devait être complexe à mettre en oeuvre. Aussi, l'organisation et le découpage en tâche que je propose, devait respecter deux grandes étapes : l'analyse et la réalisation. Comme ce projet est inclus dans un plus vaste, il est vrai que parfois, la recherche se résume à une masse impressionnante de lecture et le développement à un déploiement. Grâce au travail collaboratif CVS et à internet, clairement, j'ai tenté de faire en sorte que ce résumé soit possible à chaque fois.

[4] Flasher : Procédé visant à écrire des données sur une puce de type EPROM ou EEPROM (Electronical (Erasable) Programmable Read Only Memory). Dans ce projet, les puces sont toutes EEPROM.

b) Organisation

La première tâche consistera à décrire le projet Mangrove LinuxBios dans le projet « père » LinuxBios initié par le LANL. Ensuite, j'aurai à identifier les risques lors du développement logiciel afin de devancer et prévoir les futures difficultés ainsi que gagner du temps à la maintenance future de ce projet. Enfin, il faut pouvoir apprécier la mesure des gains escomptés en regard des difficultés. (cf. étude de faisabilité)

La seconde tâche consiste à étudier les « trous fonctionnels », s'ils existent. En d'autres termes, il faudra regarder là où les solutions proposées par LinuxBios ne recouvrent pas l'intégralité des besoins et des fonctionnalités requises par Mangrove LinuxBios. Et il faudra y parer, avec une étude des solutions externes. Le souci de la maîtrise du temps dans ce projet est capitale, car il faut parvenir à un développement fonctionnel : il s'agit aussi de faire plus qu'une étude. (cf. étude de Linux et du BIOS, étude des solutions logicielles)

Étant donné qu'il s'agit d'une technologie de pointe, et bien qu'ouvert au-delà de la stricte utilisation en laboratoire, il semble clair qu'il n'y a pas abondance de solutions, en tout cas pas au sens général de moyens de mise en oeuvre. Par conséquent dans cette troisième tâche, je m'accorderai le temps nécessaire, en accord avec mon maître de projet, à tester et valider différents moyens pour parvenir à mettre en place telle ou telle solution. (cf. étude des solutions logicielles, mise en oeuvre LinuxBios)

La quatrième tâche est la plus productive car regroupant toutes les infos recueillies et les solutions validées en une implémentation LinuxBios sur mesure à Mangrove : Mangrove LinuxBios . A cette étape, il est temps de discuter sur d'éventuelles modifications au regard du calendrier et des fonctionnalités du prototype. Il faut revisiter les travaux de la première tâche. (cf. mise en oeuvre LinuxBios)

Un autre enjeu consiste à mettre en place un capital qui puisse servir de plate forme de développement de Mangrove LinuxBios. Grosso modo, des lignes de codes sources qui compilent et produisent un programme qui puisse être paramétré afin de produire un BIOS LinuxBios plus facilement qu'en l'état actuel. (cf. mise en oeuvre LinuxBios)

La dernière tâche, non la moindre, consiste à faire en sorte que ce développement puisse être repris, le prototype compris. Il faut documenter dans le source tree en anglais (des READMEs) et produire un document de synthèse en français : celui-ci.

remarque : Nécessité de synthèse du rapport, la dénomination première, deuxième, troisième tâche me dérange un peu car connote une chronologie stricte. Il apparaît qu'en réalité, pour certaines en tout cas, les tâches s'entremêlent et vivent tout au long du projet. C'est pourquoi un confère renvoie aux résultats des différentes études qui correspondent aux tâches décrites ici.

c) Moyens, Outils, Ressources

Les ressources attribuées à ce projet sont restreintes. Peut-être les justifications en sont que : la possibilité d'une implémentation est inconnue, les précédentes tentatives sont des ratés. Le projet père, en lui-même, donne une impression confuse de bric et de broc, de vide ou de foule, d'indéfini ce qui n'encourage pas vraiment les développeurs à s'investir et les utilisateurs à la confiance. Pour ce faire un ordre d'idée, le source tree[5] de V1 compte environ 800 sous-répertoires dont le répertoire src/ qui en compte à lui seul 590. A titre de comparaison, V2 qui a été réorganisé, lui compte environ 350 sous-répertoires dont 240 dans src/. Ceci explique, entre autres, que je me sois intéressé à ce dernier. En dépit de cela et par la suite V1 restera dans le source tree de Mangrove LinuxBios : il est un vivier de code source intéressant.

Voici également une autre difficulté : il s'agit pour moi d'une première expérience dans le développement au niveau du bios et de développement de noyau Linux. LinuxBios requière un niveau de connaissance poussé au niveau informatique, en électronique; des bases en 'histoire de l'informatique' et de son matériel sont également bienvenues. Un solide passif d'architecture notamment pour les bus est indispensable. Également, il est très utile, bien sûr, d'avoir de l'expérience dans la mise en place de solution OpenSource, les mécanismes des Makefiles ne doivent poser aucun problème. Par exemple, en l'état les source trees des deux versions confondues, ne présentent pas plus que quelques documentations. Il y a quand même quelques howto[6] succincts mais intéressants.

Les ressources les plus importantes se situent incontestablement dans la mailing list et la mail archive, un forum et l'archive de ce forum qui, pour ce projet, fait également office de serveur de ressources. Ce forum pipermail est utilisé ainsi : les utilisateurs testent les programmes, font remonter des rapports de bugs et souhaits de fonctionnalités. Les développeurs, qui sont souvent aussi des utilisateurs, font « des propositions de code » au mainteneur du projet (le LANL), l'aident dans le développement et répondent avec lui aux questions du forum. Les mainteneurs du projet conserve la maîtrise de son projet en discutant des choix stratégiques d'implémentation en agissant comme un steering comitee (comité de direction).

D'autres sources de donnée importantes sont le serveur de noyau Linux kernel.org, le site du GNU bien sûr, les sites « e-zine » sur linux, pour des infos d'ordre plus général. La recherche google/linux, la question au collègue restent également des valeurs sûres ! Enfin, la dernière source, c'est le code source bien sûr ! Particulièrement pour ce projet, le fameux RTFM[7] se trouvera remplacé par le non moins fameux RTSL[8] !

[5] Source tree : l'arborescence des sources du projet.
[6] Howto : documentation donnant les étapes à suivre pour faire fonctionner un logiciel.
[7] RTFM : « Read The F*****g Manuel » soit « lit le p****n de manuel »
[8] RTSL : « Read The Source Luke » soi « lit le code source Luke »

Tout au long d'un projet, il est nécessaire de faire des sauvegardes, en cas de pannes et erreurs de manipulation. A ce sujet, LinuxBios est un projet assez « dangereux », les manipulations électriques de la puce bios, les programmes à tester en mode superutilisateur, peuvent provoquer des dommages sur les disques durs, à la puce, etc.. Donc la sauvegarde est utile au point qu'elle figure presque en ressource ! Un script bash qui fait une snapshot[9] régulière fait tout à fait l'affaire si on n'a pas envie de s'installer un CVS root.

Voici la structure d'une source tree Mangrove LinuxBios telle qu'on l'on peut la trouver dans une snapshot (1er et 2éme niveaux de profondeur):

projet-xxxxxx/

- **freebios** : source tree LinuxBios projets V1 et V2. V1 est conservé car il sert de « réservoir » de code pour des portages vers V2. (certaines fonctionnalités de V1 n'existent pas encore sous V2)
- **doc** : la documentation que j'ai récupéré au sujet de LinuxBios et des documents autour de LinuxBios, Linux Kernel (le noyau linux).
- **log** : rapports d'erreur, requiérements pour faire f onctionner LinuxBios
- **src** : les sources pour Mangrove LinuxBios. Des bios, des images, quelques tar.gz autour du projet, fait office de dossier de téléchargement
- **utils** : sont regroupés ici des outils scripts ou binaire que j'ai écris, d'autre repris, un dossier important sans aucun doute.
- **env** : script d'environnement pour mettre en place rapidement une plate forme de développement

projet-xxxxxx/freebios/

- **freebios** : le source tree de V1
- **freebios2** : le source tree de V2
- **kernel** : contient des images des noyaux paramétrés pour LinuxBios
- **log** : contient différents fichiers config de noyaux testés.
- **spec** : quelques notes et spécifications de cartes mères
- **src/drivers** : pilotes pour Xfree, framebuffer, et les chipsets VIA
- **src/utils** : utilitaire de flashage lxbios, lbflash, etc..
- **src/backup** : contient des configurations de LinuxBios pour VIA

[9] Snapshot : « cliché » à un instant donné du développement en cours. Concrètement, un fichier archive compressé tar.gz.

Mangrove a mis à ma disposition deux machines pour mettre en oeuvre ce projet. L'une, la machine de tests appelée build machine, accueille le source tree LinuxBios, les compilent pour son environnement matériel et logiciel. L'autre appelée serveur, sert effectivement, via protocole sécurisé ftp avec le logiciel vsftp, des fichiers de code, de configuration, de noyau linux, etc..

Lors des sessions de tests pendant le démarrage du bios, LinuxBios envoie des chaînes de déboggage vers le port série (ttyS0 aka COM1) : c'est le mode debug. Elles ne peuvent être affichées sur l'écran de la machine de test car l'initialisation de la carte vidéo n'est faite qu'en fin du programme BIOS (l'écran peut rester éteint !). C'est pourquoi, un câble série null modem relie les ports série des deux machines. Les informations peuvent alors être récupérées sur le port de la machine serveur puis être affichées sur son écran.

Voici le schéma de l'appareil pour l'expérimentation :

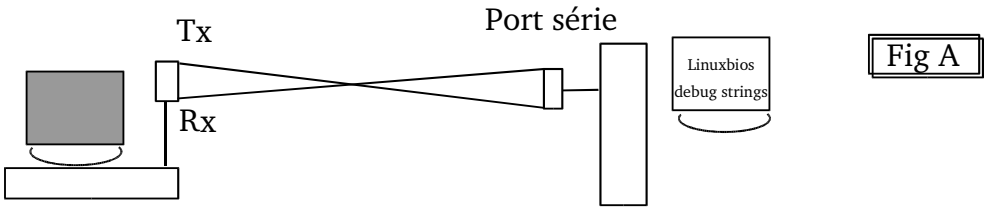


Fig A

note : Le câble figure croisé de sorte que la Tx soit branché sur la Rx (Tx : câble de transmission et Rx : câble de réception) (Cf. Annexe câble)

build machine : le client léger, la cible par définition

carte mère	: VIA EPIA M(Mini-ITX) 17cm sur 17cm
Processeur	: VIA C3/EDEN EBGA Processor, C3 1GHz
Chipset	: VIA CLE266 North Bridge , VT8235 South Bridge
PCI, ISA	: VT 8601, VT8231
IDE	: VT82C586 (2 X UltraDMA 133/100/66 Connector)
Memoire	: 1 DDR266 DIMM socket, jusqu'à 1GB memory size
VGA	: AGP graphics,MPEG Accelerator,Trident CyberBlade/i1
Extension	: 1 PCI
LAN	: VIA VT6103 10/100 Base-T Ethernet PHY
Audio	: VIA VT1616 6 channel AC'97 Codec
Divers	: Support USB et 1394.

Les documents de référence sur LinuxBios sont mentionnés dans la partie Référence à la fin de ce document. La lecture du AMD64 Port Guide est intéressant à bien des égards. (cf Références)

II. Réalisations

Tout au long de mes travaux visant à réaliser les tâches avec les moyens à ma disposition, je n'ai pas cessé de me documenter soit par une recherche d'existant soit par la création. Des références étayent cette partie pour signaler les travaux annexes, en revanche les documentations sur les différentes études sont présentes dans le source tree en doc/ et dans le reste du source tree au format de README.

a) Étude de faisabilité

Lors de la première tâche, j'ai dégagé les risques dans le développement du projet et ces potentialités. Cette étude est indispensable, car aucun de mes prédécesseurs ne l'avait fait. Or le plus gros risque pour un projet n'est il pas simplement qu'il soit irréalisable ? N'est-ce pas le rôle d'une étude de faisabilité de le déterminer ? Bien sûr, il y a bien des témoignages de succès tonitruants dans le forum, sans quoi ce projet n'aurait jamais été initié à Mangrove je suppose. Ceci dit parfois ces succès sont très humbles. Peut-être aura-t-on préféré ramener le but initial à quelque chose de plus raisonnable ? Au moins, une étude de faisabilité doit recueillir ces témoignages, constater ce décalage et analyser les risques lors du développement logiciel en rapport au potentiel du projet.

Les Risques

- Récence :

S'il naquit en 1999, LinuxBios n'a pas encore une crédibilité à sa mesure, certains tests n'ont pas été très poussés, certaines fonctionnalités ne sont disponibles que récemment et sont pour le mieux disponible qu'en version instable. Les besoins sont très mouvants et les implémentations se font et se défont. Il existe deux versions majeures différentes V1 et V2. Cette dernière répond principalement au besoins de mise en oeuvre de LinuxBios sur des cartes mères récentes.

La parade à cette difficulté consiste à faire une veille technologique constante afin de récupérer les correctifs (patchs) et les mise à jours. Il convient également d'opérer des tests longs et poussés.

- Obsolescence :

Certaines composantes de LinuxBios qui ne sont plus plébiscitées par les utilisateurs sont laissées à l'abandon. Le faible succès peut provenir de dysfonctionnement de ce composant, d'une difficulté de mise en oeuvre, ou tout simplement parce que plus personne ne sait encore à quoi il sert ! Mais si ce composant x est présent dans le source tree c'est qu'il a eu son temps de gloire. Comment faire pour ne pas se tromper et faire le mauvais choix (quand il existe) du composant que plus personne ne maintient ?

La veille autour du projet s'intéressera alors également à savoir qui maintient ces diverses composantes : regarder les changelog[10] est révélateur (quand l'auteur d'une composante a jugé bon d'en écrire un)

Une parade visant à réduire au minimum la taille de l'infrastructure selon l'idée « moins de composantes, moins de mise à jour » permet encore de réduire la complexité des portages de la version V1 vers V2.

- Isolement :

Il est clair face au constructeur de matériel et logiciel en position de quasi-monopole. C'est eux qui détiennent les infos sur leur matériels, évidement un logiciel tel que le BIOS, doit « connaître » très bien la couche matérielle afin de l'initialiser. Les informations techniques sur la structure des chipsets est quasi-impossible à obtenir, même si on est très gentil... et même si ce constructeur prétend collaborer au projet. Une anecdote à ce sujet peut être utile à quiconque s'intéressera à ce projet. La technique, qui est la plus employée par ces derniers pour rendre difficile l'accès à ces infos, consiste à noyer l'info dans des documents de communication commerciale d'apparence technique.

La parade s'il en existe, consiste à travailler avec les constructeurs partenaires du projet puis repérer les gens qui y travaillent dans la maillist. Ceci dit, pour la VIA EPIA M qui est depuis longtemps étudiée dans LinuxBIOS, la majeure partie des caractéristiques est déjà connue. Si une personne mentionne un N.D.A[11], mieux faut passer son chemin, car la moindre fuite de sa part pourrait entacher une ou plusieurs personnes de la communauté (personne 'tainted' littéralement entachée).

Un autre vecteur d'isolement vraiment paradoxal vient du principe de travail collaboratif. Il faut réussir à échanger des posts[12], tous rédigés en anglais bien sûr, avec des gens qui travaillent par exemple au Etats-Unis (le LANL est là-bas !) et à Taiwan (pour les fabriquants de microélectonique et les développeurs en ce domaine). Le décalage horaire va de moins six à plus huit heures. Planifier des questions futures est très difficile !

[10] ChangeLog : fichier texte simple, daté, il retrace les évolutions d'une composante.
[11] N.D.A : Non Disclosure Agreement, clause de confidentialité, de non divulgation.
[12] Posts : messages envoyés sur un forum.

Les potentialités

- Rapidité :

LinuxBios réduit grandement le temps de boot (1 à 10 secondes jusqu'à l'OS). Il active juste ce qui est strictement nécessaire au kernel[13].

Le projet suit un développement de type OpenSource collaboratif qui accélère vraiment le temps de développement. Les composantes qui demandent une compétence sont prises en charge et écrites par des spécialistes, des professionnels, et aussi des codeurs passionnés.

- Adéquation :

Il est important de signaler que LinuxBios offre les services bios équivalents aux services des bios constructeurs tout en épurant les codes rigides et spécifiques souvent contre performants. Il est typiquement fait pour booter Linux mais il permet, combiner à d'autres solutions, de booter Windows CE, Windows NTE[14] et aussi Plan9.

- Maîtrise :

LinuxBios offre une maîtrise entière sur le bios permettant un contrôle total de bout en bout du processus de chargement du BIOS jusqu'à OS. Cela induit une réduction des frontières entre les deux organes, ce qui repère les redondances et donc autorise optimisations et gains de performance et de place. L'idée est de réduire le rôle du BIOS au minimum, en l'état actuel le kernel fait une grosse partie des tâches faite par les BIOS. Or certaines de ces tâches ne devraient pas rentrer dans les attributions du BIOS. Ce glissement des rôles engendre ente autre des limitations de capacités conduisant à faire des mises à jour fréquentes et des rachats de matériel coûteux.

L'accès au code source du bios depuis l'OS permet des « tests en place » et également des mises à jour de bios à distance en même temps que l'OS. Ce qui réduit les coûts de maintenance.

[13] Kernel : Noyau du système d'exploitation Linux.
[14] Windows CE, Windows NTE : OS pour l'embarqué. Windows est copyrighté, il est la propriété exclusive de Microsoft.

- Personnalisation :

Un tel accès dès les premières microsecondes de boot permet de développer une capacité de personnalisation encore plus grande encore plus « tôt » :

- Modules d'authentification
- Splash screen[15] au couleur d'une société
- Sécurisation machine & réseau
- etc...

En outre, le degré élevé de paramétrage de LinuxBIOS permet bien sûr de booter sur disque dur, CDROM et disquette mais aussi et surtout de la Compact Flash ou « disque »DoC (DiskOnChip) ou DoM (DiskOnModule).

- Réduction :

Comme cela a déjà été abordé, LinuxBios ménage de l'espace libre en mémoire BIOS ce qui tend vers les objectifs de l'embarqué. De plus, si le stockage de l'OS se fait sur DoC ou DoM, il est possible de loger le BIOS juste avant en début de composant afin que LinuxBios le détecte. La réduction du nombre de mémoire, une ROM CMOS (BIOS) et une compact flash (OS) contre un DoC(BIOS+OS), permet en définitive une réduction du coût de revient par machine.

