Next Previous Contents

Project Book, Mangrove LinuxBios

Mathieu Deschamps, `mathdesc chez yahoo point fr`

v1.01, August 2005

This is a RD project book around LinuxBios, Freebios, OpenBios. It explains BIOS and boot mechanisms. It describes how to setup, build, customize, and burn a CMOS with your own Linuxbios free bios on x86 architecture. ***This document is an HOW-TO which describes a protocol to carry out a working Linuxbios prototype on VIA EPIA mainboard in a constraint milieu.*** *But much more than a HOWTO this project book : 1. develops reasons why solutions are promoted or demoted; 2. lets indications for other case studies based upon experiments; 3. explains indeep Linuxbios internal mechanisms; 4. proposes various architectures; 5. goes forward for Linuxbios developpers.*

Next Previous Contents

1. Introduction

By the time I've finished working on this project, an important question raised : How to respond best to three types of readers, my project manager, Linuxbios community and my university teachers ? Three documents wouldn't have matched the synthesis so I made up one single document, complete, in constant worry to stick to this peculiar audience without breaking its structure.

In order to fullfill Linuxbios copyright statement, this project has its own codename : Mangrove LinuxBios. In this document, I'll use Linuxbios term to consider Mangrove Linuxbios in Linuxbios. In fact, Mangrove Linuxbios is entierely what was Linuxbios when I worked on it, except concretly one Linuxbios source code update.

It has started a year ago, in a french compagny named Mangrove Systems. I've started a free BIOS research concerning embedded devices or devices running in a contraint milieu.

Of course, I want to render a complete overview of this project. I had a short project time and I've underlined the pros and the cons with the same objectivity.

Based upon my experiments, this is a first release, a synthesis, a snapshot taken from a wide project which constantly self redefines. This document thus follows the same rules. From what I know of, it is still up to date, otherwise I'am sure your relevant input would make it so.

New code names will appear as per industry standard guidelines to emphasize the state-of-the-art-ness of this document.

1.1 Copyrights

LinuxBios Licence

This document's Licence

1.2 Disclaimer

Use the information in this document at your own risk. I disavow any potential liability for the contents of this document. Use of the concepts, examples, and/or other content of this document is entirely at your own risk.

All copyrights are owned by their owners, unless specifically noted otherwise. Use of a term in this document should not be regarded as affecting the validity of any trademark or service mark.

Naming of particular products or brands should not be seen as endorsements.

You are strongly recommended to take a backup of your system before major installation and backups at regular intervals. ***Some mainboard manipulation, such as chip desoldering requieres some electronic knowledge and precautions otherwise you risk major hardware problems such as chip smelting. YOU HAVE BEEN WARNED !***

This document sums up a RD work based upon experimental tests. Thus I put some hypotesis, if you spot relevant mistakes or such, I will welcome your mail. Also in this projectbook, I come franckly speaking on what annoyed me. I hope it won't hurt anyone, I know how much it is a time-blackhole working on this type of project.

1.3 News

This is the first release.

Note that you can find the latest version number of the sgml template I derivated and used to create this document. It is fine. This template can be gleaned from this plan entry if you finger this Nyx account.

Also, the latest version of this template will be available on this web space on Nyx in a number of formats:

- HTML.
- plain ASCII text.
- SGML source.

Note that paper sizes vary in the world, A4 and US letter differ significantly.

1.4 Credits

I would like to express my sinceriest acknowledgements to

Laurent Texier, Mangrove Systems

Stephane Durand, Mangrove Systems

and all the Mangrove Systems Developpers

Ron Minnich at Linuxbios project

Stefan Reinauer at Linuxbios project and Openbios project

and the Linuxbios community

Stein Gjoen at Linux Documentation Project

Martin Wheeler at Linux Documentation Project

and the LDP community

1.5 References and papers

This project book reflects a work base upon organisation's and compagny's projects, papers, debates.

- Project References.
  - IEEE Standard Organisation.
  - The Linuxbios wikipedia.
  - The Freebios project.
  - The VIA Arena for EPIA-M specifications.
  - The Etherboot project.
  - The Openbios project.
  - The LOBOS Project.
  - The Linux Documentation Project.
  - The Linux Kernel.
  - BIOS PC/AT secrets.
  - Linux Test Project Kernel regression tests.
- Compagnies which contribute to (my) Linuxbios effort in no specific order.
  - http://www.mangrove-systems.com/
  - http://www.lnxi.com/
  - http://www.sis.com/
  - http://www.linuxlabs.com/
  - http://www.tyan.com/
  - http://www.onelabs.com/
  - http://www.cdy.com/
- And not to forget papers.
  - Minnich (R) with USENIX "Proceedings of the 4th Annual Linux Showcase & Conference Atlanta : LOBOS" Aug 14th,2000".
  - Minnich (R), Hendricks (J), Webste (D) with USENIX "Proceedings of the 4th Annual Linux Showcase & Conference Atlanta : The Linux Bios" Aug 15th,2000.
  - Reinauer (S) "LinuxBios on AMD64 Port Guide" Feb 10th,2004.
  - "Voyage au centre du noyau", Linux Magazine Hors Série n°17 Nov,Dec 2003.
  - "Linux embarqué", LinuxFrance Hurd Magazine n°36 Feb 2002.
  - "Kernel 2.4.0", LinuxFrance Hurd Magazine n°25 Feb 2001.
  - "Linux système embarqué", LinuxFrance Hurd Magazine n°17 May 2000.