Enfin, la réduction de coût devient plus évidente encore puisque sous Licence GPL LinuxBios ne requière que le coût du développement initial, il y a plus de droit d'exploitation à payer au fabriquants de puces CMOS et éditeur de logiciel BIOS.

En conclusion de cette étude, il semblerait qu'effectivement LinuxBios possède des arguments imbattables au niveau technique. Les risques se concentrent clairement autour des moyens. Par la suite, je mentionnerai seulement LinuxBios, il est acquis que l'étude porte sur V2. L'autre connaissance indispensable est aussi le grand rôle du BIOS : les fonctionnalités utiles du BIOS pour mettre les ressources en état d'être activées et disponibles pour l'OS.

[15] Splash screen : bannière ou écran de présentation qui surgit est reste affiché un instant.

b) Étude de Linux et du BIOS

Dans cette partie, j'étudie la relation Linux-LinuxBios. LinuxBios, dans son rôle et ses principales actions est vraiment un bios comme les autres, alors ce qui est décrit ci-dessous est tout à fait général. Tout d'abord, je vais décrire des généralités sur le bios, expliquer dans le détail ce qu'il fait, ce dont il est capable, ce qu'il offre. Ensuite, il sera question de la spécificité de Linux dans la gestion des périphériques vis à vis du BIOS, et de frontière entre le BIOS et OS. Enfin, un diagramme synoptique permettra de considérer l'ensemble de la séquence de boot et cette étude se terminera par mes travaux sur le noyau Linux.

Généralités

Il est indispensable d'aborder quelques notions générales avant de comprendre le comportement d'un bios et les interactions qu'il a avec un OS.

Le bios est stocké dans une EEPROM appelé encore CMOS de taille variable généralement 128Ko ou 256Ko. Cette EEPROM est maintenue par une pile de secours en cas de panne d'alimentation. La procédure de « vidange manuelle » de cette mémoire consiste à couper l'alimentation de la carte mère (station éteinte ET débranchée donc) à retirer cette pile et à patienter une dizaine de seconde avant de la replacer. C'est utile si le programme dans le BIOS ou le paramétrage de ce programme est incorrect. Au redémarrage, la partie ainsi effacée sera détectée par checksum (incorrect) et déclenchera la procédure de secours qui récupérera les valeurs par défaut dans une ROM constructeur sur la carte mère (« Checksum error --Hit F1 to restore default values »). Dernière chose à savoir, la ROM bios est souvent recopiée en RAM (en shadow RAM[16])

Lorsque que le système est mis sous tension, le processeur ou CPU teste son bon fonctionnement (voltage, fréquence) et émet un signal *PowerGood*. Il rend actif les cartes conforme PnP[17] et tous les contrôleurs de la carte mère. Le CPU s'initialise avec des valeurs de base et sait qu'il doit aller lire l'adresse de la mémoire en 0FFFF:FFFF0 et se rendre à l'adresse qu'elle contient : *c'est la première instruction du bios*. Le bios détermine la mémoire cache de niveau 1 du CPU (CPU L1 cache) puis le niveaux 2 (CPU L2 cache). Il teste les coprocesseurs ou « compagnion chip » par exemple, sur processeur INTEL, il teste la présence du coCPU arithmétique grâce à une instruction dédiée. Tout ce début de la procédure s'est déroulé dans le *mode réel* la suite se déroule en mode *protégé*.

[16] Shadow RAM : partie de la mémoire RAM réservée au programme s'y recopiant afin principalement d'être exécuter plus rapidement

[17] PnP : « Plug and Play » est un standard décrivant des procédures et des spécifications matérielles et physiques.

Le mode réel est le mode d'adressage simplifié qui s'étend sur 16 bits. Comme il ne peut pas adresser toute la mémoire il faut passer en mode protégé et son adressage sur 32 bits. En ce mode, c'est le processeur qui est garant des conversions 16-32.

Maintenant que le mode protégé est déclenché, le bios doit réinitialiser le CPU avec des valeurs plus performantes dite « optimales » afin d'initialiser les périphériques, ces composantes du matériels autour du processeur. Le bios pourra enfin obtenir l'identification du CPU grâce à une instruction assembleur CPUID. Au retour de cette instruction, les registres du CPU sont initialisés avec les valeurs adéquates à son modèle de fonctionnement.

Ce sont les routines d'initialisation des périphériques qui prennent le relais, le bios envoie des octets d'initialisation aux contrôleurs de ce matériel. Pour faire simple, tous les périphériques possèdent des contrôleurs. Un contrôleur sert d'interface de dialogue entre un matériel et le BIOS et parfois même (pour Linux en tout cas) entre le matériel et l'OS.

Un bios actuel est capable de gérer:

- Les entrées/sorties. C'est là son attribution la plus importante. C'est précisément du dialogue avec les contrôleurs pour écrire ou lire sur un périphérique dont il est question. Par exemple, la carte graphique, audio, les ports série, le clavier, l'écran ou encore les disques durs.

- Les mémoires de masses (disques durs, disquettes, etc..). Il indique sur quel disque dur il ira chercher l'OS. Il lit la géométrie du disque dur dans une ROM et fait un test basique d'accès.

- Les mémoires internes, la RAM[18] dont la quantité est calculées en même temps que sa vitesse d'accès. Le bios indique également le mode de fonctionnement de la mémoire cache ainsi que l'adresse des tables utiles au microprocesseur.

- Le timer et l'horloge interne et donc, par extension, la fréquence de fonctionnement de la carte mère et donc la gestion de l'économie d'énergie ACPI (mise en veille de composant inactif depuis un certain temps) qui s'applique exclusivement aux périphériques. En l'état actuel LinuxBios V2 ne gère pas encore correctement ACPI, il est désactivé.

- Les bus de communications reliant périphériques : PCI et AGP sont gérés par LinuxBios. Il détermine leur largeur et leur vitesse de fonctionnement.

[18] RAM : Random Access Memory : Mémoire de travail indispensable au fonctionnement, dite « vive ». Vidée au reset (redémarrage de la machine). Une partie des mémoires RAM actuelles est auto-maintenue (non vidée)

Hormis ce rôle de grand initialisateur, le bios met à la disposition des ressources, ou plus exactement, les passerelles pour y accéder. Ces ressources vont être consultées puis utilisées par l'OS. Notamment le bios :

- entretient des tables de descripteurs locaux et globaux (LDT[19] et GDT), de vecteurs d'interruptions (IDT) et des sélecteurs s'il est en mode mode réel. Au passage au mode protégé, il doit établir des tables de pagination pour la gestion avancée 32 bits de la mémoire : ce sont, en quelque sorte, des tables de conversion et de correspondance.

- assigne directement les périphériques à un espace mémoire en RAM (canaux DMA[20]). Libérant alors la charge de l'intermédiaire qu'est le CPU.

- assigne ces périphériques à une référence (IRQ[21]) qui désigne un code d'interruption permettant d'accéder en lecture/écriture au contrôleur d'interruption (PIC).

- dispose le tas (heap) et la pile (stack) permettant de faire fonctionner des programmes de seconde génération qui les utilisent. Au lieu du code Assembleur[22] , il est alors désormais possible d'exécuter du code C[23].

Une fois que l'architecture matérielle n'a plus de secret pour le système, le bios recherche un OS. Selon un ordre de boot qui lui est fourni en paramètre (le boot sequence), le bios va chercher le bloc MBR/PBR[24] sur le disque dur, la disquette, etc..

Il regarde s'il contient dans ces deux derniers octets la valeur marqueur 0xAA55h qui signalise qu'il s'agit d'un secteur « bootable » et donc d'un disque qui contient un OS. Le bios recopie alors ce secteur en RAM et donne l'instruction au CPU de pointer sur son adresse en RAM. Ce secteur contient un code exécutable qui charge et décompresse le noyau Linux (s'il s'agit de l'OS GNU/Linux) à la suite en RAM. Puis ce secteur réinitialise le pointeur de pile et le CPU donne la main à l'OS. C'est la finalité du BIOS.

Un diagramme synoptique résume ses étapes. Il reste toutefois un point délicat à aborder avant cela. S'il vrai que les périphériques sont mis en état de fonctionnement et reconnus par le BIOS, il n'en est pas toujours de même pour les différents et plus nombreux OS. Il faut que le BIOS et l'OS reconnaissent ces périphériques et puissent les utiliser.

[19] GDT,LDT,IDT : Global Descriptor Table, Local Descriptor Table, Interupt Descriptor Table.
[20] DMA : Direct Memory Assignment. Progammed Input/Output (PIO) est concurrent à ce mode.
[21] IRQ : Interrup ReQuest. Requête d'interruption. Déprécié depuis la norme PnP.
[22] Code Assembleur : Code alpha (1er lettre de l'alphabet grec), c'est le code de première génération.
[23] Code C : Code C ... c'est le code de seconde génération. (Le code B n'ayant pas existé).
[24] MBR/PBR : le Master Boot Record, Partition Boot Record fait 512 octets.

L'abstraction du matériel

A la différence d'autre OS, Linux va rajouter aux codes d'interface et de dialogue contrôleurs-bios, une interface directe contrôleurs-Linux. Mais dans tous les cas ces codes s'appellent les pilotes de périphériques (devices drivers) c'est véritablement eux qui discutent avec les contrôleurs du matériel. Schématiquement, par exemple, une requête de lecture de fichier depuis une application d'un OS non-Linux peut se concevoir ainsi :

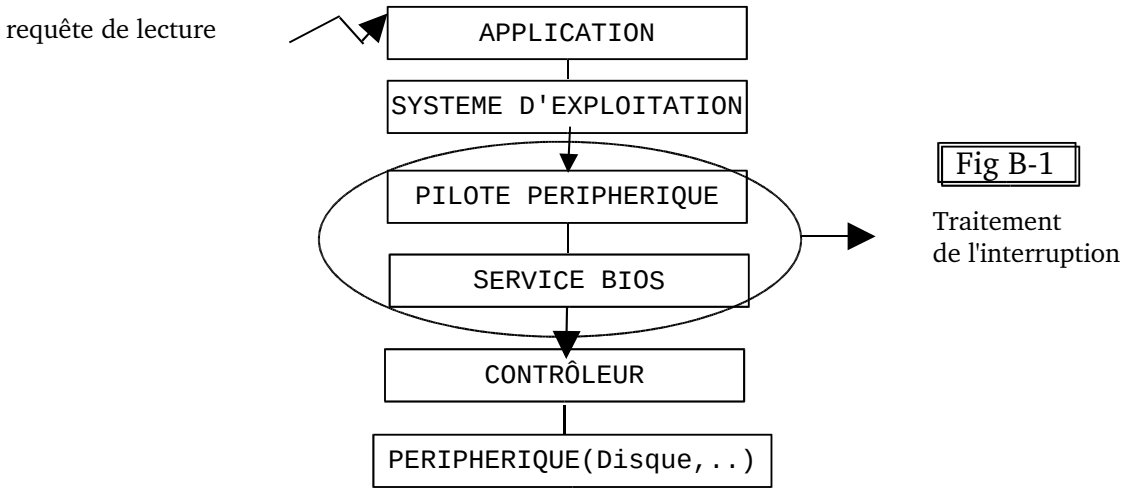


Fig B-1

Tandis que sous Linux, qui implémente un niveau d'abstraction supplémentaire, le (virtual) device driver, permettant à l'OS de dialoguer avec cet interlocuteur unique (un mappage unique, ,une centralisation des « signaux » , etc..) le schéma peut se réduit à ceci :

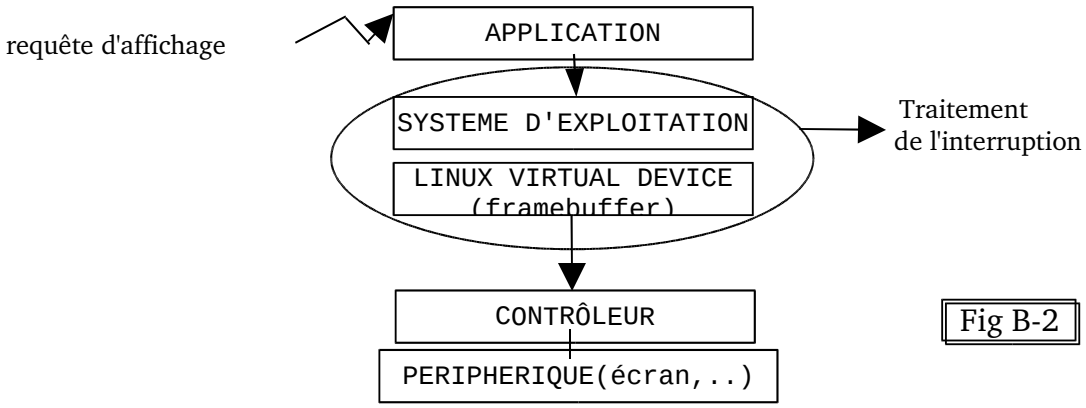


Fig B-2

Cela améliore les performances (temps de réponse, débit de pile E/S, etc..) mais quoiqu'il en soit un bios reste dépendant du matériel dont il a la charge. Linux offre une flexibilité plus grande dès que des solutions d'abstractions existent telle ce pilote d'affichage framebuffer. Du reste, c'est LinuxBios qui dispose ces solutions.

Le choix de la limite BIOS-OS

Sans le bios, ou plutôt s'il était entièrement intégré à l'OS, l'OS serait entièrement dépend du matériel. Ne semble-t-il pas inconcevable qu'il faille changer totalement un OS parce qu'on a juste changé de carte son ? Si bien sûr... d'où les bios.

La plupart des OS pour résoudre, entre autres, cette problématique, ont pratiqué une redirection totale des requêtes selon l'idée « ce n'est pas mon travail ». Elles passent par toutes les couches qui assurent à leur passage la validité de l'état de ces requêtes afin d'être décodées. Aussitôt décodées, elles sont découpées en sous-requêtes qui sont reconditionnées et dirigées vers la couche suivante où elles subissent les mêmes opérations (Fig A). En fait, ce n'est pas tout à fait le cas et c'est encore moins valable pour Linux.

Par exemple en faisant une redescription de l'IDT, Linux empiète, certes, sur le domaine du BIOS. Également, ceci prend le temps qu'il faut, une bonne fois pour toute, au chargement du noyau. Mais ce faisant, il propose de libérer le BIOS du coût que représente le dialogue contrôleur-bios lors d'une interruption. Non seulement, cela se traduit par une économie non négligeable du temps de calcul du processeur *avant* qu'il ne soit interrompu, mais en plus, cela offre à l'OS la maîtrise du mode de déclenchement de ses interruptions (Fig B-2) et leur planification au noyau. Puisque les interruptions contrôlent tout du traitement des opérations liées aux matériels, tous les périphériques sont affectés et gagnent en efficacité.

Ceci n'est qu'un exemple parmi d'autres tel aussi l'initialisation MTRR[25] mais cela tend néanmoins à souligner que, ce « flou » entre les frontières BIOS-OS, s'il ne facilite pas dans un premier temps la compréhension, à le mérite d'être vraiment efficace. Dans un deuxième temps, cela est vraiment paradoxal, étonnant et pourtant ré-écrire du code, impieter sur les attributions du BIOS permet un gain de performance dans ce cas. Il faut peut-être trouver cela normal ou « classique » après tout, la double initialisation du CPU en mode réel puis en mode protégé permet aussi une définition plus fine, improbable, avant. Enfin, dans un autre domaine celui de « l'assurance du minimum de fonctionnement » que dire des dispositifs doublons de type failsafe[26] ou fallback[27] ? Ils sauvegardent le système, sont les derniers remparts avant le « kernel panic » ou la « fatal error »[28]

Il est difficile de définitivement établir une frontière, les applications potentielles sont nombreuses, s'accélèrent. Il faudra certainement redéfinir cette frontière à l'usage de ces applications, pour l'instant si elle reste ouverte cela restera possible. La véritable limite sera fixée à l'usage sans intention « de chasse-gardée ».

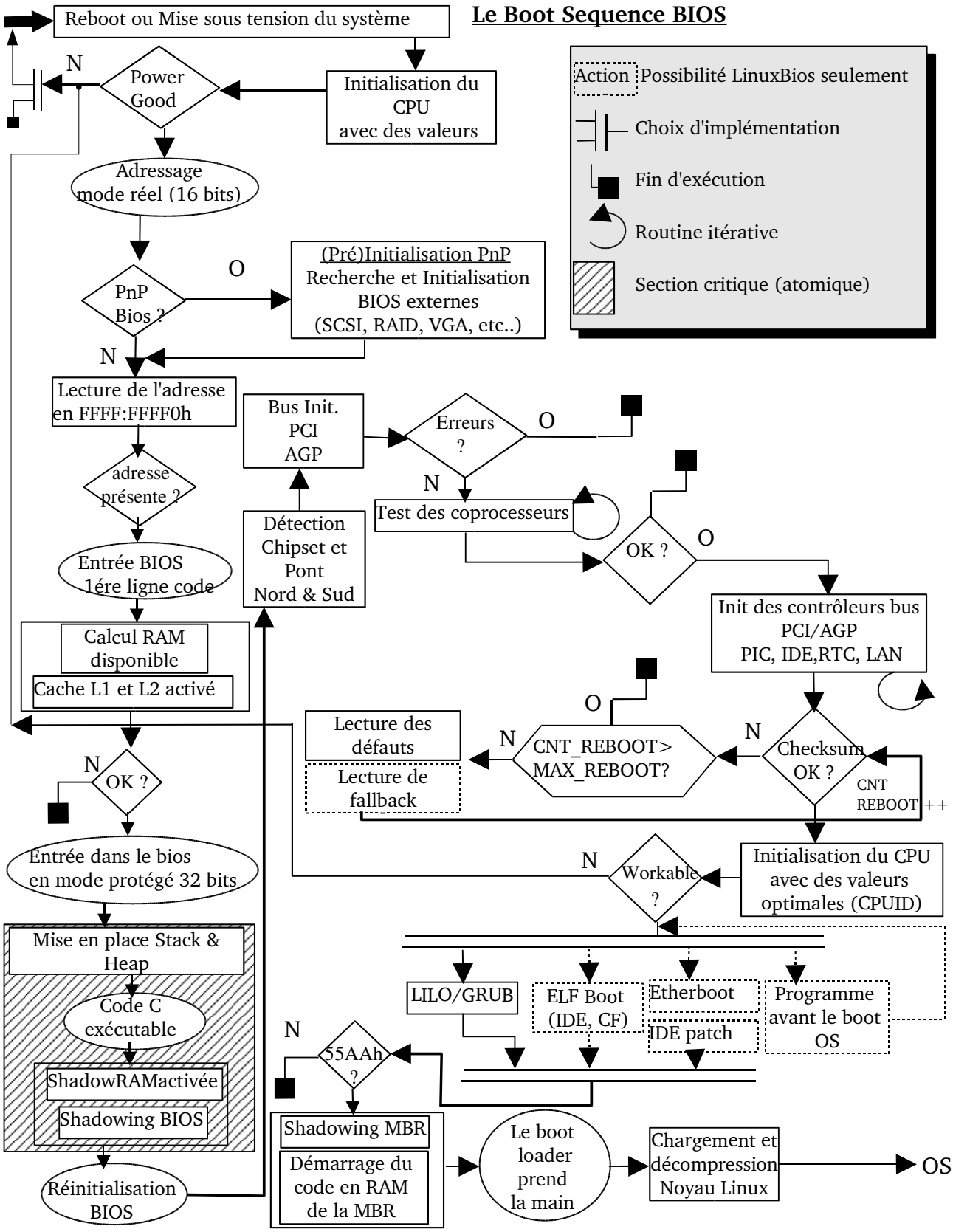
---

[25] MTRR : Memory Type Range Register, grosso modo, ce sont des registres pour l'initialisation et le dialogue des bus PCI et AGP avec la carte vidéo par exemple. Permettent un gain de performance notable.