1.6 Reading tracks

I know how convinient it is for you to read HOWTOs. But that is rather a project book, so I have adapted it for you from the very initial version to look like a HOWTO. I propose in this project book two reading tracks :

Either you read it all and, apart from getting the spirit of this RD project, you'll know why and how it's important to derivate a specific Linuxbios branch for your Linuxbios project. Most readers should run on this track because prototyping a Linuxbios is made simpler with a step-by-step approch;

Or you can jump directly to the HOWTO section explaining how to build a BIOS for EPIA-M motherboard. This is to say that, either you already read enough paper on the subject, either you consider yourself well experiencied in Linuxbios and in Bios matter, in the boot process and the multiboot chaining, in the linux kernel customization and in VIA EPIA specifications. Also note that, if you have any other hardware, you would certainly prefer the first track.

Please keep on mind, this has to be accessible to newbies. To do so there is a glossary and actually this document is full of advices, precautions, examples, schemes and clarifications --surely too many for experts !. But be aware that you can find pointers along this document to go directly to your specific point.

Next Previous Contents

2. Project

2.1 Abstract

At the edge of kernel technology, operating systems and libre softwares, proprietary logic still keep the hand on. If theses systems tends to openess, softwares and hardware specifications used to initialize them via a code, the BIOS, remains proprietary.

But several years ago, pushed by necessity, researchers and some manufacturers team up in order to widen the field of use, and to free BIOSes from their initial, deprecated and buggy implementation. In this aim, some projects are now available out of laboratory use, such as OpenBios, FreeBios, LinuxBios.

The goal of **this libre BIOS research and development project** is to boot a linux kernel onto a thin client from a customized and derivated LinuxBios BIOS. Linuxbios is a project lead by the LANL (Los Alamos National Laboratory) in close collaboration with OpenBios and Free Bios projects. Primarily it has been made to work in a cluster environnement in order to wake and initialize machine grapes. Applications : Grid computing, parallel computization, massive calculus.

What Linuxbios offers is simple : build, cutomize and flash a BIOS into a chip : the CMOS.

I've started working on the initial version as Linuxbios evolved into Version 2. Totally refactored it is more easy to port, derivate, understand. This book describes how and why I have derivated Linuxbios to fit embedded device. Developments had already targeted theses devices but none led to a documention support available to the largest audience like this is.

2.2 Structure

First task was to qualify Linuxbios, to identify "the pros and cons", and detect future difficulties during development process. See the feasability study.

Second was to look for 'functional gaps' if exists and propose external simplest solutions. In other words, "integration is my friend". See the Linux and the BIOS study. See the software solution study.

Third task consist of an implementation I made in order to carry out a prototype. This is the pure HOWTO part. See the implementation.

2.3 Tools and resources

Linuxbios father project from which my Linuxbios has derivated made me shiver by the time I realized plenty the sense of "bazaar organization". Do not let this blizzard frizz you to the bones : welcome to the Permafrost ! The source tree is "just" 800 subdirectories including 590 for the src/ directory ! All right, this WAS version 1.

Don't worry, version 2 is now just 350 including 240 in src/. But take this advice to you, do not dump too quickly version 1 from your CVS tree because this is a treasure cave talking about source code.

Let's also say this frankly : there were very few documentations in the immediate surroundings of Linuxbios. Nevertheless, useful informations have aggregate along Linuxbios pipermail forum and mail archive : forum questions/remarks and answers mechanisms works correctly. (cf. quick landmarks in Linuxbios maillist archive)

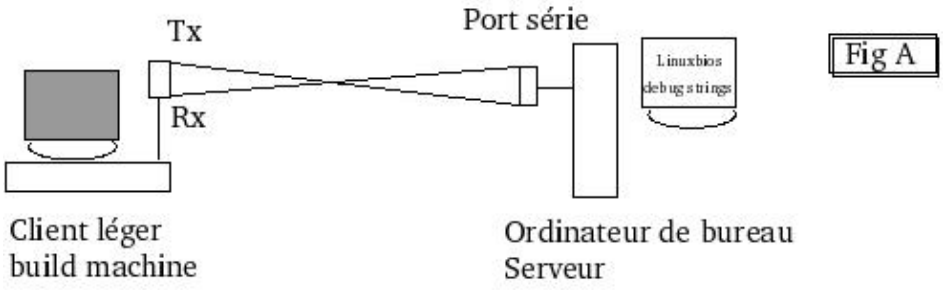
If you are ready to get started, I'am sure it's no use asking if you read the disclaimer or remembering you to save your personnal data, or remembering you to have a boot cd or a boot floppy at hand, or remembering you to save your CMOS state, or not to test code in root mode (as much as you could). hehh don't be that touchy, that was just to mention... ;)

Well, let's talk about architecture.

Please note this doc will not cover crosscompiling. I've worked in the simplest case : one machine, called build machine, will (guess what) build your Linuxbios from the source depending on software and hardware environnement. Actually, it is also the target machine. The other, called server machine, will recovers debug string from the BIOS while running. Moreover, it can serve kernel dynamically to the build machine to boot Linux or another OS : Windows, NETBSD, Plan9... You will see that the boot matter is merely described in this documentation.

When you'll have your Linuxbios BIOS burnt and running, you'll have to read debug lines to see and check out if sequence is ok or ko : this is debug mode. Typically Linuxbios send theses debug strings to serial port (ttyS0 aka COM1) but this could be changed. Theses strings couldn't be displayed onto the build machine because video card/chip initialization is done at the end of the BIOS sequence. Thus, build machine screen could remain off until video is set up. And this is why a null modem cable is needed to connect build and server machine serial ports. This enables networking... basic networking yes but networking anyway.

Linuxbios Test appareil showing the null modem cable connection to retrieve debug strings from client to server.



You can note that this is a classic serial cable, where Rx (Reception pin) and Tx (Transmission pin) crosses (crosswired). See here to make your null cable modem.

My target machine (build machine) is a VIA EPIA-M mainboard for which a bios will be build. It is powerfull enough machine to build a kernel self without having to wait too long. By the way, that is why crosscompiling can be avoid. See the complete hardware description of my build machine. Please refer to Linuxbios web site to get a list of supported, tested mainboards.

2.4 Feasability

In order to qualify Linuxbios project but also to determine whether Mangrove Linuxbios could succeed or not, and under which conditions, I made various studies. Actually, Linuxbios as every project has its risks and

ablities, typically if you derivate into another use.

There were visible success and hourrahs about BIOS first run in the forum. But some success were very humbles. This study's main objective was to collect testifies and experiences to draw a neutral review and to qualify Linuxbios.

Risks

- Novelty

  Born in 1999, Linuxbios has grown but not all functionalites are tested. Some are newly available but are still unstable. There are two Linuxbios major versions V1 and V2. The later mainly responds to user's need to run it on recent motherboard. Best is to stand a constant technology review to get the lastest patchs and updates. It's also good to test functionalities and running BIOSes on a stress test protocol basis. Lastest notes : Linuxbios home site is fully refactored in a wiki style. More ports, ressources, docs are available.
- Deprecation

  Some low usage Linuxbios items are left abandonned. Most of the time, either that item malfunctions, either it is too much of a pain to get it working properly. Even sometimes, no one seems to understand its usage but if it is present in the main CVS branch, it surely had a past glory. If you don't want to make a bad choice, remember reading the very relevant changelog (when author had time filling it). Also look for the maintainer activity : too busy maintainer are like soccer's goalkeeper defending on several balls. If you'll have to work with V1 and V2 (eg. code porting) remember another simple idea : many compounds, many updates !
- Exclusion

  Apart from some manufacturers and hardware vendors, still most of them retain their specifications and informations. Obviously, such a bios program has to 'know' all of hardware layer to initialize it, so this is a major risk. Worse sometime you believe having endly an access to specifications whereas you just hold a commercial or marketing advert. looking like true papers. Stick working on manufacturers and vendors that actually support Linuxbios and get your stuff with them. Also be careful because, if you get relevant infos from an employee who is tied with a N.D.A (Non Disclosure Agreement) you could bring problems to the community, and your work could be tainted.

Abilities

- Quickness

  Linuxbios is reknown in enabling fastest boot time ever, 1 to 10 seconds to OS hand over depending on the OS. This is mainly because Linuxbios activates what is strictly needed by the kernel. On the other scale, Linuxbios project which follows an FLOS colloborative development have various active and skilled vulunteers.
- Adequation

  It is important to note that what Linuxbios offers aims to be at least as good as original proprietary BIOS. Even better, most of the time theses BIOS are buggy, more rigid and less efficient leading to counter-performances. Because Linuxbios is ported to many hardware, it is bound to fit to your specific machine and it offers a wide OS boot abilities like Linux, Debian, Windows CE, Windows NTE, Plan9 and many more.

- Power

  Linuxbios propose you to master the BIOS from the begin at BIOS init. time, down to the end at OS boot time. Thus, you have complete hand on the entiere loading of your machine, reducing the frontier of theses two codes. Theorically this spots redundancy and, if erased, permits some gains. Also, this spots that some attributions until now attributed to BIOS have to be redesign. Moreover, this type of bad assignments and drove-off generates capacity limitation which forces frequent updates and costly hardware renewal. At least but not last, accessing the BIOS source code permits inline tests from OS but also distant BIOS updates.
- Customization

  This early access to the BIOS widen the field of software actions. To the very first microsecondes of BIOS initialisation you can add your own compound : authenfication modules, splash screens, pre-networking pane, security pane, etc.. And of course, you can boot your OS from a IDE disk, a CDROM, a floppy, a compact flash (CF), a disk on chip (DOC), a disk on module (DOM), a distant disk (NFS)..
- Reduction

  Linuxbios arrange your BIOS memory regarding to your chip size. If you boot your OS from a DOC/DOM, your BIOS could reside just beneath it, reducing BIOS+OS space to fit one uniq chip. Reduction it is also of the cost, know that your Linuxbios is GPL'ed, sparing you the expensive licensing costs of CMOS chip and BIOS software.

To conclude this study, indeed Linuxbios has fair argues on many points. Customization and reduction abilities tends to favor my embedded device development. Maybe you can find your trend in this bilan but it's surely not enough to draw out your Linuxbios project. From now on, I'll use Linuxbios to designate Linuxbios V2. Let's consider Linux and the Bios together and then to see that frontier.