[26] Failsafe : Mode de fonctionnement minimal du système, permettant la maintenance afin de restaurer le système en mode normal.

[27] Fallback : Mode de fonctionnement minimal du système LinuxBios, si l'image ROM normal du BIOS ne peut être chargé, dès la moindre erreur, alors c'est ce mode qui est activé.

[28] Kernel panic, fatal error : erreur système majeure, incontournable, incurable, parfois imprévue.


Le Boot Sequence BIOS
Reboot ou Mise sous tension du système
Power Good
N
Initialisation du CPU avec des valeurs
Action
Possibilité LinuxBios seulement
Choix d'implémentation
Fin d'exécution
Routine itérative
Section critique (atomique)
Adressage mode réel (16 bits)
PnP Bios ?
O
(Pré)Initialisation PnP
Recherche et Initialisation BIOS externes
(SCSI, RAID, VGA, etc..)
N
Lecture de l'adresse en FFFF:FFFF0h
adresse présente ?
Entrée BIOS 1ére ligne code
Calcul RAM disponible
Cache L1 et L2 activé
Bus Init. PCI AGP
Erreurs ?
O
N
Détection Chipset et Pont Nord & Sud
Test des coprocesseurs
OK ?
O
Init des contrôleurs bus PCI/AGP PIC, IDE,RTC, LAN
Checksum OK ?
N
CNT REBOOT++
CNT_REBOOT> MAX_REBOOT?
O
N
Lecture des défauts
Lecture de fallback
OK ?
N
Entrée dans le bios en mode protégé 32 bits
Workable ?
N
Initialisation du CPU avec des valeurs optimales (CPUID)
Mise en place Stack & Heap
Code C exécutable
ShadowRAMactivée
Shadowing BIOS
LILO/GRUB
ELF Boot (IDE, CF)
Etherboot
IDE patch
Programme avant le boot OS
55AAh ?
N
Shadowing MBR
Démarrage du code en RAM de la MBR
Le boot loader prend la main
Chargement et décompression Noyau Linux
OS
Réinitialisation BIOS

Le Noyau de Linux

Oeuvre de Linus Torvals, le noyau Linux choisi durant ces tests expérimentaux est le 2.4.25 : le dernier noyau de la série 2.4.xx. C'est une contrainte qui m'a été fixée, les produits logiciels Mangrove, bien que portés vers la série 2.6.xx, reposent encore en partie sur des composantes compatibles avec la série 2.4.xx et les clients ne sont pas tous prêts à la migration. De plus en l'état actuel, LinuxBios également connaît ce genre de contrainte.

Tout au long de mes travaux, j'ai recompilé et testé des noyaux. Il est conseillé de faire des tests de régression(cf. Kernel regression) pendant au moins 72 heures. Les différentes configurations que j'ai testées sont présentes dans le source tree.

L'OS est tout à fait indépendant du BIOS, toutefois comme abordé auparavant Linux refait une partie de l'initialisation faite par le BIOS à des fins d'efficacité.

Cette plus grande indépendance requière donc une source de données extérieures au BIOS : ces données sont présentes dans le noyau. Elles ont été installées lors de la construction du noyau. Pour ce faire, il faut les sources du noyau, et un fichier de configuration paramétré pour le matériel qu'on décide que Linux (ré)initialisera. Les documentations dans les sources aident à la configuration. Néanmoins, la précaution prime et la méthode par tests unitaires des fonctions à installer est toute indiquée bien que coûteuse en temps. Un noyau qui « crash » (qui bloque la machine) entraîne toujours au minimum dix à vingt minutes de maintenance pour retourner dans l'état de fonctionnement antérieur sauf si on a une procédure de dépannage bien rodée (cf. Annexe kernel help!).

Pour compiler un noyau Linux, les composantes logicielles de versions au moins égales aux suivantes sont requises pour ne pas avoir de surprise :

Composante	Version requise	Version courante
Gnu C	2.91.66« egcs 1.1.2»	gcc --version
Gnu Make	3.77	make --version
Binutils	2.9.1.0.25	ld -v
util-linux	2.10o	fdformat --version
modutils	2.4.0	insmod -V
e2fs progs	1.19	tune2fs --version
pcmcia-cs	3.1.21	cardmgr -V
PPP	2.4.0	pppd --version
isdn4k-utils	3.1 beta 7	isdnctrl 2>&1 | grep version

Pour faire fonctionner la carte mère VIA EPIA décrite auparavant dans la partie Moyens, il faut par exemple installer entre autres le chipset VIA CXX, le CPU C3. Pour pouvoir envoyer les chaînes de déboggage vers le port série, il faut activer également une option. Ces options peuvent être activées de deux façons : soit en tant que modules, intégrés au noyau Linux dès « qu'on veut ! » soit en tant que composante fusionnée au noyau et donc son chargement se fait en même temps que le noyau.

Bien sûr, plus on ajoute de composantes dans le noyau plus il devient gros : ajouter en tant que module permet d'avoir des noyaux compacts et petits qui se chargent plus vite. Typiquement voici les commandes qui permettent de construire un noyau :

- make config
- make depend
- make bzImage
- make modules
- make modules_install

Les deux dernières commandes permettent de construire et d'installer les modules. Par défaut, les modules sont installés en /lib/module/<versionKernel>.

Dès lors on peut insérer un module dans le système avec :

```
insmod /lib/modules/2.4.25/kernel/drivers/net/via-rhine.o
```

Le module installé dans cette exemple est le module du pilote réseau dirigeant le contrôleur réseau VIA RHINE intégré au chipset de la VIA EPIA. Si il a été possible de faire ceci c'est que, dans le fichier de configuration noyau, l'option CONFIG_VIA_RHINE était paramétrée en tant que module :

```
# paramétrage du pilote réseau pour le contrôleur rhine en tant
# que module
CONFIG_VIA_RHINE  = m
# paramétrage du pilote réseau pour le contrôleur rhine en tant
# partie intégrante du noyau, insmod impossible alors.
# CONFIG_VIA_RHINE  = y
```

Également l'ajout d'un support DiskOnChip (DoC) dans le noyau requière déjà un patch des sources Linux. Voici quelques paramètres intéressants à initialiser :

CONFIG_MTD_DOC1000	Prise en charge DoC 1000
CONFIG_MTD_DOC2000	Prise en charge DoC 2000/ME
CONFIG_MTD_DOC2001	Prise en charge DoC 2001
CONFIG_MTD_DOCPROBE	Autoscan DoC sur le système
CONFIG_MTD_DOCPROBE_55AA	Permet de faire booter une DoC

LinuxBios rajoute lui-même quelque lignes dans ce fichier de configuration, tandis que pour éviter certains conflits (ACPI par exemple) il est parfois utile de désactiver certains paramètres (manuel !). Quoiqu'il en soit, un mauvais paramétrage ici n'affectera pas le démarrage du BIOS. Vraiment au pire, l'erreur affectera le bootloader[29] au moment du chargement et de la décompression du noyau.

Le bootloader c'est le petit code binaire qui est dans la MBR il est chargé à la fin du boot sequence. Ce code, ce peut être les classiques LILO[30] ou GRUB[31] ou n'importe quel programme au format elf ou a.out, c'est selon qu'il a été paramétré dans le noyau. Une partie suivante de l'étude reviendra sur les différents *bootloaders* qu'il est possible de combiner avec LinuxBios.

Le bootloader lui aussi est indépendant, dans son code la partition racine *root* du système peut être définie par défaut. Alors lorsque le noyau démarre il connait cette partition indispensable.

Il est possible d'indiquer au booloader en ligne de commande (s'il la propose : option prompt) que l'on veut prendre une autre partition racine root, c'est un simple passage de paramètres texte. En fait plus exactement, les paramètres du bootloader sont recopiés dans les paramètres du noyau.

Donc, il est possible via le bootloader de donner très tôt au noyau, des instructions au travers de ce type de paramétrage. Les paramètres doivent être espacés sur la ligne de commande.

Voici ces quelques exemples dédiés LinuxBios :

root=/dev/hda3	Indiquer la partition racine
console=/dev/ttyS0	Activer E/S sur le port série comme console
vga=ask	Demander le mode ligne vga désiré

[29] Bootloader : Chageur de système d'exploitation.
[30] LILO : LInux LOader.
[31] GRUB : GRand Unified Bootloader.

Dans la pratique, la procédure du chargement du noyau se fait en deux étapes. D'abord le bootloader charge un kernelstrap ou morceau de noyau, un message s'affiche du type « In first stage loading ». Dans ce premier temps, le pré-noyau installe le VFS[32] pour monter la partition root qui contient s'il on ose dire la seconde image du noyau. Ce détour est obligé, la taille de 512 octets de la MBR ne pourrait accueillir tout le noyau !

Ensuite, c'est ce noyau qui « prend la main » et c'est là que tout est fait. Dans ce second temps, un message « Second stage loading » s'affiche et sont initialisés les pilotes périphériques. A ce moment, le noyau peut remonter sa partition root en lecture/écriture (option remount,rw). Les informations provenant du BIOS sur le paramétrage du matériel sont stockées dans des structures pci_dev du type:

```
struct pci_dev
        {struct pci_bus *bus;
         struct pci_dev *dev;
         struct pci_dev *next;
                ...
         unsigned int devfn;
         unsigned short vendor;
         unsigned short device;
         unsigned int irq;
         unsigned long base_adress[6];}
```

Toutes ces étapes ne peuvent être suivies qu'au travers des chaînes de déboggage ou avec une carte PCI POST. Un mode de LinuxBios met en place une gestion POST[33].

Ensuite, s'il y a un Ramdisk[34], encore une option du noyau, il est exécuté. Un pilote de périphériques a mis en place une console principale. A partir de cette console, ce même pilote peut créer d'autres consoles mais virtuelles. Enfin, un script bash du type rc.sysinit s'exécute et prend la main dans un terminal. Accessoirement, le processus INIT peut démarrer, il met en place la mémoire d'échange (swap), démarre les serveurs et démons[35]. Enfin, le serveur X peut être initialisé et affiche un gestionnaire de bureau et de fenêtres. De là, on interagit avec le système suivant les standards du mode graphique fenestré.

Compte tenu de ces éléments issus tantôt du BIOS et tantôt du Kernel, des fonctions qui feront l'intermédiaire dans LinuxBios peuvent être dégagées et étudiées. Les dépendances, induites de cela, doivent faire l'objet d'une attention particulière. Et la clarification de la structure de LinuxBios est une étape obligée en ce sens.

[32] VFS : Virtual FileSystem.

[33] POST : Power On Self Test : surtout un standard de dénomination de phases dans le boot sequence.

[34] Ramdisk : Issu du RAMFS, c'est un système de fichiers normal mais il est stocké en RAM. RAMFS est un cousin proche de TMPFS (TeMPorary FileSystem)

[35] Démon : programme tournant en « tâche de fond ». Il écoute la connexion de clients à un service déterminé. Il relaye l'offre de service d'un serveur à une connexion cliente.

c) Étude des solutions logicielles

LinuxBios s'est positionné comme solution presque unique de constructeur de BIOS libre. Il est lui-même composé de solutions de différentes natures qui, d'abord externes, se sont ensuite constituées en solutions Linuxbios propres au fil du temps, et ont été intégrées.

Pour construire un BIOS LinuxBios il faut LinuxBios bien sûr et il faut une solution de flashage afin d'écrire le BIOS sur une puce BIOS. Il faut une solutions de « logging » visant à récupérer et à mettre en forme les chaînes de déboggage. Il faut une solution de boot, un bootloader, et la « charge utile » à booter ou payload. Elle doit correspondre aux besoins actuels et prévus de Mangrove. Enfin, il faut une solution d'affichage, non implémenté complètement pour la carte mère EPIA en LinuxBios.

Solution principale : LinuxBios

- Généralités

Lorsque qu'il a été initié par le LANL, LinuxBios devait booter des noyaux Linux dans un environnement de clustering[36] de machines hétérogènes. Elles allaient chercher leur noyau sur un serveur. Sa vocation a été étendue à bien d'autres applications. LinuxBios a aussi comme vocation de combler des déficits, des lacunes et des erreurs des BIOS actuels qui rendent la maintenance difficile nécessitant l'intervention humaine. Par exemple, le « Press F1 » après une mise à jour est une tâche simple, anondine mais gigantesque à exécuter sur petit parc d'une centaine de machines. Ou bien qui aimerait attendre le test mémoire à chaque mise en fonction de son lecteur de poche MP3, ou le « Press F1 » sachant qu'il n'y a pas de clavier ?

Plus sérieusement, en dehors de ces incommodités et au-delà du fait que les sources de ces BIOS soient opaques, c'est-à-dire non OpenSource, ces BIOS maintiennent encore des fonctions qui ne sont plus utiles (support DOS) tandis que des fonctions vraiment requises ne sont pas développées (support ROM BIOS grande capacité, support mémoire vive grande capacité).

[36] Clustering : réseau de machines souvent hétérogènes dont les ressources et les tâches sont mises en communs et réparties suivant par exemple des modèles de traitements parallèles.

La structure de LinuxBios répond à ces attentes et, est inspirée par la volonté d'écrire le minimum de codes possibles afin de laisser le soin à Linux de faire le reste, ce qu'il fait depuis le début. Effectivement, LinuxBios se décompose en deux parties majeures : un bootstrap[37] et un noyau Linux . Pour avoir plus de détails sur le principe même de Linux BIOS, il est inconcevable de ne pas lire, des initiateurs du projet, les papiers « Proceedings of the 4th Annual Linux ShowCase & Conference Atlanta : Linux BIOS» et « Proceedings of the 4th Annual Linux ShowCase & Conference Atlanta : SEBOS» (cf. Références).

Afin de ne pas avoir de problème de construction dans l'une ou l'autre des parties et compiler LinuxBios sans encombre, voici les 'requièrements' qui vaut mieux respecter :

Composante	Version requise	Version courante
Gnu C	3.2.2	gcc --version
Gnu Make	3.70	make --version
Binutils	2.14.90	ld -v
CVS	1.11.6	cvs --version
Python	2.3	python -V

Pour construire LinuxBios, il faut trois éléments : un fichier makefile, un fichier de code assembleur crt0.S pour le mode réel, et un fichier des liens vers les fichiers C pour le mode protégé ldscript.ld.

Crt0.S met en place juste assez de code pour faire démarrer le bootstrap écrit en C. Ldscript.ld définit les adresses de réallocation de ce bootstrap pour qu'il puisse démarrer sans incident d'une ROM 256Ko ou d'une DoC 2000 8Mo.

Une des caractéristiques principales de LinuxBios, c'est que chaque carte mère possède ses informations de construction propres réparties dans les catégories composantes matérielles des fichiers sources. C'est l'utilisateur en définissant un script de configuration de haut niveau qui indique pour quelle carte mère, et par extension pour quelles composantes matérielles de cette carte mère, le BIOS doit être construit.

[37] Bootstrap : code amorce de boot.

Arbres des sources en src/

arch/	Crt0.S, ldscript.ld ainsi que les tables LinuxBios et pirqs pour i386.
boot/	Code des bootloaders : elfboot et filo.
config/	Le fichier de configuration indépendant de l'archi. Fichier Options
console/	Le code installant la console de base (text ou vga) et UART8250[38]
cpu/	Code de init. et réinit. des CPU, code MTRR, code cpuid.
devices/	Pour l'assignation des ressources PCI, la gestion PnP.
include/	**A**rborescence similaire pour les fichiers d'entêtes,de définitions.
lib/	**C**ode redéfini des allocations mémoires
mainboard/	Code d'initialisation des différentes carte mères supportées.
northbridge/	Initialisation du pont nord du chipset
southbridge/	Initialisation du pont sud du chipset
pc80/	Initialisation de composantes communes au 8086 (clavier, etc..)
sdram/	Code pour détecter la SDRAM
stream/	Code du contrôle du flux du bootloader

remarque : En gras sont représentés les répertoires classiques d'une arborescence OpenSource. Soulignés sont les répertoires qui revêtent un intérêt particulier pour le développeur LinuxBios.

Des fichiers de configuration Config.lb jalonnent l'arborescence. Ils ressemblent tantôt à des mini-makefile tantôt à des fichiers de pseudo code externes. Ils ont cette forme comme par exemple le fichier « fils » boot/config.lb :

```
object elfboot.o
object hardwaremain.o
if CONFIG_FS_STREAM
        object filo.o
end
```

- Configuration

Ces instructions et directives de compilation qu'on aperçoit ci-dessus, ne sont pas nécessairement directement comprises par Make mais elles le sont par buildtarget.

Buildtarget c'est un script shell exécutable, il se situe en target/. C'est lui qui paramètre et constitue le makefile, le crt0.S et le ldscript.ld afin que tout soit au point pour la compilation finale.