Next Previous Contents

3. Linux and the BIOS : Linuxbios !

In this part, I explain the link Linux-LinuxBios and Linuxbios role and action. Linuxbios result is a BIOS like any other. First as a general matter, I describe what a BIOS do in details and what resources it offers and handles. Second, I'll explain the specific Linux device handling regarding to other OSes. Third, I'll try to explain how vital it is to consider the frontier between BIOS and OS helped by a synpotic boot chart. At least, you'll also get a simple howto customize Linux Kernel for your Linuxbios.

3.1 Generalities

Maybe it's useful to explain some general notions of BIOS behavior.

Bios code is stored in an EEPROM called variable size CMOS chip, often it is a 128Ko or a 256Ko chip located on the motherboard near a flat round battery to sustain electrically this code in case of power cuts.

When you got a bad BIOS code or a bad configuration "a manual override" processus can be undertaken. It consists in powering down the motherboard and taking off that battery for a dozen of seconds. Upon rebooting, BIOS (partly) ereased code will raise a checksum control error. As a result, the rest of this code will launch a default recovering procedure as you'll act according to "Checksum error -- Hit F1 to retore default values" display. Last thing to mention, nowadays BIOS codes are copied into RAM memory, this common BIOS ability is called 'bios shadow' or 'Shadow RAM'.

When system is on, cpu self tests its efficience : voltage, frequency, etc.. If correct it emit a *PowerGood* signal which spreads to activate PnP compliant cards and other controllers connected to motherboard. CPU initialize with its default values and know it has to read 0FFFF:FFF0 address and to jump to the address it holds (See the complete boot sequence). *this is the first BIOS code line*. Then it determines the CPU L1 cache memory and the CPU L2 cache memory. Remember thoses leveled memory caches as temporary zones for CPU calculus chains. When it is done it tests so called 'compagnion chips', for instance on some motherboard, an arithmetic coCPU is tested with a built-in dedicated instruction.

All theses steps are made into the ***real mode*** but the rest will run into the ***protected mode***. Real mode is based in 16 bits and thus can't map all memory. That's the main reason why it has to go into protected mode which is based in 32 bits. At this junction, CPU is up and running fine so it stands as a grant for 16 ot 32 conversions, that why it's called 'protected'.

Now that a protected runtime environnement has been triggered, BIOS has to reinitialized CPU to rerun it with the *optimal values* . This phase is essential to define and initialize all devices and hardwares around the CPU. Bios could then use assembly call CPUID to identify CPU. This instruction returns best fitted values matching CPU model and working.

This is the end of the 'core intialization'. Then it is devices init. time, BIOS sends activation bytes to hardware controllers. To keep it simple every device use a base controller --a chip. It is an interface with a lowlevel dialog protocol between hardware and BIOS. Even better, for Linux and from what I have understood, the dialog is set from hardware to kernel itself. (See Linux Hardware Abstraction Layer).

3.2 Specifications

Linuxbios as any other conventionnal BIOS is able to handle/do :

- Input/Output (I/O) : Main attribution. This is precisely how to read and write to thoses upper said controllers of a given device (eg. graphic card, audio AC'97, serial port, keyboard, hardisks, ..)
- Mass Storage : Indicates where to find OS on a disk. It reads the CHS structure within this disk's special ROM, and validates with some basic tests.
- Internal memories : It mainly describes RAMs. This computer item is one that has most changed but quantity and response time are still computed at the same time. The BIOS also indicates what protocol to use to make it work as well as it point where to find back some specific tables and descriptors.
- Internal Clock and Timer : By extension, it also handles the power saving ACPI. From what I know of, Linuxbios still doesn't handle a stable ACPI so it must be turn off.
- Buses : Theses are devices freeways policies. PCI et AGP are handles by Linuxbios which determines their width and speed.

Except this grand initializer role, BIOS offers resources, to be more precise, gateways to access them. Theses resources could then be consulted and accessed. Notably the Linuxbios :

- Maintain table descriptors (LDT,GDT,IDT) and some selectors if it's in real mode. When protected mode happen, it have to establish pagination tables for 32 bits advanced memory handling, that's a kind of addressing conversion and matching tables 16-32.
- Maps direclty a RAM memory space to hardware devices via the DMA channels sparing CPU the trouble of this charge.
- Assign theses devices to a reference : IRQs which targets an interruption code enabling I/O access to programmable interruption controlleur (PIC).
- Creates heap and stack permitting second generation code execution. Thus instead of assembly code, it is then possible to run C code.

By the time computer architecture have no more secret to this pre-system, BIOS looks for the OS. Depending on boot order which is a BIOS option that Linuxbios regard as a parameter, it goes to fetch MBR/PBR on disk, floppy, whatever it is... Then it looks for an id value 0XAA55h which tags a *bootable sector* and obviously a disk that contains an OS. BIOS copies this MBR in RAM and orders CPU to point to this RAM MBR copy. This sector is actually executable code which loads and decompress the Linux Kernel (if you deal with a GNU/Linux). At least, this short stage (called 'Stage 1' in GNU/Linux) ends after positionning stack pointer to first line of OS code. Well that's it, that's a BIOS life.