[38] UART8250 : Universal Asynchronous Receiver/Transmitter. Sert au protocole de communication RS232 pour la transmission sur port série ttyS0

Il se lance normalement avec deux paramètres:

```
[root@ge targets]# ./buildtarget
usage: buildtarget target [path-to-linuxbios]
```

Le premier est la cible ici via/epia, le deuxième est le chemin des sources s'il est facultatif il est très recommandé, il faut éviter de construire ses cibles dans les sources afin d'éviter toute corruption. Buildtarget appelle l'interpréteur python avec NLBConfig.py , c'est véritablement lui qui comprend le pseudo-code. Voici le schéma du déclenchement et des exécutions lors d'un appel à buildtarget :

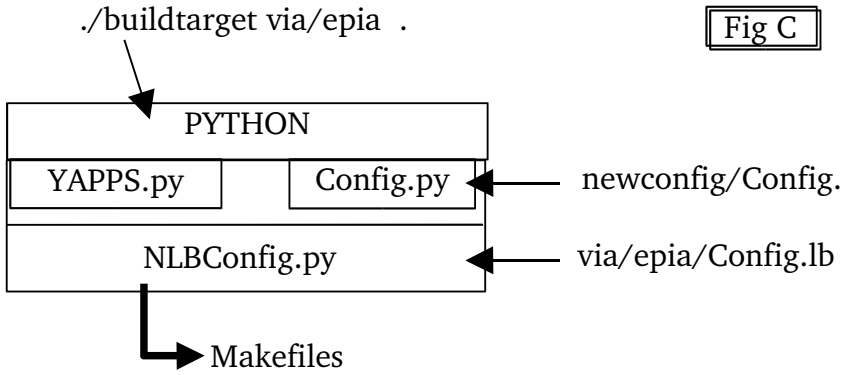


Fig C

NLBConfig.py parcourt le fichier config.lb et y analyse la synthaxe et les mots clés employés. Ces mots-clés sont définis dans un fichier d'inclusion Options.lb situé en src/config. Il définit les valeurs par défaut. Il ressemble à cela (tronqué):

```
        define LINUXBIOS_COMPILER
                default "$(shell $(CC) $(CFLAGS) -v 2>&1 | tail -n 1)"
                export always
                comment "Build compiler"
        end
[...]
        define ROM_IMAGE_SIZE
              default 65535
              format "0x%x"
              export always
              comment "Default image size"
        end
```

Le fichier Config.lb « père » apelle tous les fichiers « fils » config.lb qui ordonne la compilation des fichiers C unitaires au moyen de l'instruction object (comme le fichier « fils » boot/config.lb décrit juste auparavant). La liste des définitions et des commandes est disponible en annexe (cf. Annexe config.lb file commands), elle est issue du AMD64 Port Guide (cf. Références).

Voici un exemple(tronqué) d'un Config.lb principal ainsi que quelques repères :

```
# Sample config file for EPIA
# This will make a target directory of ./epia
#
# Change for experimentations MD
# ven avr 16 16:50:07 CEST 2004
#
loadoptions
target epia

uses ARCH
uses CONFIG_COMPRESS
uses CONFIG_IOAPIC
uses CONFIG_ROM_STREAM
[...]

# Comm settings
option DEBUG=1
option TTYS0_BAUD=115200
option CONFIG_CHIP_CONFIGURE=1
option MAXIMUM_CONSOLE_LOGLEVEL=8
option DEFAULT_CONSOLE_LOGLEVEL=8

# Graphics settings
option CONFIG_CONSOLE_SERIAL8250=1
option CONFIG_CONSOLE_VGA=0
option CONFIG_CONSOLE_BTEXT=1

# Misc options
option CONFIG_COMPRESS=1
option MAX_REBOOT_CNT=2
option CONFIG_SERIAL_POST=1
option HAVE_HARD_RESET=1
option AUTOBOOT_CMDLINE="hda1:/vmlinuz root=/dev/hda3 console=ttyS0 "
[...]
```

Nom du répertoire cible

Déclaration des variables obligatoires avant de les utiliser avec le mot-clef option

Activation du déboggage
Vitesse de transmission (Bps)

Niveau de verbosité du déboggage

Serial 8250 enverra le déboggage vers le port Série. Btext est le mode texte.

Active la prise en charge de Bios compressé
Nombre max. de reboot autorisé en cas
Affiche le déboggage du POST
Prise en charge du CPU reset
Root partition au Boot et paramétrage kernel

La différence entre les fichiers config.lb « fils » disséminés partout dans l'arborescence et le fichier Config.lb « père » se fait avec l'habitude sans trop de problème.

Mais d'une façon encore plus maladroite il me semble, un autre fichier en src/mainboard s'appelle Config.lb. S'il est vrai qu'il sert à configurer comme son congénère en src/target et qu'il utilise le même pseudo-code la ressemblance s'arrête là. Lui, définit le paramétrage de fonction inhérente à la carte mère, ces paramètres sont des défauts[39]. En général, si c'est une carte mère éprouvée, il n'y a guère de modification manuelle à faire ici. Des modifications erronées ici se payent assez cher en 'phase buildtarget'.

Il faut également particulièrement faire attention aux options qu'on utilise, certaines s'excluent mutuellement (dans ce cas buildtarget le signale) d'autres s'acceptent mais les résultats sont indéterminables (dans ce cas aucun garde-fou). Ce document abordera par la suite les différents bootloaders, et les outils de flashage qui sont incarnés par des options. Ils sont parfois inclus en Config.lb quand ce sont des solutions internes ou des solutions externes souvent sollicitées. Plus il y a d'options plus il y a d'erreur possibles au rang desquelles celles de manipulations ne sont pas les moins pénibles.

Pour toutes ces raisons, j'ai créé LBCC (LinuxBios Control Center) un outil en bash portable et robuste qui, typiquement, appelle buildtarget, qui fabrique une LinuxBios ROM, qui récupère un fichier journal. Il unifie toutes les grandes fonctionnalités suivantes:

- ✔ Configure LinuxBios
- ✔ Fait un journal de son activité
- ✔ Calcule le temps de construction
- ✔ Affiche les réglages de la ROM
- ✔ Choisit un payload
- ✔ Configure un payload
- ✔ Construit le payload
- ✔ Construit la ROM
- ✔ Flashe la ROM

Le code source de ce programme est disponible en annexe (cf. Annexe Lbcc).

[39] Paramètre défaut : réglage d'une valeur initiale stable, pouvant être redéfinie.

Maintenant, que buildtarget a rempli son office, sur l'écran doit apparaître en toute fin :

```
Build ROM size 262144
Verifying ROMIMAGE fallback
Verifying ROMIMAGE normal
Verifing global options
Creating via/epia/epia/fallback/static.c
Creating via/epia/epia/fallback/Makefile.settings
Creating via/epia/epia/fallback/crt0_includes.h
Creating via/epia/epia/fallback/Makefile
Creating via/epia/epia/fallback/ldoptions
Creating via/epia/epia/normal/static.c
Creating via/epia/epia/normal/Makefile.settings
Creating via/epia/epia/normal/crt0_includes.h
Creating via/epia/epia/normal/Makefile
Creating via/epia/epia/normal/ldoptions
Creating via/epia/epia/Makefile.settings
Creating via/epia/epia/Makefile
```

C'est assez clair, cette commande a créé les makefiles fallback, normal et le Makefile générique comme le suggère la fig C. Les makefiles sont prêts, il ne reste plus qu'à exécuter le make. Avant de lancer ce processus relativement long, il peut être utile de vérifier les options. Celles-ci sont reportées dans le fichier Makefile.settings. Ci-dessus, on reconnaît également le fichier d'inclusion crt0 et le fichier ldoption. Ce dernier présente peut-être plus simplement encore, toutes les options de compilation et de linkage.

- La compilation

Typiquement, aprés la configuration c'est la compilation qui suit. Il suffit de se positionner dans le répertoire de la cible configurée c'est-à-dire target/<vendor>/<product>/<target_dir> et d'exécuter un make. Il va en fait exécuter deux makes : dans fallback et normal, les deux sous-répertoires de build. Une autre façon d'afficher les options de compilation et linkage consiste encore à faire un make echo.

Il possible de constater qu'à l'intérieur du makefile il y a beaucoup de cibles parmi lesquelles les plus intéressantes sont :

linuxbios.rom , echo , build_opt_tbl, linuxbios, linuxbios.strip, romcc, build rom, clean.

remarque : buildrom, romcc, build_opt_table sont compilés dans un appel à un compilateur représenté par la variable $HOSTCC. Tandis que les autres programmes, le compilateur appelé est $CC. Ceci pour prendre en compte si la machine qui compile LinuxBios n'est pas la machine cible, c'est-à-dire la machine sensée l'accueillir par la suite. Sinon, les compilateurs sont les mêmes et les valeurs des deux variables devraient être 'gcc'.

Ces cibles s'auto-référencent, et référencent des objets ça et là. Un éclaircissement s'impose, voici le schéma général de la compilation et des dépendances entre les cibles :

```
All
 +Linuxbios.rom
  +Linuxbios.strip
      +Linuxbios
            +Crt0.o
            +Initobjects
            +Linuxbios_payload
            +ldscript.ld
  +Buildrom
```

Les Initobjects sont tous des fichiers sources .c de contenu si l'on peut dire. A l'inverse de crt0.o et de ldscript.ld qui sont le squelette de l'application.

Les paramétrages de précompilation se font principalement pour les Initobjects. Concrètement, dans le code des directives dites de précompilation, sont présentes sous forme de `#define` et `#ifdef` les options qui ont été choisies. Ces paramétrages correspondent pour gcc à des options -D listées dans ldscript.ld. Il n'y a point de commande -D sur les lignes de commande gcc des fichiers qui font ce squelette.

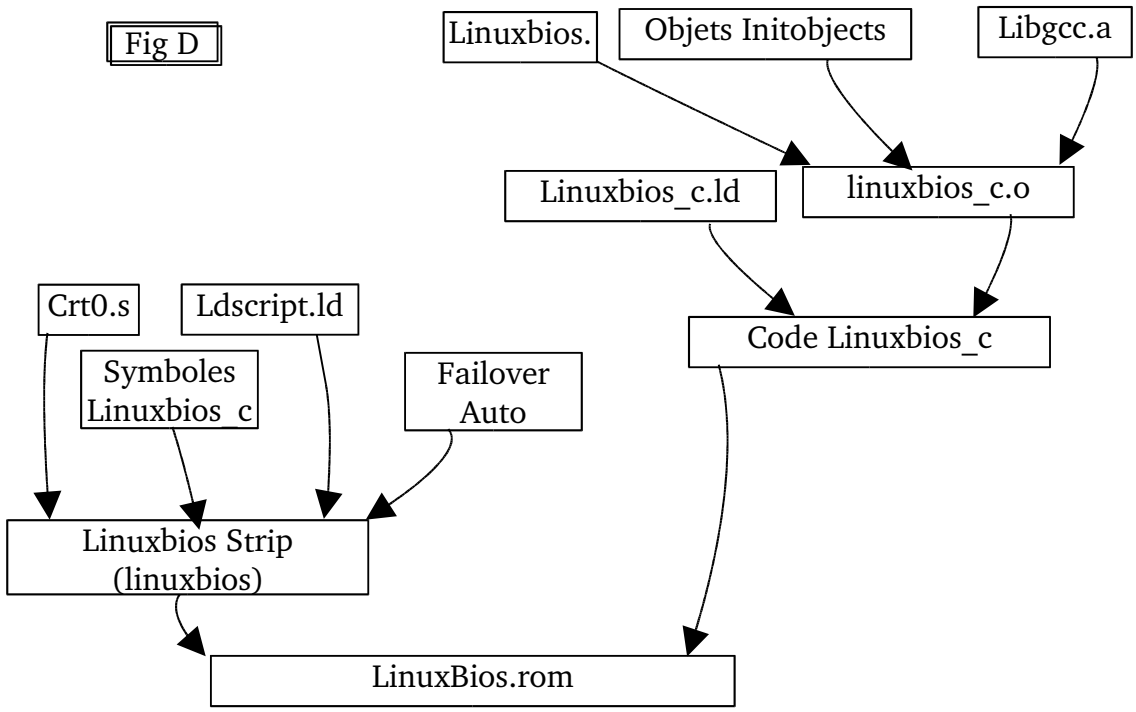
Voici les étapes de la compilation, les actions en détail :

1. le fichier de version LinuxBios est compilé.
2. le fichier romcc est compilé.
3. Crt0.S est assemblé, failover.c et auto.c sont compilés avec romcc.
4. Les objets Initobjects sont compilés.
5. Les références InitObjects sont récupérées en linuxbios.a.
6. L'objet linuxbios_c est compilé avec les objets InitObjects, linuxbios.a et la libgcc.a.
7. Le code exécutable linuxbios_c est généré avec linuxbios_c.ld et

linuxbios_c.o.

8. La liste des symboles de ce fichier est récupéré en linuxbios_c.map.
9. Le code amorce linuxbios est généré avec ldscript.ld et crt0.o.
10. Les objets de linuxbios sont recopiés en linuxbios_payload.
11. rv2b est compilé, appliqué à linuxbios_payload (compression) le nouveau fichier est renommé en linuxbios.rom.
12. Le code amorce linuxbios est généré avec ldscript.ld et crt0.o.
13. La liste des symboles de linuxbios est stocké en linuxbios.map.
14. Les objets de linuxbios sont recopiés en linuxbios.strip.
15. Le payload est compilé s'il existe.
16. Buildrom est compilé et appelé avec linuxbios.strip, linuxbios.rom et payload.rom.

Diagramme de la construction des cibles



Si la compilation de la ROM LinuxBios n'a pas renvoyé d'erreur, un large volume de messages reste affiché à l'écran et, si on a utilisé LBCC, ce volume a été redirigé également dans des fichiers logs. La ROM est disponible dans target/<vendor>/ <product>/<target_dir>/ et elle s'appelle linuxbios.rom. C'est ce code binaire, adapté au processeur tel qu'il est paramétré dans mainboard/Config.lb, qui va être flasher dans une EEPROM.

GCC a compilé beaucoup de fichiers sources pour produire cette ROM mais certains fichiers ont été compilés au moyen d'un autre programme : ROMCC. Avant d'aborder le flashage et les solutions de flashage, il est important d'expliquer pourquoi ces fichiers requièrent une compilation particulière et d'aborder les mécanismes avancés de LinuxBios.

- Les mécanismes avancés de LinuxBios

Une autre caractéristique importante de Linuxbios est qu'il est entendu qu'il doit tourner sur une machine en tant que programme standalone[40]. Il faut donc qu'il n'ait aucune référence externe en dehors du groupement de fichiers LinuxBios vu précédemment. C'est pourquoi on retrouve le code du memset (mappage mémoire) dans src/lib: il est impossible dans le contexte de LinuxBios d'utiliser les sources memset du noyau qui présupposent un service bios encore non mis en place.

Je ne résiste pas à exposer ici la simplicité de ce code qui sert tant pour le BIOS, notamment pour le shadowing ou caching BIOS. A quoi d'autre pourrait-on s'attendre ?

```
void *memset(void *s, int c, size_t n)
{
        int i;
        char *ss = (char *) s;

        for (i = 0; i < n; i++)
                ss[i] = c;

        return s;
}
```

Pourtant ce petit code ne pourrait être appelé dans le mode réel. Écrit en C, il requière une pile et un tas (STACK & HEAP) qui ne sont définis que pendant le mode protégé. Quoiqu'il en soit, il n'a vocation qu'a être appelé à cet instant, et c'est donc bien gcc qui compile ce code. Mais par exemple, le code de calcul de la RAM doit être appelée en mode réel (pas de pile et de tas) alors gcc ne peut pas compiler ce code, même s'il est écrit en C.

Romcc est un compilateur qui génère, à partir du C, du byte code spécifique pour ROM. Cela signifie que le code n'a besoin d'aucune RAM pour fonctionner. Romcc a été apporté par V2, avant (avec V1) tous les codes du mode réel devaient être en assembleur.

[40] Standalone (programme) : programme n'ayant aucun autre programme père. Il est isolé et doit pouvoir fonctionner dans un environnement vide de références à d'autres codes.

Deux gros avantages du C sont sa lecture plus simple par rapport à l'assembleur et sa grande portabilité : le code de l'initialisation RAM est le même pour toutes les cartes mères. Romcc émule des piles avec les registres CPU et fait des permutations : le code exécutable a tendance à prendre un peu de volume mais le jeu en vaut la chandelle. Le temps de boot ne s'en trouverait pas affecté significativement.

Au moment du boot, une autre fonctionnalité avancée entre en jeu. Linuxbios crée des tables appelées LinuxBios tables. Elles résident en mémoire basse et contiennent des informations sur la carte mère utilisée, l'adresse du shadow BIOS (dans une partie non volatile de la RAM) où les paramètres LinuxBios sont stockés. Le kernel préserve naturellement cette table afin qu'elle soit pas altérée aprés la fin du boot, dans l'OS. Dans le source tree, l'utilitaire lxbios sert à récupérer et lire ces tables afin de vérifier la validité d'un checksum sous l'OS par exemple.

LinuxBios peut également utiliser cette partie non volatile de la mémoire pour stocker de l'information persistante. Ces infos tels le compteur du nombre de boot(boot counter: CNT_BOOT) sont définies dans une structure appelée CMOS table. Toujours visible depuis l'OS, elle contient entre autres les infos suivantes :

```
#startbit  length   config  ID         name
#-------------------------------------------------
0          384       r       0        reserved_memory
384          1       e       4        boot_option
385          1       e       4        last_boot
386          1       e       1        ECC_memory
388          4       r       0        reboot_bits
392          3       e       5        baud_rate
400          1       e       1        power_on_after_fail
412          4       e       6        debug_level
416          4       e       7        boot_first
420          4       e       7        boot_second
424          4       e       7        boot_third
428          4       h       0        boot_index
432          8       h       0        boot_countdown
1008         16      h       0        check_sum
```

Ce fichier est stocké dans les sources de mainboard. Les utilitaires cmos_xxx sont disponibles dans le source tree également. Ils permettent de faire des modifications du CMOS en temps réel sous OS. (cf Annexe CMOS tables)

Amélioré dans V2, les mécanismes de la gestion chaînée des devices est un autre point fort à mon sens de Linuxbios. Les chips et les devices sont transparentes. Surtout, il s'agit d'une aisance d'appel aux fonctions de ces structure. Sur ce même modèle, la structure des devices PCI elle aussi à été modifiée (par rapport à celle du Kernel ou de V1 qui sont identiques). Tout le squelette des composantes matérielles est dévoilé dans ces structures.

La structure chip (voir static.c)

```
 struct chip_control northbridge_via_vt8601_control = {
.enumerate = enumerate,
.enable    = northbridge_init,
.name      = "VIA vt8601 Northbridge",
};
```

La stucture device (voir device.h) et les pointeurs de device (device_t)

```
struct device {
        struc_bus *         bus;
        device_t  sibling;
        device_t            next;
        struct device_path          path;
        unsigned short      vendor;
        unsigned short      device;
        unsigned int        class;
        unsigned int        hdr_type;
        unsigned int        enable :1;
        uint8_t             command;
        struct resource     resources [MAX_RESSOURCES];
        unsigned int        ressources;
        struct bus          link[MAX_LINKS];
        unsigned int        links;
        unsigned long       rom_adress;
        struct device_operation          *ops;
        struct chip         *chip;
}
```

A elles seules ces structures suggèrent l'organisation chaînée en LinuxBios : elle est dite PCI-centrique (PCI-centric).

Enfin l'une des fonctionnalités les plus séduisantes consiste à détacher une partie du payload sur le réseau. Effectivement, LinuxBios permet d'excentrer, le kernel par exemple, sur un serveur DHCP ou TFTP. Dans le cas d'utilisation de clustering, la distribution de noyau semble particulièrement intéressante. On peut imaginer également des dispositifs de mise à jour distants sans aucune fiction. Plus de détails sont disponibles dans AMD64 Port Guide (cf. References). Les possibilités d'architecture concernant les payload sont décrites plus en avant dans la section 'solution de boot' de cette partie.

Si LinuxBios a bien compilé c'est qu'il a été correctement paramétré, et c'est une affaire assez complexe. Mais, le paramétrage de la ROM dont je n'ai pas parlé, est resté par défaut notamment la taille de la ROM est définie par défaut à 256Ko. Il sera peut-être nécessaire de la modifier, tout depend du composant que l'on a choisi.

Solution de flashage

Le flashage qui consiste à écrire la ROM linuxBios.rom sur un composant BIOS reste un point central. Si aucune solution de flashage n'avait été trouvée, une mise en oeuvre de LinuxBios aurait été inutile. Comme les solutions de flashage dépendent des composants, ceux-ci sont au départ de cette problématique. Virtuellement, il est possible d'écrire et de programmer n'importe quel composant électronique à mémoire. Là où l'interdépendance frisse l'interblocage c'est que, la solution de base, LinuxBios pourra concrètement reconnaître que certains type de composants.

- les composants : les EEPROMS ou « flashs memory »

Les EEPROMS pouvant accueillir un BIOS de type LinuxBios sont relativement nombreuses. Mais, les différents types sont eux restreints. Un composant est relié à un bus avec son protocole. Grosso modo, d'après les structures exposées ci-avant, cette relation désigne directement un périphérique. Linuxbios n'est capable de lire un BIOS que sur le bus interne du CPU - RAM.

Il y a beaucoup de puces sur le marché de taille, de format, de type différents. Cela pourrait être très compliqué si des standards ne venaient pas normaliser tout cela. Le standard JEDEC est un groupe de normes sur le boîtiers EEPROM et sur le jeu d'instruction. Et LinuxBios est compatible JEDEC.

En somme, il est donc possible d'utiliser ces types d'EEPROM:

- Puce bios commune 128Ko à 512Ko

Pour mes expérimentations, j'ai utilisé des puces SST39SF020A, SST29EE020, WinBond W49F002UP12B, Winbond W29C020CP90B, AMD AM29F040B. Elles n'ont pas toutes les mêmes tailles par exemple soit 256Ko ou 512Ko. Il est recommandé de se reporter au datasheet[41] des constructeurs qui décrivent le boîtier de ces mémoires.

[41] Datasheet : Documents techniques issus des constructeurs décrivant les spécifications d'un matériel.

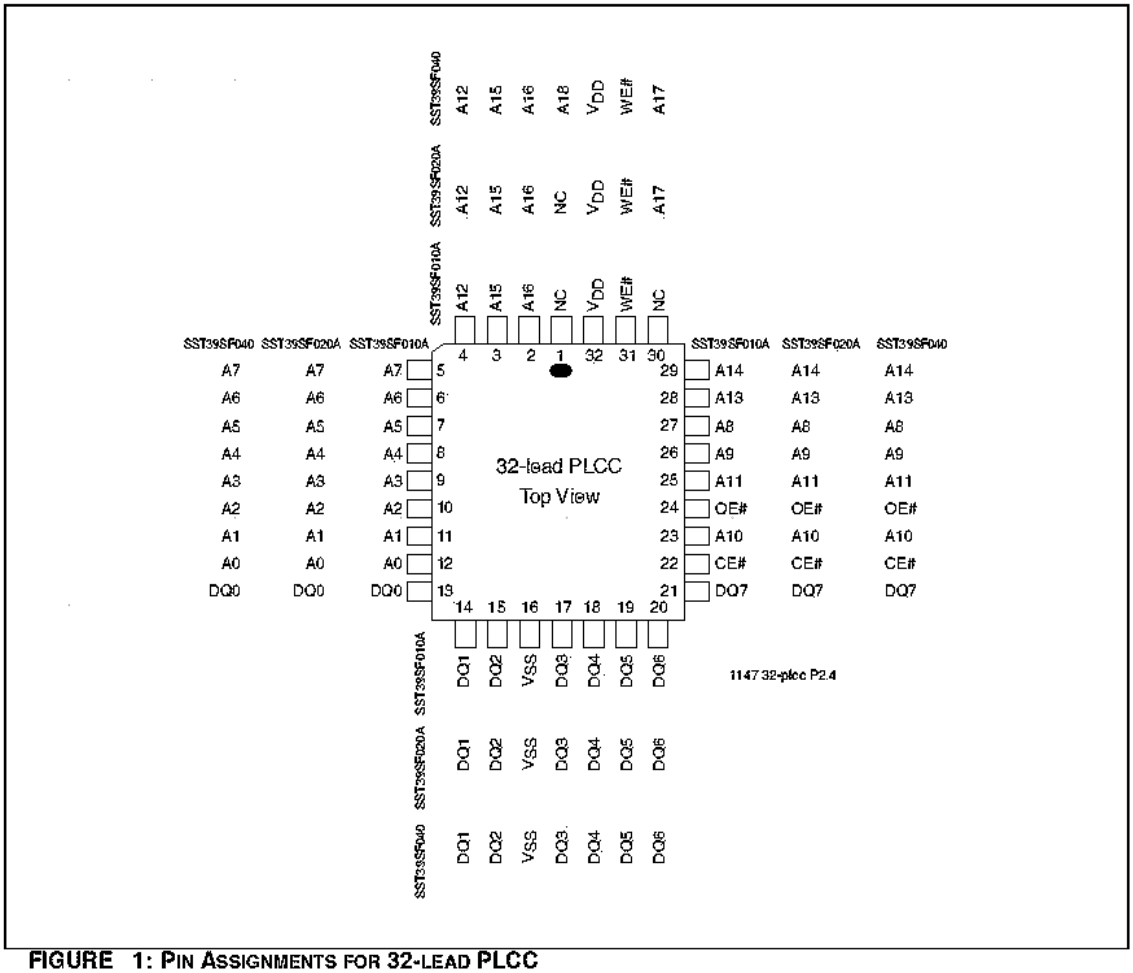


FIGURE 1: PIN ASSIGNMENTS FOR 32-LEAD PLCC

Ci-dessus le boîtier au format PLCC issu du datasheet SST de la puce bios SST39SF020A. Les formats les plus communs de ces type de puces sont PDIP, PLCC et TSOP.

- Disk On Chip (DoC) 1Mo – 8Mo –16Mo – 32 Mo – 64Mo – ..

C"est un dispositif de type MTD (Memory Technology Device). Une mémoire flash qui peut être accédée et contenir un système de fichier comme un disque dur.

Mangrove est intéressé par cette solution de stockage car pouvant remplir un de ses objectifs : regrouper OS et BIOS en une seule mémoire qui est assez grande pour accueillir les deux. Ce gain améliorerait peut-être encore le temps de boot. Même si j'ai décrit auparavant comment configurer une DoC dans le Kernel, je n'ai pas eu le temps de pousser plus loin dans Linuxbios notamment, mes recherches avec ce dispositif.

- Différence « flashage à chaud, à froid ».

J'ai tout d'abord cherché coûte que coûte à réaliser cette opération de flashage, qui était décrite comme délicate, à froid. Mais aucune solution de ce type n'ayant fonctionné, le flashage retenu est à chaud. L'appréhension à propos de ce type de flashage a été grandement exagérée, avec une pince adaptée au format de la puce et un mininum d'habileté, ce n'est pas plus dangereux que de débrancher un câble parallèle de son port. Un instrument qu'il est recommandé de posséder est un BIOS Savior, une sorte de 'T' pour mettre deux bios, l'original et le LinuxBios, dans la même prise sur la carte mère. Changer de bios sans avoir à le démonter avec une pince c'est tout l'intérêt de ce dispositif. En fait, il permet de changer de position un jumper/interrupteur et ainsi de changer de ROM.

Le flashage à chaud, cela consiste à réitérer la puce BIOS originale de la carte mère sous tension et y placer celle que l'on veut flasher. Puis on la flashe. Et on redémarre pour voir si cela boot, ou déjà si on obtient les chaînes de déboggage.

A froid, cela signifie que la carte n'est plus sous tension lors de l'opération de retrait/insertion. Généralement, cela requière un programmeur d'EEPROM.

- Matériel et Logiciel

L'une de mes premières tâches de flashage consista à faire une copie de sauvegarde du seul BIOS que j'avais à ma disposition et de travailler après sur cette copie.

Lors de mes expérimentations de flashage à froid, j'en suis venu à utiliser un programmeur Willem EEPROM Programmer v3.1 20-03-2002, un outil « amateur » consistant en une plaque epoxy avec un emplacement au format DIP où mettre l'EEPROM et des switchs et jumpers. La lecture des deux feuillets de documentations, si j'ose les appeler ainsi, et le bricolage d'un adaptateur PLCC vers DIP m'aura fait perdre autant de temps que l'installation d'un OS Microsoft pour faire fonctionner le pilote. Si j'ai réussi à lire à peu près le contenu d'une puce, l'écriture fut impossible. Voici le protocole de flashage (à froid donc) que j'ai suivi:

1. Grâce au pilote lire le bios original sur SST39SF320A (256Ko) et le recopier en AM29F040 ou en STT29EE020.
2. Arrêter le système
3. Interchanger les puces
4. Tester

Je me suis rapidement tourné vers des types de solutions logicielles. DevBios d'OpenBios permet d'écrire mais, qu'à chaud. Il voit sous Linux le bios comme un device et on peut écrire dessus comme sur une console ou un disque dur ce qui est élégant. Mais il fallait oublier le flashage à froid pour utiliser devbios.

Le problème est que devbios ne lit pas les instructions du processeur C3 de l'EPIA. A l'écriture devbios fait des erreurs de segmentation mémoire. De plus, il ne supporte que peu de puces.

Comme il semblait clair qu'il fallait oublier également DevBios et que je ne pouvais flasher qu'à chaud, le plus simple était d'utiliser l'outil natif de flashage de LinuxBios : flash_rom. Cet outil ne m'a posé aucun problème et gère de nombreuses puces. L'appel sans argument de flash_rom propose les options de son utilisation.

- Structure d'une ROM LinuxBios

Pour terminer cette section, voici une présentation de la disposition des plages mémoire dans une ROM LinuxBios. Les adresses basses sont en haut.

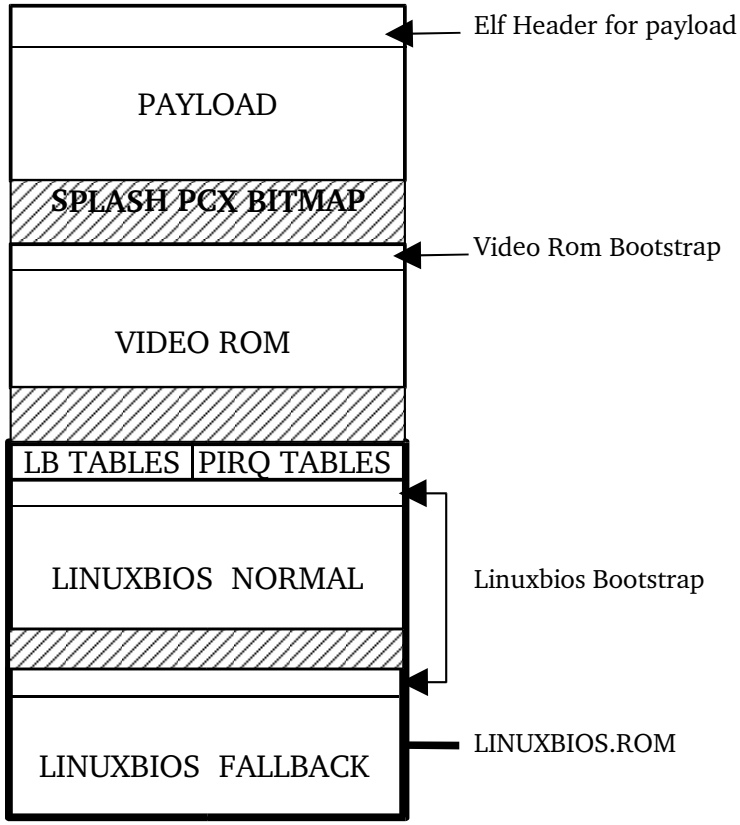


Fig E

Solution de logging

Une ROM LinuxBios qui compile et qui est flashé avec succès ne doit pas laisser préjuger qu'elle fonctionnera correctement. Or tant que le mode VGA ou BTEXT n'a pas été mis en place, la seule solution pour le vérifier c'est le dispositif tel qu'il a été présenté dans la partie Moyens,Outils et Ressources.

Si l'option SERIAL_CONSOLE a été correctement paramétrée ainsi que la vitesse de transmission, normalement des messages provenant du BIOS doivent être envoyés sur le port série ttyS0.

Pour les intercepter, j'ai étudié tout d'abord minicom. C'est un programme qui émet et reçoit des trames sur les ports séries pour les modems par exemple. Après plusieurs tentatives de paramétrage infructueuses, minicom est apparu difficile à manipuler : commandes en combinaison de touches avec CTRL, des barres de menu en mode text qui se rafraîchissent mal, etc.. En fait, il fait beaucoup plus que ce qui est recherché : récupérer les chaînes sur ttyS0 et les afficher à l'écran.

Dans ce cas, j'ai décidé d'utiliser un script minimal e bash de ma création assurant ces tâches. (cf Annexe communication scripts)

```
[root@ge utils]# ./hear.sh
Usage ./hear.sh log_to_file
```

La manipulation est très simple :

Tout d'abord, il faut adopter l'installation comme l'indique la fig A. La machine LinuxBios mérite désormais amplement son nom, car c'est la puce bios contenant la ROM LinuxBios qui s'y trouve en place.

Sur l'autre machine, se placer dans un terminal et lancer un stty. Si la vitesse de transmission ne correspond pas à celle paramétrée dans le fichier Config.lb, alors il faudra lancer par exemple un `stty ispeed 115200`. Il ne reste plus qu'à lancer Hear.sh, il restera en attente en écoute.

Enfin, démarrer la machine Linuxbios, et Hear.sh enregistrera toutes les chaînes dans le fichier qui lui aura été spécifié.

Voici un le début typique extrait d'un fichier journal crée au moyen de Hear.sh

```
Hear! Boot Log started on 14/05/04|15:48:02
tty in use : /dev/pts/3  from ttyS0
************************************************
64      LinuxBIOS-1.1.6.0Fallback Fri May 14 15:25:51 CEST 2004 starting...$
65      87 is the comm register$
66      SMBus controller enabled$
67      vt8601 init starting$
68      00000000 is the north$
69      1106 0601$
70      0120d4 is the computed timing$
71      NOP$
72      PRECHARGE$
73      DUMMY READS$
74      CBR$
75      MRS$
76      NORMAL$
77      set ref. rate$
78      enable multi-page open$
79      Slot 00 is SDRAM 04000000 bytes $
80      0080 is the chip size$
81      0008 is the MA type$
82      Slot 01 is SDRAM 04000000 bytes $
83      0040 is the chip size$
84      0008 is the MA type$
85      Slot 02smbus_error: 04$
86      Device Error$
87       is empty$
88      Slot 03smbus_error: 04$
89      Device Error$
90       is empty$
91      vt8601 done$
92      00:06 11 01 06 46 00 90 a2 05 00 00 06 00 40 00 00 $
93      10:08 00 00 e0 00 00 00 00 00 00 00 00 00 00 00 00 $
94      20:00 00 00 00 00 00 00 00 00 00 00 00 06 11 10 60 $
95      30:00 00 00 00 a0 00 00 00 00 00 00 00 00 00 00 00 $
96      40:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 $
97      50:fe df c8 98 00 00 10 10 88 00 08 08 10 10 10 10 $
98      60:3f aa 0a 30 e4 e4 e4 c4 42 ac 65 0d 08 7f 00 00 $
99      70:c0 88 6c 0c 0e 81 52 00 01 f4 01 00 00 00 00 00 $
100     80:0f 45 00 00 00 00 00 00 03 00 70 07 00 00 00 00 $
101     90:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 $
102     a0:02 00 20 00 07 02 00 07 00 00 00 00 6e 02 00 00 $
103     b0:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 $
104     c0:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 $
105     d0:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 $
106     e0:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 $
107     f0:00 00 00 00 00 00 01 01 22 42 00 b0 00 10 00 00 $
108     Disabling cache$
109     Clearing mtrr$
110     Setting XIP$
111     Enabled the cache$
```

```
112     Copying LinuxBIOS to ram.$
113     Jumping to LinuxBIOS.$
```

Il y a deux moyens de terminer la connexion : classique, en envoyant une fin de fichier (par exemple : /dev/null) ou interactif en appuyant sur CTRL+C.

La fin de l'écoute, Hear.sh propose d'ajouter un commentaire, mieux vaut y placer les conditions d'expérimentations, les options spécifiques testées, des impressions... La fin du journal contient ces commentaires ainsi que les temps d'exécution.