All right, you can see it as a whole in a synoptic flowchart. But before if you wish there is still a delicate point for you to see. If it's a fact that devices are set in a way BIOSes are bound to identify them, it's not all the same for the OS. There are many reasons to this in which glitchy proprietary BIOS codes, counter-performances boot sequences and unsuitable OS-BIOS frontier, both leads to a fragile design. Next follows what Linuxbios exploits which explains its power upon other BIOS.

3.3 Abstract device layer

Linux adds an immediate hardware controller-Linux interface, shortcutting greatly classical controller-bios dialog and bios-OS dialog (bios service). Note is precisely true when theses interfaces are in the kernel land (builtin) and not as much when they are in the user land (modules). Equally, theses codes are simply called *device drivers*, they establish a link between an architecture and a protocol to enable dialog.

For instance, an interruption : a file read request emitted by a lambda application whereas the main handler is OS.

"Layers OS" will waste energy in echoing this simple instruction throughout various layers until delivry at bottom hardware layer. Also, worse is that resulting treatment, the answer, has to travel back until the top layer. You can notice in the following flowchart that actually effective interruption's handler relies on sole BIOS.

In other words, end-of-the-line request is processed by a proprietary, "rigiddy" hardware oriented BIOS code which is based on an old BIOS services concept.

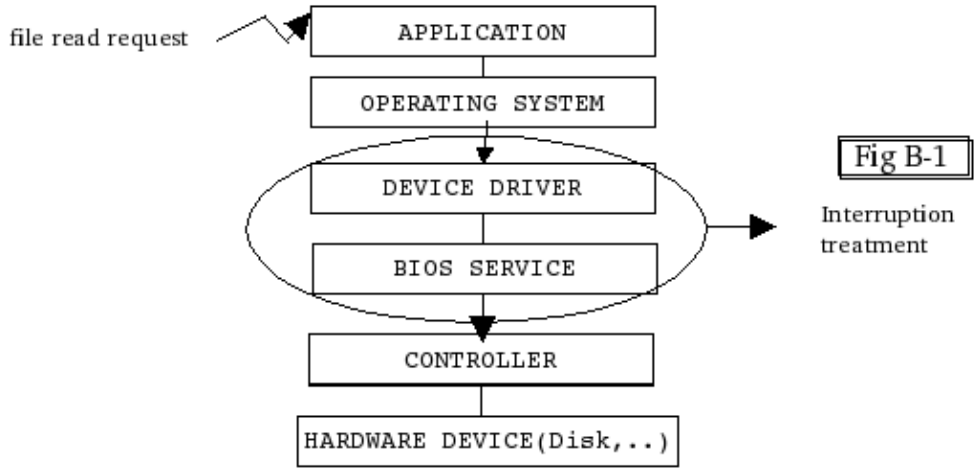


Fig B-1

This didn't bother anyone until relative recent widespread use of networking and personnal computing has made computers to be used in a new way. Hardware handling has reached a new point; where grid processing and mass storage centralization, or the opposite, small scaled hardware (embedded devices) and massive client display requieres more efficience and quickness; where security and viability wouldn't obviously lead to many time-consuming integrity checks at I/Os. Linuxbios seems to aim also to theses objectives because of the Linux Kernel.

The Linux kernel as it is, is naturally at this rendez-vous point. Linux self implements a "generic layer", an "hardware abstraction layer" or still a "virtual device", whatsoever you call it. Thus it does not need any bios service from BIOS. Linux is to fit every type of device in order to prevent too frequent BIOS updates. This also means, that you could buy your hardware regardless of what your classical BIOS would accept.

Ok now, let's walk back on the ground, in fact obviously, this means porting effort anyway in the kernel notably but not only. Also, if you've made your own card, you got to make your own driver to fit in that layer. However, Linuxbios already supports many mainboards and handles succesfully numerous chipsets. In below flowchart, you can see difference and realize that this layout enables a single mapping and a signal centralization controlled by Linux OS solely. You'll see later on in the next section that indeed maintaining upwards this logical unity is a fair defy. To sum up roughly : it enhance global performance for response time are shorten. Useless to mention that this is utterly wished for upper said applications field (embedded, constraint milieu as well as clustering, grid calculus).

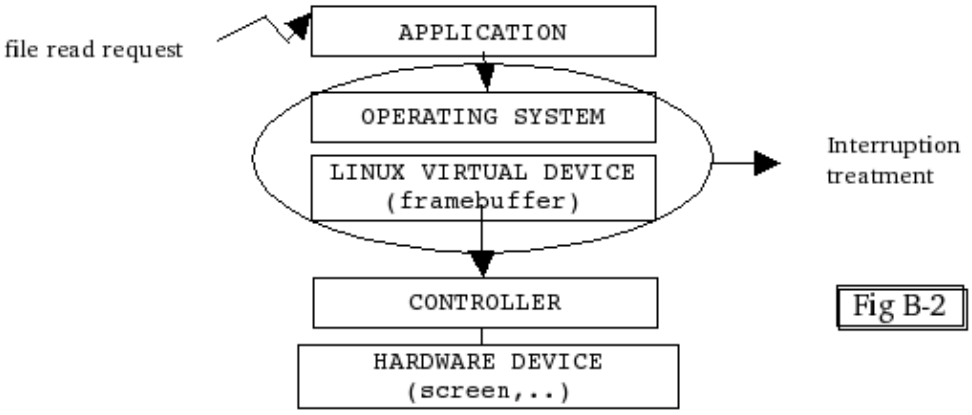


Fig B-2

This architecture boosts performance from BIOS to OS. But BIOS obviously remains hardware dependant. Linux offers a great flexibility without it wouldn't be able to run a freebios such as Linuxbios. For beneath flowchart instance, to display, a frambuffer driver exists which is attached to this abstracted device layer. Does such an abstracted device layer would exist for your typical needs, if for example you feel like booting from a Compact Flash (CF) ? This is what you have to consider if you ask your freebios to be viable and flexible. Because you don't want to rebuild and burn a BIOS every time you change a device, or you change a boot order, (do you ?) then you need to consider where to put the frontier between BIOS and OS.