Voici la fin extraite du même fichier journal.

```
269     found PCI IDE controller 1106:0571 prog_if=0x8f$
270     primary channel: native PCI mode$
271     Waiting for ide0 to become ready for reset... ok$
272     Testing for hda$
273     Probing for hda$
274     LBA mode, sectors=40021632$
275     Init device params... ok$
276     hda: LBA 20GB: Maxtor 32049U3                          $
277     Partition 1 start 63 length 2040192$
278     Unknown filesystem type$
279     Command terminated by signal 2
real 13.67
user 0.01
sys 0.01
---------------------------------------------
Comments :

This is a BTEXT_CONFIG test
 no payload, no videoROM, no VGABIOS
from a cold boot : no  screen "video mode on" (led goes green)
from a warm boot : the screen flashes but nothing written
```

Quelques précisions/commentaires avant de refermer cette partie :

Le texte 'real 13.67' signifiant 13.67 secondes temps réel au total comprend plusieurs boot de suite.

L'affichage de chaîne de déboggage ralentit le temps de boot considérablement. La preuve, si elle en est une, est le texte 'user 0.01' qui indique 1/100éme de seconde en temps utilisateur. L'augmentation de la vitesse de transmission permet de réduire ce ralentissement.

Le 'unknown filesystem' indique que mon payload, elfboot, n'accepte pas le type de filesystem qui se trouve sur hda, partition 1.

Solution de boot

Cette section va approfondir un concept important de LinuxBios, le payload ou charge utile. Bien que représentée (fig E) en haut c'est-à-dire stockée dans les adresses faibles, elle n'est lue qu'en fin de l'exécution de Linuxbios, comme le suggère la fin du fichier journal en page précédente. Dans cet exemple, c'est le payload ELFBOOT qui avait été construit avec l'ordre de booter sur hda, partition 1, etc..

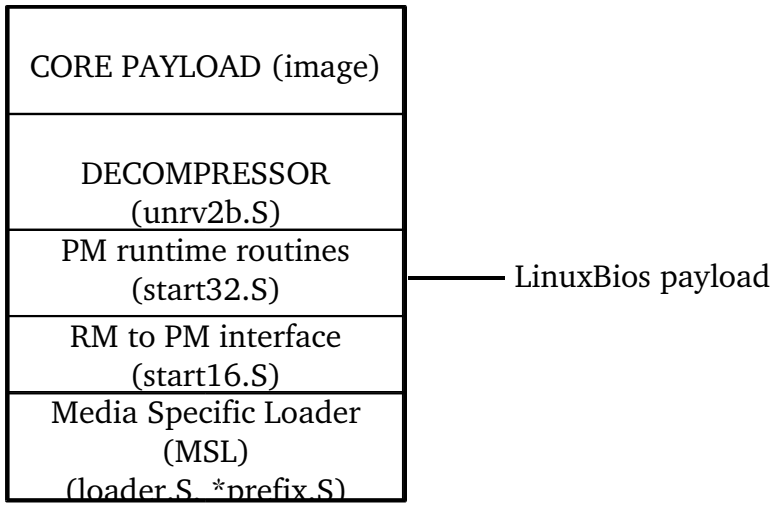
Il faut faire attention à ce qu'on entend par boot, c'est une notion un peu confondue. Dans cet exemple avec FILO, la solution de boot ou booter est véritablement un bootloader (FILO) car son payload est un noyau. Le bootloader FILO a comme payload un noyau Linux, dans ce cas FILO est utilisé comme un chargeur de système d'exploitation. Il est tout à fait envisageable de faire booter un programme qui ne sert pas à charger un OS. Dans ce cas le booter ne sera pas bootloader : il ne sera que booter.

C'est d'ailleurs ce qu'il est recommandé de faire au début, un payload peut être, en tant que test, un petit programme rigolo qui affiche « hello tragic world ! ». Il y a une relative liberté avec les payloads : soit un bootloader charge un noyau soit le bootloader ou justement dans ce cas le booter, boot un autre booter ou bootloader.

La seule limitation concernant les programmes qui servent de payload est qu'il faut s'assurer qu'il peut se positionner dans un espace mémoire, libre et contigu. Il faut également que ce programme soit lié statiquement (sans dépendance externe ou standalone).

Voici la structure interne d'un payload :

Fig F



Si l'on veut Lilo comme bootloader, le payload consistera en un MSL de type liloprefix.S (cf source tree ADLO), puis les codes assembleur des starts. Ensuite, le décompresseur unrv2b agissant sur le noyau image du payload (cf Fig F : 'core payload') compressé avec rv2b. Et enfin ce noyau image lui-même.

Faire un payload « à la main » prend du temps, autant utiliser des solutions toutes faites. Il y a deux catégories de solutions de boot : les boots natifs LinuxBios et les boots externes. Le but recherché par la direction du projet LinuxBios, est faire en sorte que toutes solutions de boot soient externes au source tree Linuxbios de façon à rendre l'implémentation et la maintenance facile.

Le format du payload standard est ELF, les kernels sont de ce format. MkelfImage est un programme qui permet de fabriquer des images bootables à partir de n'importe quel fichier, il s'utilise ainsi :

```
mkelfImage --command-line="root=/dev/hda3 console=ttyS0,115200" \
        --kernel vmlinux
```

Il existe deux comportements de bootloader. Soit il est simple et il détecte une image elf dans les premiers 8Ko, ou alors plus évolué et il analyse le système de fichier à la recherche d'une image.

A noter par conséquent, que les images doivent être dans le début de la ROM pour les bootloaders simples.

- LinuxBios : elfboot (simple)

C'est la solution de base. Elle peut booter sur IDE un payload de type elf, typiquement il s'agit d'un noyau Linux au format elf (cf. Étude de Linux et du bios). Cela peut aussi être un de ces petits programmes rigolos dans ce cas il devront être convertis, au besoin, au format elf.

- (LinuxBios :) filo (évolué)

C'est une solution qui a été internalisée rapidement. Les informations sur filo que j'ai pu obtenir concernerait plutôt filo en externe. Aussi, il m'est impossible d'affirmer que telle fonction appartient à filo interne ou externe. Filo externe est utilisé à la manière d'elfboot mais développe plus de capacités.

Il peut :

- booter sur IDE : Disques Durs et CDROMs
- booter des ROM standard (bus mémoire systeme – CPU)
- booter depuis un périphérique « brut » avec un offset spécifique tel la Compact Flash (CF) ou DiskonChip (DoC)
- supporter les systèmes de fichiers suivants : ext2,fat,jffs,reiserfs,xfs and iso9660
- supporter des payload au format : ELF, zImage, bzImage (vmlinux)
- supporte rla console VGA, le clavier, le port série

Cette solution est entièrement écrite en code 32 bits (en C) sans appel à LinuxBios. C'est la solution que j'ai choisi d'utiliser : elle semble être particulièrement testée. Basée en partie sur elfboot, je crois qu'elle a vocation à le remplacer.

- USB boot (simple ?)

C'est une solution tout à fait récente. Elle ajoute un support de boot sur USB au BIOS LinuxBios. Elle aurait les mêmes capacités qu'elfboot portées pour l'USB. Plus d'informations sont disponibles sur la maillist de LinuxBios.

- Etherboot (simple et existence de quelques versions évoluées)

Etherboot est un projet éprouvé. Il permet de booter un payload depuis le réseau. Il construit une image selon le type de carte réseau ou chip réseau dont la carte mère dispose. En fait, il faut indiquer à LinuxBios de charger cette image. D'autre part, un serveur FTP est prêt à répondre à une requête de téléchargement de noyau.

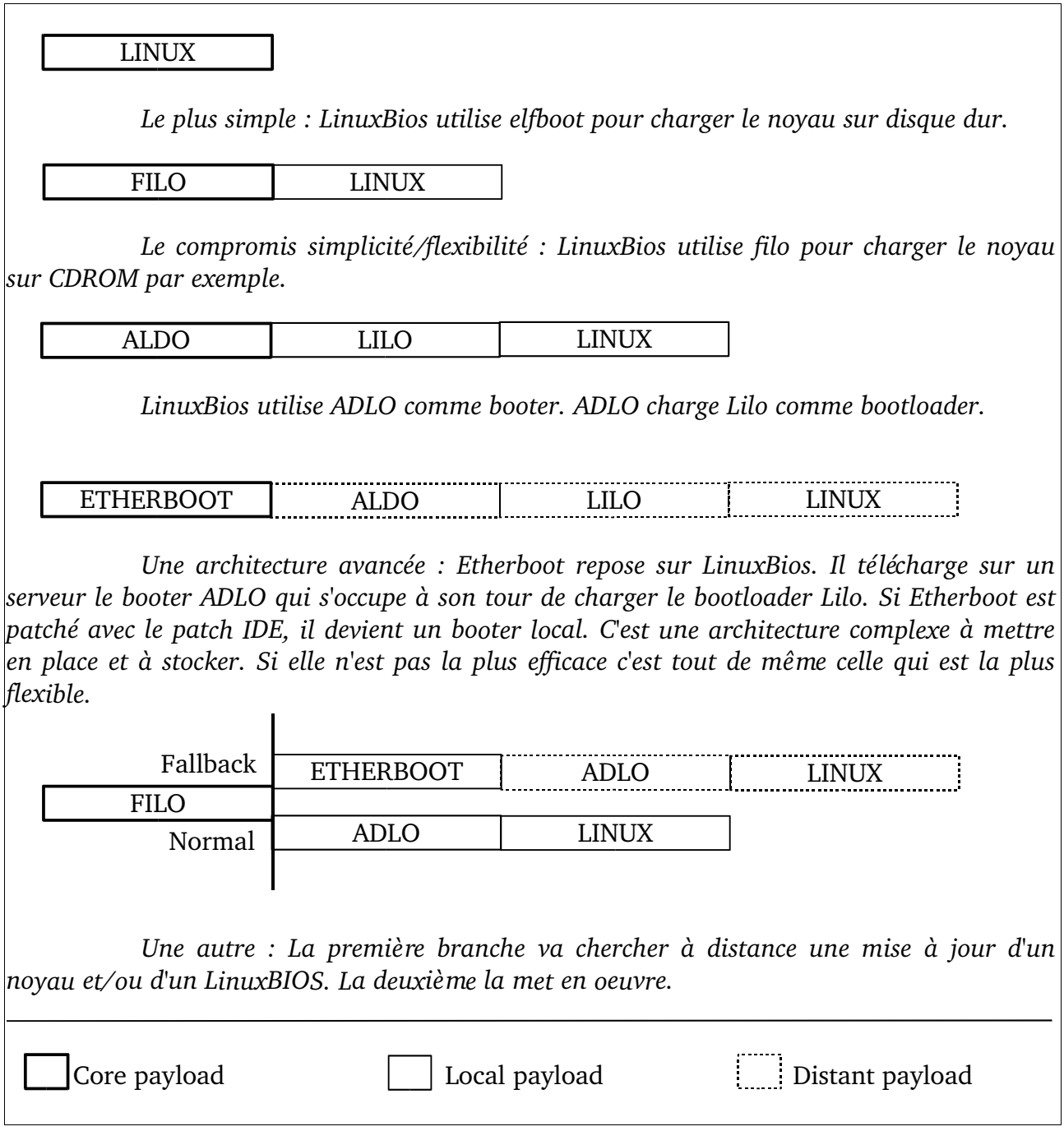
Il est possible de faire booter Etherboot sur IDE également grâce à un patch. L'intérêt par rapport à un bootloader de type elfboot est de conserver la possibilité de booter via le réseau.

- BOCHS/ADLO

Cette solution est également très intéressante et a été très utilisée, plus qu'elle l'est désormais. ADLO est basé sur BOCHS, et propose de faire un payload sur mesure pour ainsi dire. Il propose encore une implémentation de VGABIOS. Les sources sont disponibles depuis le source tree en V1.

Enfin, cette partie ne saurait être complète sans indiquer qu'il est possible de combiner ces solutions afin de faire une chaîne de boot (boot-chain) selon les capacités de chacune d'entre elles. Comme LinuxBios est la solution de base, ces chaînes implicitement l'implémentent au début.

Exemple d'architecture de chaîne de boot



Le plus simple : LinuxBios utilise elfboot pour charger le noyau sur disque dur.

Le compromis simplicité/flexibilité : LinuxBios utilise filo pour charger le noyau sur CDROM par exemple.

LinuxBios utilise ADLO comme booter. ADLO charge Lilo comme bootloader.

Une architecture avancée : Etherboot repose sur LinuxBios. Il télécharge sur un serveur le booter ADLO qui s'occupe à son tour de charger le bootloader Lilo. Si Etherboot est patché avec le patch IDE, il devient un booter local. C'est une architecture complexe à mettre en place et à stocker. Si elle n'est pas la plus efficace c'est tout de même celle qui est la plus flexible.

Une autre : La première branche va chercher à distance une mise à jour d'un noyau et/ou d'un LinuxBIOS. La deuxième la met en oeuvre.

Dans ces 'payloads chains' ou 'boot chains', jamais le payload en cours de fonction ne rendra la main au précédent. En d'autres termes, 'aucun n'a de retour d'exécution : ceci signifie qu'il est possible de récupérer de la place en rom. Ceci dit, il faut conserver des « traces » du travail qui à été réalisé par les payloads successifs. Il est question, que ces informations soient accessible depuis la LinuxBios Table.

Les solutions de boot sont, au sein de LinuxBios, les composantes qui sont les plus travaillées. En fluctuation constante, elles représentent l'intérêt premier des utilisateurs, le boot, et doivent pouvoir être modulable à souhait. Selon l'architecture de la chaîne de boot, le composant BIOS choisi varie lui aussi. Pour débuter, mieux vaut choisir des chaînes de boot les plus simples et éprouvées. La mise en oeuvre et une proposition d'architecture sera décrite dans dernière partie (cf. mise en oeuvre de LinuxBios)

Solution d'affichage

Les travaux d'expérimentation n'exigent pas vraiment d'affichage graphique pour faire fonctionner LinuxBios et constater son fonctionnement. Cela n'est pas possible dans l'optique de développement produit, d'ailleurs la machine LinuxBios a bien vocation à être autonome. Les développeurs LinuxBios ne s'y sont pas trompés, et différents modes pour la console sont disponibles.

Lors de son chargement, Linux initialise et met en place la console graphique et pilote l'affichage VGA[42] avec un grand soin. Partant de cet état de fait, le rôle de LinuxBios est d'initialiser les registres graphiques et de mettre la carte graphique ou le chipset graphique en l'état 'prêt à afficher'. Cela ne représente pas un code important mais sa réalisation requière une connaissance pointue de la structure de ce composant ainsi que les valeurs d'initialisation.

Il n'y a qu'une seule façon d'aborder le problème.

Ces informations qui sont « délicates » à obtenir, empêchent la conception d'un code vidéo. Mais il y a quelque chose de disponible, c'est la video ROM original du BIOS original ou VGABIOS. Les codes d'initialisation sont contenus dans cette ROM (Fig E).

[42] VGA : Video Graphic Array.

Difficulté supplémentaire, ces initialisations reposent souvent sur des appels BIOS en mode réel 16 bit alors que LinuxBios est totalement (ou presque) en 32 bit. Il est possible de remarquer au boot d'une machine que la première chose qui apparaît à l'écran (et c'est bien normal) c'est le nom de la carte graphique, du constructeur ainsi que la mémoire dont elle dispose et cela avant même le calcul de la RAM disponible (cf. Boot Sequence)

Il n'y a qu'une solution : utiliser un émulateur en dehors de LinuxBios, x86emu. Un émulateur ce n'est pas la panacée mais il ne sera pas permanent : il pourra être remplacé par vesa (vesafb) ou mieux framebuffer (fbdev) ou encore mieux X11. Ces dispositifs nécessitent juste que le périphérique soit assez fonctionnel pour qu'ils le redéfinissent.

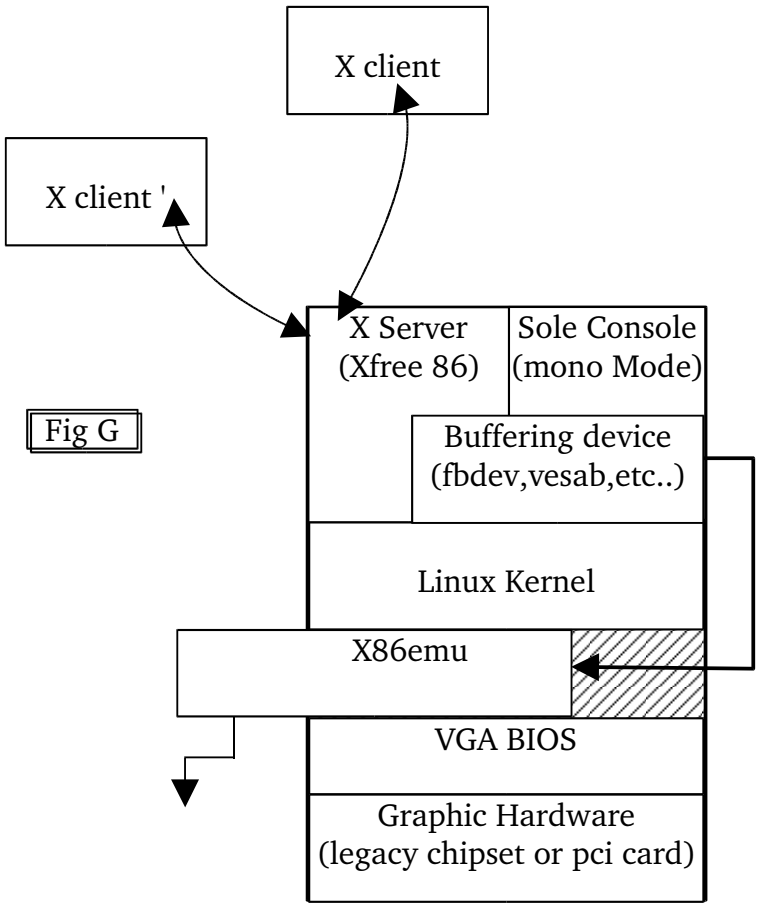


Fig G

Epilement logiciel pour la gestion graphique avec LinuxBios

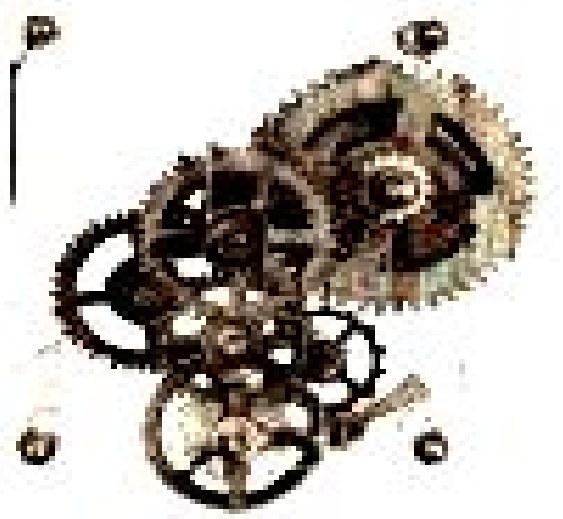
La configuration et l'utilisation du VGA n'empêchent pas la réception des chaînes de déboggage sur le port série. En général, il est bienvenu de désactiver le SERIAL DEBUG , les mêmes informations étant disponibles localement.

Le source tree de Linuxbios met à la disposition un émulateur de test testbios pour vérifier la validité sous Linux, d'une video ROM.

Pour conclure cette section et, en général, cette étude de solutions, il a été mis en évidence que Mangrove LinuxBios peut répondre à l'ensemble des besoins qu'un système d'information actuel requière. Ces travaux sur LinuxBios et sur les solutions autour de LinuxBios permettent d'implémenter un Bios très orienté vers Linux, vers la répartition et la distribution d'OS en réseau, et en général vers l'acquisition de ressources d'exploitation très spécifiées pour des systèmes embarqués.

La mise en oeuvre suivante détaille les opérations qu'il faut réaliser afin d'implémenter un LinuxBios pour les systèmes embarqués Mangrove. Je n'ai pas été plus loin dans l'étude pure pour respecter cet objectif tout aussi important.

MISE EN OEUVRE



L'objectif d'une mise en oeuvre est de faire booter un noyau avec un BIOS LinuxBios le plus abouti possible. Afin d'atteindre ce niveau d'élaboration sont déployés les solutions LinuxBios. Cette partie a pour vocation première d'aider les futurs développeurs à mettre en oeuvre les solutions LinuxBios rapidement. Il condense grandement les directions suivies dans l'étude et fait office de résumé aussi.

Dans l'étude, des résultats sont prévus, le comportement du système estimé. Dans la mise en oeuvre, il est observé et mesuré. La vocation seconde de cette partie est d'exprimer ce décalage, et de donner les limites du développement actuel.

Enfin, sa dernière vocation est de proposer des pistes futures de recherche et de développement afin de continuer à réaliser le projet Mangrove LinuxBios.

Tout d'abord, il est primordial de réunir les documentations sur la carte mère cible. Avant de travailler sur EPIA M 1000, j'ai commencé les travaux avec une carte mère à base de chipset SiS530/5513 (North) et 85C503 (South). Il m'a été impossible de trouver une amorce en tout pour ce SiS530 alors il a fallu changé. Pour récupérer des infos sur la carte mère, un `lspci -vv` est suffisamment détaillé, il faut penser à préciser cela quand on poste sur la maillist.

Les how-to et les faq peuvent être gardés à proximité car utiles constamment. J'ai utilisé comme plate forme de développement ou toolchain, la Red Hat 9.0 car regroupant tous les composants et outils plate forme dont LinuxBios a besoin. C'est plus rapide et facile à installer et à réinstaller (« au cas où... ») sur une machine que de mettre à jour chaque partie de la plate forme. Il faut mieux préférer le partitionnement en ext2, le support ext3 n'est pas encore géré.

Ensuite, la préparation d'un noyau avec les supports à (re)définir. Les besoins en fonctionnalités doivent être bien connus. Je n'ai pas travaillé cet aspect si ce n'est pour le support filesystems et DoC car les noyaux prévus pour Mangrove LinuxBios sont des noyaux spécifiques LBT/LBA. Ils ont une grande flexibilité et peuvent être adaptables autant que Mangrove LinuxBios. Dès lors, il est possible d'installer l'appareil tel qu'il est décrit en Fig A. Au sujet de la vitesse de transmission du port série, la commande à utiliser est `stty / dev/ttyS0 ispeed <vitesse>.`

Après avoir disposé le source tree (Mangrove) LinuxBios dans le système, lancer un `lbcc via/epia -a` . Lbcc va d'abord, proposer de configurer votre LinuxBios (Config.lb). Il est parfois utile de s'inspirer des Config.lb des autres configurations même si elles sont différentes.

Pour les options de ce fichier, chacune d'entre elles doivent être déclarées (uses) puis utilisées (options).

Quelques explications sur les options a activer :

DEBUG=1

Active le mode déboggage.

TTYS0_BAUD= <vitesse > (124800,76800,48000,28800,19200,9600,1200)

Vitesse de transmission sur le câble. Plus elle est grande plus le boot sera rapide si debug est activé.

LOGLEVEL =<number> dft: 6

Les niveaux sont décrits en arch./i386/include/printk.h. Écrire un message à un niveau info est réalisé grâce à printk_info. Voici la numérotation qui correspond au niveaux d'importance du message.

1 : EMERG	*4 : ERR*	*7 : INFO*
2 : ALERT	*5 : WARNING*	*8 : DEBUG*
3 : CRIT	*6 : NOTICE*	*9 : SPEW*
		10: tous

MAX_REBOOT_CNT = <number> dft: 3

Nombre de tentative de boot maximale.

CONFIG_SERIAL_POST =1

Active les valeurs POST indiquant le niveau d'avancement du boot sequence.

CONFIG_COMPRESS = 1

Signale que l'image en ROM est compressée.

HAVE_HARD_RESET = 1

Gestion reset à chaud.

I686 = 1 et i586 = 1

Des réglages registres spécifiques pour ces architectures.

CPU_FIXUP = 1

Nécessaire dans certains cas (lequels ?),patch pour le cpu.

INTEL_PPRO_MTRR = 1

Registres pour gestion graphique.

HAVE_OPTION_TABLE = 1

Partie de la LinuxBios Table.

HAVE_FALLBACK_MODE = 1

Disposer un mécanisme de secours de boot en cas d'erreur.

ROM_SIZE = <number>*1024 dft: 256

Taille du composant BIOS à voter disposition.

FALLBACK_SIZE = <number>*1024 dft: 192

Taille de la partie Fallback dans la ROM LinuxBIOS.

_ROMBASE = 0x00004000
Départ du code C LinuxBIOS en ROMBASE.
CONFIG_IDE = 1
Active le boot sur bus IDE.
CONFIG_FS_STREAM =1
Ouvre un flux sur un système de fichier. Active le booter/bootloade LinuxBios Native FILO (ceux supporter par FILO).
CONFIG_ROM_STREAM = 0
Ouvre un flux sur une mémoire à structure linéaire.
AUTOBOOT_CMDLINE = « hda1:/vmlinuz root=/dev/hda3 console=tty0, ttyS0 »
Ligne de commande de boot du bootloader et passage de paramètres au noyau Linux. Si aucun flux n'est ouvert, c'est ELFBOOT qui fera booter ce noyau.
CONFIG_LEGACY_VGABIOS = 1
Active l'émulation x86emu pour le VGABIOS.
VGABIOS_START=0xfffc0000
Pointeur sur le BIOS VGA en LinuxBios.
CONFIG_CONSOLE_SERIAL8250 = 1
Relaye les chaînes de déboggage de la console au port série.
CONFIG_CONSOLE_BTEXT = 1
Active une console texte unique (cf. Fig G 'Sole console')
CONFIG_CONSOLE_VGA = 1
Active une console graphique pouvant supporter X11.

Structure `romimage`
romimage 'fallback'
Titre de section : fallback ou normal.
USE_FALLBACK_IMAGE = 1
Est la rom de secours en cas d'incidents.
ROM_IMAGE_SIZE=0x10000
Taille de la rom de secours .
mainboard via/epia
carte mère qu'elle est destinée à booter.
payload null
charge utile à booter. Null indique qu'il s'agit d'un payload issu d'une solution interne : c'est soit Linuxbios native ELFBOOT ou soit Linuxbios native FILO. C'est une image au format elf.
##payload_type=filo
sert à Lbcc. Indication fonctionnelle. Double dièse afin de ne pas être pris en compte dans buildtarget.
end

Après avoir sauvegardé cette configuration et quitté vi, Lbcc, appelant builtarget, entre dans la phase ROM Building. Cela correspond exactement avec ce que fait buildtarget.

Voici les messages tels qu'à l'écran durant la phase de ROM building

```
 Build Directory found as                via/epia/epia
 Normal Payload Directory found as       /dev/null (type : filo)
 Fallback Payload Directory found as     /dev/null (type : filo)
 PROCEED TO INIT PROCESS  IN 5 SECONDS...
 Build target log   : /
home/root/projet/freebios/freebios2/targets/via/epia/make.log
 Rom making log     : /
home/root/projet/freebios/freebios2/targets/via/epia/epia/make.log
 Payload making log : <none>
 Normal process stated...
 Prepare logging facility
 Editing via/epia/Config.lb  ...
 PROCEED TO BUILD TARGET IN 3 SECONDS...

[...]

Build ROM size 262144
Verifying ROMIMAGE fallback
Verifying ROMIMAGE normal
Verifing global options
Creating via/epia/epia/fallback/static.c
Creating via/epia/epia/fallback/Makefile.settings
Creating via/epia/epia/fallback/crt0_includes.h
Creating via/epia/epia/fallback/Makefile
Creating via/epia/epia/fallback/ldoptions
Creating via/epia/epia/normal/static.c
Creating via/epia/epia/normal/Makefile.settings
Creating via/epia/epia/normal/crt0_includes.h
Creating via/epia/epia/normal/Makefile
Creating via/epia/epia/normal/ldoptions
Creating via/epia/epia/Makefile.settings
Creating via/epia/epia/Makefile

real    0m9.605s
user    0m8.950s
sys     0m0.120
10 buildtarget warnings
```

Lbcc entre ensuite dans la phase ROM Binding, en fait cette phase n'est, pour l'instant, qu'un affichage des valeurs des adresses mémoires.

```
PROCEED TO ROM BINDING IN 10 SECONDS...
----------------------------------------
Normal rom sizing value :
export ROM_IMAGE_SIZE:=0x10000
export PAYLOAD_SIZE:=0x0
export _ROMBASE:=0xfffc0000
export CONFIG_ROM_STREAM_START:=0xfffc0000
export ROM_SIZE:=0x40000
export FALLBACK_SIZE:=0x30000
export ROM_SECTION_SIZE:=0x10000
export XIP_ROM_SIZE:=0x10000
export XIP_ROM_BASE:=0xfffc0000
----------------------------------------
fallback rom sizing value :
export ROM_IMAGE_SIZE:=0x10000
export PAYLOAD_SIZE:=0x20000
export _ROMBASE:=0xffff0000
export CONFIG_ROM_STREAM_START:=0xfffd0000
export ROM_SIZE:=0x40000
export FALLBACK_SIZE:=0x30000
export ROM_SECTION_SIZE:=0x30000
export XIP_ROM_SIZE:=0x10000
export XIP_ROM_BASE:=0xffff0000
----------------------------------------
PROCEED TO *ROM MAKING* IN 10 SECONDS...
```

La rom LinuxBios est alors compilée et on obtient un fichier LinuxBios.rom en via/epia/epia si on peut constater aucune erreur comme ici :

```
real    0m49.490s
user    0m37.870s
sys     0m1.810s
0 warnings, avertissements,no such file, not found
------------------------------------------------
1 only of theses seems to be relevant :
44:0 warnings, avertissements,no such file, not found
done.
```

Lbcc est dans l'optique de compiler n'importe quel payload issu d'une solution externe. Dans ce cas présent, le payload c'est filo, une solution interne, par conséquent tout s'est trouvé compilé dans la phase précédente.

Enfin, Lbcc appelant flash_rom flashe, à chaud, le bios dans la puce qui a été disposée à la place de l'ancienne. S'il advenait qu'il faille changer de solution de flashage il faudrait une autre étape entre la précédente et celle-ci pour construire cet outil externe.

```
Starting flashing rom : /
home/root/projet/freebios/freebios2/targets/via/epia/epia/linuxbio
s.rom...
Place target bios rom onboard.
!!!  WARNING WARNING !!! ALL IN CHIP DATA WILL BE ERASED !
Continue (y/n) ? y

Erasing chip... Stand by for 40 seconds approx...
Calibrating timer since microsleep sucks ... takes a second
Setting up microsecond timing loop
159M loops per second
OK, calibrated, now do the deed
Enabling flash write on VT8231...OK
W49F002U found at physical address: 0xfffc0000
Part is W49F002U
Programming Page: address: 0x0003f000

real    0m10.613s
user    0m10.160s
sys     0m0.020s

Programming chip... Stand by for 40 seconds approx...
Calibrating timer since microsleep sucks ... takes a second
Setting up microsecond timing loop
144M loops per second
OK, calibrated, now do the deed
Enabling flash write on VT8231...OK
Trying W49F002U, 256 KB
probe_jedec: id1 0xda, id2 0xb
W49F002U found at physical address: 0xfffc0000
Part is W49F002U
Programming Page: address: 0x0003f000

real    0m10.613s
user    0m10.160s
sys     0m0.020s
All right.done.
```

Voilà pour la mise en oeuvre globale, c'est tout. Lbcc se charge de tout faire

pourvu que Config.lb soit correctement configuré. Toutefois, il reste l'affichage vidéo si les options ont été activées dans Config.lb. En fait en l'état, Lbcc ne le fait pas encore automatiquement. Il faut donc le faire « manuellement », mais cela a été tellement travaillé que cela est très abordable.

Pour mettre en oeuvre l'affichage vidéo, il faut tout d'abord récupérer la video ROM du bios original comme l'étude l'a expliqué. Elle fait au maximum 64Ko (soit 65535 octets). A titre indicatif, celle de de l'EPIA fait plus exactement 44Ko. Cela dépend des constructeurs, cette taille varie.

On suppose que la video ROM commence en 0xC0000, souvent c'est le cas (option VGABIOS_START) et se terminera en 0xD0000. Mais on sait qu'il faut faire un décalage de 0x01000 pour commencer en 0xC1000 , il s'agit d'une entête elf.

Kcore est un périphérique particulier qui contient toute la mémoire vive donc le noyau en activité, qui fluctue en temps réel. Par défaut dans la RH9[43], il a été mis en place au format elf grâce à une option du noyau. A ce sujet le noyau qui tournera sur la LinuxBIOS machine pourra être au format elf afin de faciliter cette opération.

Ce video Bios est récupérable dans Kcore mais il est inutile de prendre l'entête elf qui sera reconstruite par la suite, alors lui aussi est évité. Donc, le code exécutable pur du bios video se situe en 0xC1000 -- 0xD1000 (soit du 790528^{éme} octet au 856064^{éme}).

Cette commande va récupérer ce code en kcore :

```
dd if=/proc/kcore of=/video.bin bs=1\
              count=65536 skip=790528
```

Le fichier obtenu video.bin peut être testé avec testbios dans V1/util/vgabios/testbios. Pour ce faire, il faut installer le module du pilote graphique de la carte ou du chipset, puis démarrer testbios.

```
insmod fbdev.o
./testbios -s 65536 video.bin
```

[43] RH9 : Red Hat 9

J'ai conçu quelques scripts de tests et de régression graphique qui peuvent être lancés un moment pour vérifier la stabilité (cf. Annexe les scripts graphiques)

Si cela fonctionne correctement, il ne reste plus qu'à intégrer cette ROM dans l'empilement BIOS (Cf. Fig E) soit dans le fichier linuxbios.rom à la sortie de la phase du 'rom making'. Ceci peut être fait en lançant `lbcc via/epia -m` puis après que l'intégration soit faite dans linuxbios.rom, il 'y a plus qu'à faire `lbcc via/epia -f`.

Ces travaux sont en l'état tels qu'il m'est possible de dire, qu'il y aura plus de limitations et de correctifs à venir qu'il y en a actuellement. Quand le V2 est sorti, les développements de types patchs et codes d'interfaces ont du être intégrés à ce qui ne s'appelait que Linuxbios (en fait V1). De telle sorte qu'aucun patch particulier ne soit nécessaire pour l'EPIA si on travaille sur V2, et donc assez peu de limitations en ce sens.

En revanche, certains fonctionnalités, options, fonctions, ne sont pas encore portées de V1 vers V2 , pas complétement. Ceci requiert du temps de test et de lecture, autant que voire plus que, de temps de codage. J'ai fait des patchs dont il faut faire mention ici au cas où ils seraient compromettants et qu'il faille revenir en arrière (rollback).

Voici le seul portage auquel je n'ai trouvé aucune autre parade (que son existence):

Pour faire fonctionner mdelay (erreur type : undefined reference from linuxbios_c.o) , il faut activer CONFIG_UDELAY_TSC. Cela requière les définitions de rdtsc (low,high) et rdtscll(val) issu de V2/cpu/p6/msr.h vers V2/cpu/p5/delay_tsc.c.

J'ai travaillé par snapshot et maintenu un source tree Mangrove, ce n'est pas forcément mauvais. Mais même si les patchs ne furent pas nombreux, il me semble que cela va changer. Alors un CVS devient indispensable pour gérer les versions sans confusion. Je n'ai pas utilisé cette outil car je le trouve pour ainsi dire, assez compliqué à manier, surtout pour des besoins élémentaires. Toutefois, ces derniers devenant plus nombreux et surtout plus complexes, les tests et les versions se multiplient. Ceci tandis qu'il faut produire un logiciel Mangrove pour clients, partenaires, ceux-ci n'ayant pas à subir de désagréments issus d'une recherche plus dense.

Pour parer à cela, il est indispensable d'utiliser CVS avec deux branches de développement différentes afin de distinguer :

Une branche de recherche (BR), expérimentale et;
Une branche de développement (BD), des produits.

La BR génère de l'instable mais est virtuellement illimitée, elle prépare les fonctionnalités aux besoins des clients de préférence en devançant un peu leurs demandes. La BD génère une stabilité mais subit des limitations, elle répond à la demande et dispose des 'produits' aux 'commandes' des clients en temps réel.