3.4 BIOS-OS frontier : a crucial choice

I come more on the theorical point in this part...

Without BIOS, or rather if it was completly integrated into the OS, this latter would be solely and wholly in charge of hardware. Because it seems inconceivable to change OS as you change your sound card, BIOS exists. I see you start bending over yourself, does this frontier recalls you the kernelspace and userspace 's one ? Heh heh it is, so why not asking yourself if Compact Flash handling, for instance, could be directly flashed in BIOS rather that present in the OS?

The answer is obvious for such a easy question. If you usually use a CF to BOOT your OS, you'll merge the handler within BIOS. It would be a pity to wait for OS to take over the system so you could use a functionality whereas you could have use it earlier in Linuxbios. Moreover, for the CF instance, it become compulsory to have it earlier within the BIOS to boot from it.

Ok, I feel you might start thinking that we are running to nonsense by pushing down to BIOS layers what's used to be into OS one. If you can't reconsider this maybe you just don't see the scope of this project...

Now let's stray for a while, as evoqued in the previous section most OS, to carry a problem out of its sight, layers proceeds requests, by redirection, that is nearing the idea : "What's not my job is surely lower/upper/lessloaded layer's one". Look, BIOS is far from being alone to work like that, in fact all computing items, real or virtual, works upon this idea from the most low-level piece, BIOS, to the most high-level one, Internet. It is just a very "pyramidal concept" which is looking to dispatch and assign jobs or requests.

This is not a strange coincidence, it has historical reasons. For instance, if a software realize the cpu can't do 150 * 33 because it lacks a bigger operation register for example, then we write it to do ((150*3)*11) and even (((15*10)*3)*11). Surely, You catch my meaning : this requieres a I/O stack, and processes to picks up and compute operations. All right, and when all this runs in more complexity, speed, capacity we have to

wonder why still go on writing it up that complex way, why using it still the same way ?

Also in order to get things running faster, to shorten render time while systems naturally tends to more complexity, we just aim to keep whole computer software stack as thin as possible, I/O between layers as fast as possible, etc.. That's it for embedded systems as for computer clustering, parallel and grid computization. We have "granularated" up/down that 'basic protocol' and until we have nothing new, more intelligent, to fit compatible (or not ?!) with this, we'll be keeping on running after Moore's law.

Enough 'trolls' on new theory, let's come back to our herd. If you need a proof of the existence of that new frontier, for example, when Linux does a new IDT description it overlap onto the classical BIOS attributions. Also, this cost extra time to do it once and for all (session time) at kernel loading. But after that, OS (the kernel) is able to spare BIOS the cost of dialog in frequent cpu interruption situation, leading to greater efficiency. Realize this simple cursor slide to OS side thus represents a cpu time spare *before it is interrupted* . Because by setting OS as true interruption leader, which manages time planning and interruption planning together, which masters interrup trigger mode, it can be more efficient. Since interruptions are everywhere talking about devices, talking about grid computing also, guess how up can reach this "optimization"...

This is a 'cursor adjustement' example if you prefer. Another is MTRR initialization, but you could get more infos onto LANL's Linuxbios site. Most optimizations comes with code to refactor or to rewrite, also sometimes enabling a high profit comes nevertheless with a redundancy. This is quite paradoxal isn't it, while we talk about efficiency ? True, but not weird. After all, this is rather "normal", for example the CPU is defined and initialized twice, the second stage bring advanced values enabling a more precise setting up, to exploit hardware the best way. Nobody find it weird or anormal. Same way, failsafe or fallback mechanism doesn't figure as secondary functionality. If Linuxbios has chosen the fallback mechanism, it is because it saves system from going into kernel panic or fatal error. So, indeed a better cursor adjustement will automatically raise the quality of your Linuxbios, and satisfaction of your Linuxbios users. Also, think of what is going to happen by bringing such code from BIOS to OS, and such other from OS to BIOS.

It is still very delicate to establish a frontier on the long term. Actually, I think we should let this as a floating frontier, moving according to various applications. It requiers to be redefined to match your applications needs. I can feel the most skeptical of you fearing for compatibility, don't worry we'll find other bridges. Until this raffines it needs to remain open, to enable people to think about this.

Now you can refer to boot sequence flowchart. It is my proposal to fit most common Linuxbios needs as well as mine according to IEEE Standard.

3.5 A linux kernel built for Linuxbios - howto -

This piece of work is essential, if you never build a kernel...err... summon up your patience/perseverance (SUYP). Now, you must have been enabled to qualify your needs, designate your platform and to test it. The next step is to customize a Linux Kernel in order to boot a given OS from a given device. I have chosen end of 2.4.xx series Linux Kernel 2.4.25 because it was the last kernel before 2.6.xx series. Moreover if you want to try other "exotic" kernel codes, don't hesistate to stress test 72 hours long or more. (cf. ref Linux Test Project (LTP) regression tests).