A l'heure actuelle, le portage VGA est presque terminé, en fait VGABIOS s'appuie sur la structure pci_dev de V1, tandis que V2 s'appuie sur la structure device (cf. mécanisme avancée de Linuxbios). Normalement, je pourrais tester très bientôt l'affichage vidéo. Il restera à l'intégrer au Mangrove LinuxBios source tree.

Pour ce qui est du noyau, pour l'instant les bootloaders ne supportent pas ext3, je n'ai pas eu le temps de tester avec mon dernier noyau sur ext2, mais je pense qu'il devrait être booté sans encombre d'après les messages de débogagges des bootlogs. Ce noyau possède ce qui requis pour LinuxBios, il très stable pour l'EPIA. Il a tourné plusieurs semaines durant sous un BIOS constructeur de sorte qu'avec un BIOS Linuxbios, il ait toutes ces chances.

Dans l'optique de développements futurs, à court terme (6 mois à 12 mois max.) voici ce qu'il faudrait poursuivre et pourquoi :

- BR-CT-1)

Intégrer l'installation de la video ROM en Lbcc afin d'automatiser le procédé. Le gain de temps permettra des tests de régression avec EPIA fbdev et d'autres plus poussés en BTEXT puis VGA mode. Et enfin, sous X.

- BD-CT-2)

Stocker un noyau sur Compact Flash, paramétrer LinuxBios pour le faire booter. Faire des tests de régression avec de boot et de lecture/écriture sur le média. Faire de même avec une Disk On Chip et réitérer. Mettre ces produits à disposition des clients et partenaires.

- BR-CT-3)

Mettre sur pied une architecture avancée avec Etherboot pour booter sur réseau. A dessus de ce développement, orienter Mangrove LinuxBios afin qu'il opère des mises à jour du BIOS et de l'OS à distance par téléchargement VSFTP.

- BR-CT-5)

Construire un outil qui stocke des images dans la ROM pour créer de bootsplash personnalisable. L'intégrer à Lbcc. Ceci devrait pouvoir répondre aux besoins de personnalisation naissants émergeant un moment après la disposition de BD-CT-2).

Enfin à plus moyen terme, rechercher à mettre en place des outils de sécurité et d'authentification sera un enjeu. Ceci afin de garder la maîtrise d'un concept naissant, les flux (I/O) des BIOS ! Ce concept incarné par le contrôle d'accès de lecture (chargement BIOS au boot), d'écriture (mise à jour BIOS sous OS), de parcours par le réseau (visualisation d'un plan réseau, partage de ressources) va t-il prendre du service à une granularité inférieure à celle où il avait l'habitude de fonctionner ?

Quels peuvent/vont être les défis de sécurité qui devront être relevés en cas de déploiement à plus grande échelle ? Comment va se disposer le marché de l'embarqué face à ce type de technologies ? Comment s'assurer que l'esprit d'ouverture qui permet leur naissance perdure face aux dispositifs techniques et légaux déployés de plus en plus en tant que verrous ?

CONCLUSION

Cette étude dans son ensemble est parvenue à décrire le projet LinuxBios, et à spécifier un projet fils Mangrove LinuxBios.

A l'issue de cette synthèse, les réalisations que je laisse à Mangrove sont une base solide, avec un potentiel et un degré d'élaboration à développer encore d'avantage, afin de passer peut-être d'un produit expérimental à un produit commercial. Enfin dans la mise en oeuvre, mes travaux ont mis les objectifs de Mangrove LinuxBios restants désormais en vue, à portée.

Pour ma part, ces travaux m'ont véritablement passionné, j'ai appris à bien des niveaux. J'ai essayé de faire en sorte de retourner mon expérience ainsi que l'esprit de ces travaux, j'espère y être parvenu. Enfin, j'espère que cette contribution au travers des programmes et de ce document pourra aider.

Peut-être celle-ci donnera le goût à d'autre personnes de travailler sur ce projet avec la communauté du logiciel libre ? Ce n'est pas toujours simple, loin de là mais travailler ainsi c'est encore, mais aussi, plus que développer des technologies.

REFERENCES

Theses are the main references :

- **Ieee Standard Organisation**, n° 1275-1994 *http://www.ieee.org*
- **Freebios & LinuxBios**, *the seed, rich FAQ and mailist http://www.linuxbios.org, http://www.freebios.org*
- **Spécifications VIA EPIA,** *via arena http://www.via-arena.com*
- **Etherboot project**, great work at *http://www.etherboot.org/*
- **Openbios project,** great work at *http://www.openbios.org/*
- **LOBOS project**, Maryland Information Systems Security Lab (MISSL) « *LinuxBIOS: A Study of BIOS Services as used by Microsoft Windows XP. » http://www.missl.cs.umd.edu*
- **The Linux Documentation Project**, Various works ad helpful infos. sources *http://www.tldp.org*
- **The Kernel,** *http://www.kernel.org*
- **Les secrets du BIOS PC/AT,** in deep at *http://www.qsl.net*
- LinuxBios Maillist posts from other contibutors in which, in no specific order :
  http://www.lnxi.com *http://www.linuxlabs.com*
  http://www.tyan.com *http://www.sis.com*
  http://www.cdy.com *http://www.onelabs.com*

And not to forget books/mags/papers too :

- MINNICH (R) with USENIX « Proceedings of the 4th Annual Linux ShowCase & Conferece Atlanta : LOBOS» , August 14th,2000
  http://www.linuxbios.org/papers
- MINNICH (R), HENDRICKS (J), WEBSTE (D), with USENIX « Proceedings of the 4th Annual Linux ShowCase & Conferece Atlanta : The Linux BIOS» , August 15th ,2000
  http://www.linuxbios.org/papers
- REINAUER (S) « LinuxBios on AMD64 Port Guide », February 10th,2004
  http://www.openbios.org
- « Linux embarqué » , LinuxFrance Hurd Magazine n°36 Février 2002.
- « Kenel 2.4.0 », LinuxFrance Hurd Magazine n°25 Février 2001.
- « Linux système embarqué » , LinuxFrance Hurd Magazine n°17 Mai 2000.
- «Voyage au centre au noyau » , Linux Magazine HS n°17 Novembre,Décembre 2003.

ANNEXES

Récapitulatif des schémas

Quelques repéres rapides dans la maillist LinuxBios

Pipermail Recoding Date	Subject title or Topics
2002-October, 2002-November	ADLO
2003-April	Xfree, IDE BOOT, CF BOOT
2002-December	VGA Splash screen
2003-September	Command line boot options
2004-February	Use of compact flash

Préparer un payload elf

Voici comment préparer un payload elf avec MkelfImage :

```
mkelfImage --kernel=/usr/src/linux-2.4/vmlinux \
            --output=vmlinuz.target\
            --command-line=''console=ttyS0,115200n8 \
                      root=/dev/hda3''
```

A la place du noyau dans l'option kernel n'importe quel file Image compressé avec gzip fait l'affaire. Si on lance la commande `file vmlinuz.target`, la réponse mentionnera 'not stripped'. Pour dégarnir (strip) l'image des sections de déboggage, de commentaire, qui font la font grossir, lancer ces comamndes :

```
strip file.elf
objcopy -R .note -R .comment file.elf
```

Le fichier file.elf.stripped qui peut être renommé sans problème en file.elf contient maintenant ce qui est strictement necessaire pour fonctionner. Il est possible de vérifier que le fichier ne contient aucune section de note ou de commentaire ainsi :

```
objdump file.elf -t
```

Pour qu'il soit booté, il doit être mis en /dev/hda3 c'est la partition root /. Il faut le copier sur la partition hda3. On cherche les premiers 4ko, voici la commande à lancer :

```
dd if=file.elf of=/dev/hda3 bs=4096 seek=1
```

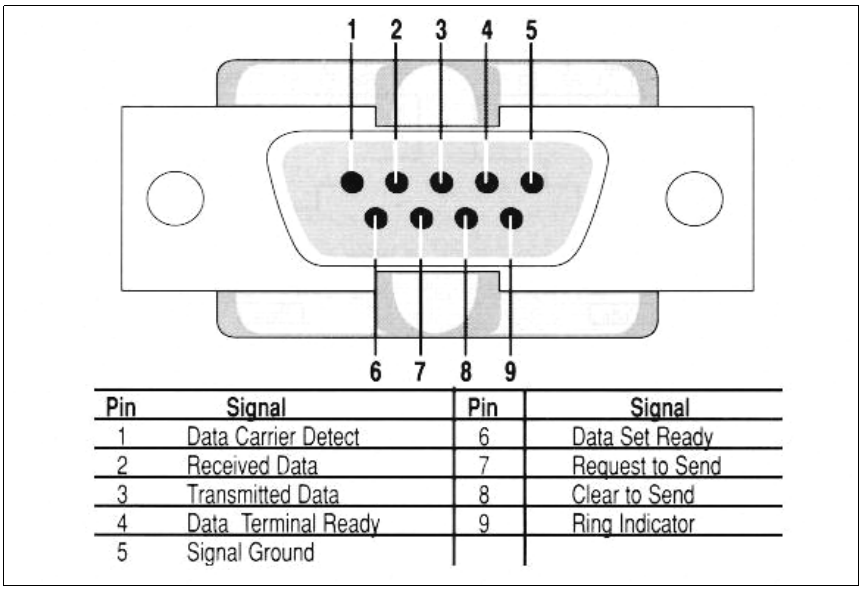
Comment récupérer la Pirq Table

Voici comment récupérer la Pirq Table depuis Kcore :

grep -a "\$PIR" -b0 -A0 irq.bin </proc/kcore

Ensuite, il faut faire un peu de travail avec un éditeur hexadecimal car le binaire doit faire exactement 256 octets. En ouvrant ce fichier, dans le contenu, le motif $PIR apparaît. L'important est de commencer « la découpe » au dollar de $PIR puis de compter 256 octets.

Câble



Pin	Signal	Pin	Signal
1	Data Carrier Detect	6	Data Set Ready
2	Received Data	7	Request to Send
3	Transmitted Data	8	Clear to Send
4	Data Terminal Ready	9	Ring Indicator
5	Signal Ground		

Ci-dessus l'interface RS232 9pin (DB9).Pour mettre en place une connection null modem. Il faut croiser les cables selon les chiffres de ce tableau :

Prise A	Prise B
2 et 3	3 et 2
4 et 6+1	6+1 et 4
5	5
7 et 8	8 et 7

Les scripts graphiques

Script de stress du device frame buffer

```
#!/bin/bash
#
#
i=0
while [ $i -ne 60 ]; do
 cat /root/img >/dev/fb0
# sleep 1
 xrefresh -display 0:0
i=` expr $i + 1 `
done
```

Script de tests des modes graphique du device frame buffer

```
#!/bin/bash
#
#
if [ "x$1" = "x" ]; then
   echo "Usage : $0 -test | -do "
fi

if [ "$1" = "-test" ]; then
for i in `cat /etc/fb.modes | grep "mode " | cut -f2 -d"\""`;do echo -n
"Testing $i"; T=`fbset -x --test "$i"`; [ $T ] && echo -e" \t\t '!'" ||
echo -e " \t\t Ok ";done
fi
if [ "$1" = "-do" ]; then
for i in `cat /etc/fb.modes | grep "mode " | cut -f2 -d"\""`;do echo -n
"Doing $i"; T=`fbset -x  "$i"`; [ $T ] && echo -e" \t\t '!'" || echo -e "
\t\t Ok ";done
fi
```

LinuxBios Command Center (LBCC)

Paramètres en ligne de commande.

```
[root@ge targets]# ./lbcc
usage: ./lbcc target_config.lb_dir action [options]
actions : -all,-a | --make-only,-m | --flash-only,-f | --payload-only,-p
         --all,-a                Edit all config files and make them all
and flash
         --make-only,-m          Make ROM only
         --flash-only,-f         Flash a linuxbios ROM
         --payload-only,-p       Make payload only
[options], --no-log,-n   --bind-debug,-b   --err-dump,-e
v.24/05/04|09:50:26 MD for Mangrove-Systems
```

Config.lb file commands

Inutile de répéter cf. Reference « LinuxBios on AMD64 Port Guide » section Langage

Les scripts de communication

Hear !

```
#!/bin/sh
#
# Hear script  listen & log onto ttyS0 (serial port a.k.a COM1)
#
#
VERSION="24/05/04|09:32:49 MD"
i="Bienvenu(e) dans Hear! v. $VERSION j'ecoute sur ttyS0 ..."
ter=`tty`
if [ "x$1" = "x" ]; then
  echo "Usage $0 log_to_file"
  exit 0
 else
  LOG="$1"
fi
echo $i
echo  "Logging in $LOG"
stty ispeed 115200
stty | tee -a "$LOG"
echo "**************************************************" >>"$LOG"
echo " Hear! Boot Log started on `date +%d/%m/%y\|%T` " >> "$LOG"
echo "  tty in use : $ter  from ttyS0 " >>"$LOG"
echo "**************************************************" >>"$LOG"

  /usr/bin/time -pao "$LOG" -- cat -sbA </dev/ttyS0 | tee -a "$LOG"

  ./digest.sh "$LOG" -s

  echo "-------------------------------------------" >> "$LOG"
  echo -e "\n"
  echo "Please add a comment to this boot log .."
  sleep 3
  vi tmp
  echo -e "Comments :\n" >> "$LOG"
  cat tmp >> "$LOG"
  rm -f tmp
  echo "done."
```

Digest.sh

```
#!/bin/bash
#
# Script to digest hard-human readable hear! logs
#  Mathieu Deschamps for mangrove-systems
#
VERSION="11/05/04|09:40:28 MD"
SUPP=0
DAT="`date +%d\-%m\-%y\ %T`"
if  [ "x$1" = "x"  ]; then
        echo "usage: $0 hear!_log_file [--suppress-log,-s] "
        echo "v. $VERSION"
        exit 0
  else
        FIL="$1"
fi

[ "x$2" = "x-s" ] || [ "x$2" = "x--suppress-log" ]  && SUPP=1

if [ -f "$FIL.adv" ]; then
  mv "$FIL.adv"  "$FIL.adv.$DAT"
fi

cat $FIL | while read line;
        do
                [ "x$line" != "x$" ] && echo "$line"
        done >  "$FIL.adv"

mv -f $FIL $FIL.digested
mv -f $FIL.adv $FIL
( [ $? -eq 0 ];  echo "$FIL digested, rename old as $FIL.digested" ) ||
echo "Digestion gets harden ...blurp (error) "

[ $SUPP -eq 1 ] && rm -f $FIL.digested 2>&1 >/dev/null
```

La CMOS Table

bios_restore et bios_save (avec leurs pages man)

bios_restore – restauration d'une image ROM
bios_restore /dev/mtd<n> <romimage> [--vendor VENDOR] [--part PART]
bios_save – sauve le contenu du BIOS dans un fichier image.
bios_save /dev/mtd<n> <romimage> [--vendor VENDOR] [--part PART]

cmos_to_file, cmos_utils, file_to_cmos (avec leurs pages man)

cmos_util - un utilitaire pour editer la configuration CMOS RAM settings.
cmos_util [-d|-u|-s|-h] [--image file] [--output_image file]
[--layout file] [--new_layout file] [--settings file]
cmos_to_file – Un utilitaire pour sauver la CMOS RAM dans un fichier image.
cmos_to_file file_name
file_to_cmos – Un utilitaire de restauration d'une image CMOS depuis un fichier.
file_to_cmos file_name

lxbios

Lit ou écrit les paramètres interne LinuxBios tel la LinuxBios table

lbflash

Solution de flashage LinuxBios dédié au MTD (Memory Technology Device) de type DoC.
Il vérifie la validité avec le type de carte mère.

Kernel Help !

En cas tout à fait probable de panne de bootloader, un lilo qui reste bloqué sur "LI" pa exemple. Un noyau qui crash est un problème surtout si d'autres sains n'auraient pas été prévus pour accéder à une ligne de commande sans crash. Ceci afin de lancer un `lilo` par exemple. Voici une procédure pour restaurer la MBR quand on a pas de lecteur de disquette mais juste un CDROM d'installation Linux:

But: avoir un shell, monter les partitions, vérifier l'intégité du disque, lancer un lilo de restauration.

Taper `linux rescue` depuis un CD d'install Linux par exemple type RedHat. Paramétrer le clavier et le langage en la langue choisie. Paramétrer eth0 et ftp si éventuellement il est faut récupérer des fichiers par le réseau (rc.sysint ou lilo.conf, etc..). Après le lancement d'Anaconda, acceder au terminal virtuel 1 CTRL+ALT+1 pour vérifier le présence du device bootable /dev/hda?? sinon il faut poursuivre quelque écrans plus loin (CTRL+ALT+7).

Remonter la partition en lecture écriture `mount -no remout,rw /`. Monter la partition root par exemple `mkdir /hda3; mount /dev/hda3 /hda3`. Faire pareil avec les autres partitions surtout celles où se situnt /etc et /sbin. Chrooter en /test : `chroot /test /bin/sh.` Un `loadkeys fr` peut être necessaire si la clavier est mal configuré. Une fois dans le shell créé editer au besoin lilo.conf et faire /sbin/lilo, enfin sortir avec CTRL+D. Synchroniser les partitions `sync` et rebooter `reboot.`

GNU LICENCES

LinuxBios is release under the GNU GENERAL PUBLIC LICENSE
Version 2, June 1991
Copyright (C) 1989, 1991 Free Software Foundation, Inc.
675 Mass Ave, Cambridge, MA 02139, USA

LinuxBios Licence

This software and ancillary information (herein called SOFTWARE) called LinuxBIOS is made available under the terms described here. The SOFTWARE has been approved for release with associated LA-CC Number 00-34.

Unless otherwise indicated, this SOFTWARE has been authored by an employee or employees of the University of California, operator of the Los Alamos National Laboratory under Contract No. W-7405-ENG-36 with the U.S. Department of Energy.

The U.S.Government has rights to use, reproduce, and distribute this SOFTWARE. The public may copy, distribute, prepare derivative works and publicly display this SOFTWARE without charge, provided that this Notice and any statement of authorship are reproduced on all copies. Neither the Government nor the University makes any warranty, express or implied, or assumes any liability or responsibility for the use of this SOFTWARE.

If SOFTWARE is modified to produce derivative works, such modified SOFTWARE should be clearly marked, so as not to confuse it with the version available from LANL.

Copyright 2000, Ron Minnich, Advanced Computing Lab, LANL,rminnich@lanl.gov

This document's Licence