OS to be loaded by Linuxbios is totally BIOS independant although as presented earlier, Linux partly redoes more efficiently part of what a standard BIOS is used to do. So there's no much miles from using a kernel as an initializer. Indeed, a Linux kernel will boot an OS, which by the way can be a-n-other Linux kernel. I won't go for a extended howto on kernel building but I'll give you some quick info so you can go directly to the point.

You need the kernel source (cf www.kernel.org) and a setup file to configure kernel. Normally, you'll find documentation in the source to do it right. Be careful before replacing your build machine kernel, actually don't replace, add it in plus to your standard kernel. It'll serve as a failsafe kernel. Really, do extensive tests although they are time consuming they will prevent you from realizing further on that it's no a trouble with Linuxbios you have but one with your primary kernel. A crashing kernel which often freeze system will waste twenty to thirty maintenance minutes to restore previous working status... except if you have a stated recovery procedure.

In order to compile your Linux kernel, you need a minimal requierement software toolchain version. It is minimal also you could even try to strip off PPP or ISDN to gain more space if you have no use of them.

I can't set in detail here what you'll have to enable in your kernel configuration to accord your Linuxbios kernel to your motherboard hardware. You can display this type of information via a *lspci -vt* or even more in detail with a *lspci -vv*. For example, and corresponding to my experiment build machine (description) I had to install VIA CXX chipset and the C3 Ezra CPU. *Devices supports can be taken either **statically** e.g merged to kernel, either **dynamically** e.g insmoded as modules in system from boot configuration files. Also keep on mind that the more supports you add to your kernel the more will it be weighing. Modules insertion is time consumming while booting. Light and compact kernel will decompress and load faster. I insist heavily (ruler slide), a sensible policy on this matter infers directly to boot time performance.*

For example for VIA RHINE driver support (i.e VIA EPIA integrated chipset network controler), here is how you can specify in the kernel configuration file this choice (specify only one of theses of course)

```
#VIA RHINE network controler support as a module
CONFIG_VIA_RHINE=m
#VIA RHINE network controler support as native
CONFIG_VIA_RHINE=y
#VIA RHINE network controler support deactivated
CONFIG_VIA_RHINE=n
```

Also you might be interested in knowing some other kernel configuration option about Memory Technology Devices (MTD) such as DiskOnChip(DOC). Of course, remember to add m/y/n to activate the option.

```
# DOC1000 serie chip handling
CONFIG_MTD_DOC1000
# DOC2000/ME series chip handling
CONFIG_MTD_DOC2000
# DOC2001 serie chip handling
CONFIG_MTD_DOC2001
# Onboard DOC chip autoscan (probing)
CONFIG_MTD_DOCPROBE
# DOC chip Bootable sector handling and probing
CONFIG_MTD_DOCPROBE_55AA
```

Typically, here are commands that launch kernel compilation and modules building :

```
$make menuconfig [or] $make config
$make depend
$make bzImage
$make modules
$make modules_install
```

The two last commands build and install modules into the default location /lib/module/_KernelVersion_. Then you could insert a module for example VIA RHINE support module into system with *insmod /lib/module/2.4.25/kernel/drivers/net/via-rhine.o*

Linuxbios add itself some line in configuration file. By default, it enable ACPI function, you have to deactivate manually it in order your bios functions at best. However, a bad set up on this point won't affect BIOS starting. At worse will it raise an error in bootloader while loading and decompressing kernel.

Bootloader is a short binairy code located in MBR, it's loaded at the end of BIOS startup. This code can be classical LILO or GRUB or whatever program formatted in elf or a.out, depending on what was set up in kernel configuration file, depending on what format the kernel has been told to recognize. Further booting study will describe different possible bootloader combinaisons in Linuxbios. A bootloader is perfectly standalone thus in its code a system root partition must be defined as default. Nevertheless, via a commandline you always could (value "prompt") instruct it to consider another root partition. Actually, bootloader parameters are passed by copy (recopied) in kernel parameters line. Thus, you can early instruct bootloader and kernel with some options (cf. a full Kernel Howto). *Let's state some Linuxbios dedicated useful options :*

```
root=/dev/dha3      - Tell where root partition is.
console=/dev/ttyS0  - Activate I/O on serial port as a console.
vga=ask             - Prompt for desired vga mode line.
```

In deep to explain kernel loading, this procedure have two stages. Fisrt, bootloader loads a strap, a small kernel piece, and a message displays something like "In first stage loading". During this stage, pre-kernel install VFS to mount root partition in readonly mode that holds a second kernel image. Remember this detour is compulsary because bootloader is just a 512 bytes long block it obviously cannot holds the entiere kernel. Second, kernel have "hands-on" and oftenly a message displays "Second stage loading". Then it initializes devices drivers and it remounts its root partition in read/write mode (mount option remount,rw). Bios data about device parameter are stored in pci_dev structure.

```
struct pci_dev {
struct  pci_dev         *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];
}
```

All of theses steps can be followed via Linuxbios debug strings and this given experiment appareil. You can also do this if you spend some money in a PCI POST card because Linuxbios sets a POST handling.

If there is a RAMdisk, another kernel option, then it is executed. A device driver has installed the main console so it be copied as virtual consoles, to create more console. At least, a bash script such as rc.sysint is executed in a terminal console. Note in V5 compliant system (all of GNU/Linux and most Linux distributions) that INIT process can start up and install swap memory as well as starting up servers services and deamons. At last, X server is available can be initialize and then you have your usual favorite windowed working space.

Preparation stage is finished. Matter is now on knowing how Linuxbios works to carry out a free BIOS prototype. And to do so, I've explained what kind of software solutions you may use. You could by yourself consider which one best fit your Linuxbios need.

Next Previous Contents

4. Software solutions for your Linuxbios

When I started working on this project, I wander around to see different possibilities to build a free bios. The first question was thus indeed : what type of free bios GPL'ed source codes are available ? But second vital one was : which one would enable me to carry out rapidly a prototype ? The stakes were simple, answer theses question, realize this answer concretely, and compile all of my remarks and notices in a reference documentation.

Early in this process Linuxbios set itself as the uniq base solution, and I can say now obvious. Itself falls into many aggregated solutions, Freebios project, Openbios project and Linuxbios have united forces. Indeed, other smaller projects primarily external then became "internalized" solutions as more and more people came by and wanted to use it.

To build a Linuxbios BIOS you basically need to know more about Linuxbios base solution but you need also a flashing solution to burn your Linuxbios into a BIOS chip. Of course, flashing solution study will explain what to use from flash utility to BIOS chip. Moreover, you'll need a logging solution to "see" and record your Linuxbios first run activity. Next is for you to consider what booting solution and what payload i.g Os to boot you will need. At last, you will see that the display solution is still quite a piece of problem, whatever not fully implemented on EPIA motherboard.

4.1 Base solution study : Linuxbios configuration

Well here we are. If you read "why Linuxbios" and "played with kernel building", still you need to know what tools it offers to configure your free and open specifications bios. It's not as complex as you think... except if you are looking to master the subject and/or to develop upon Linuxbios. All the same, you have to go through some short generalities before configuring properly Linuxbios. If you intent to master, you could then dive in Linuxbios advanced features.

Generalities

When it has begun at LANL, Linuxbios was meant to boot kernel in a clustering environnement (cf. glossary cluster) it was fetching on a server kernels. Application has now extended. Linuxbios fills the gap left by actual proprietary BIOS and corrects their errors which make unrealizable, unsupportable maintenances on machines and which harden human-machine interactions. For instance,"F1 press" after every BIOS update is a dull, repeatitive task when you have hundreds of computer in a grape. Also who would wait memory test to pass at every turn on of a MP3 pocket reader ? And how to "Press F1" without any keyboard if a BIOS update could be made possible ?

In all seriousness, beyond it is obviously unconfortable, beyond it is utterly closed source, it's a matter of fact theses proprietary bioses still maintain alife old fashioned functions. Theses are not used anymore such as DOS support whilst newly requiered functions are not developped such as large ROM storage support, as large RAM storage support, etc..

Linuxbios thanked to full configuration "compilation" mechanism satisfy to this. Since primarily Linuxbios is a divertment, its design and developement is inspired by the wish to write as minimal code as possible in order to let Linux kernel do the rest, what it does from the start. Indeed, as evocated before Linuxbios falls into two major parts : a bootstrap and a Linuxbios Linux Kernel prepared in the howto part of the linux kernel

study.

In order not to have any troubles compiling Linuxbios itself, refer to Linuxbios software toolchain

All right, let's come to Linuxbios working. To build up Linuxbios from its source, three elements are involved :

- A makefile file;
- A assembly code crt0.S file for real mode;
- A link file ldscript.ld to vital C files for protected mode;

Crt0.S sets enough code to start up binary Linuxbios bootstrap (although written in C). Ldscript.ld defines this bootstrap's reallocation adresses in order it could be found equally into a 256KB BIOS ROM or into a DOC 2000 8MB.

Another Linuxbios fundamental feature is that, all of the info on your build machine hardware is dispatched in all source tree directories involved in your BIOS building. It's you, when tweaking highest level configuration script, whom indicate for what motherboard, extensions and hardwares you wish your BIOS to be built. Below is clarified what you can find in Linuxbios source tree from src/ directory :

```
arch/        Crt0.S, ldscript.ld as well as Linuxbios Descriptor tables and other pirqs tables fo
boot/        bootloaders code : elfboot and filo.
config/      High level Configuration file architecture independant. Options file
console/     Code installing base console (text or vga) as well as UART825038 file
cpu/         Init. and reinit. codes  CPU, MTRR, cpuid.
devices/     For PCI resources assignment and PnP handling.
include/     Same nested tree for headers and definition files.
lib/         Redefined memory allocation code file.
mainboard/   Supported mainboards initialization code. (if it's not in, you're not)
northbridge/ Northbridge chipset init.
southbridge/ Guess what ...
pc80/        8086 arch. common compound init. such as keyboard, etc..
sdram/       Code to probe SDRAM
stream/      Bootload stream check code.
```

Config. files stands out as landmarks all along the source tree. Either they look like a mini-makefile, either they look rather like dialect codes. For instance, have a look to this one, boot/config.lb, which configure boot behavior :

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

Configuration

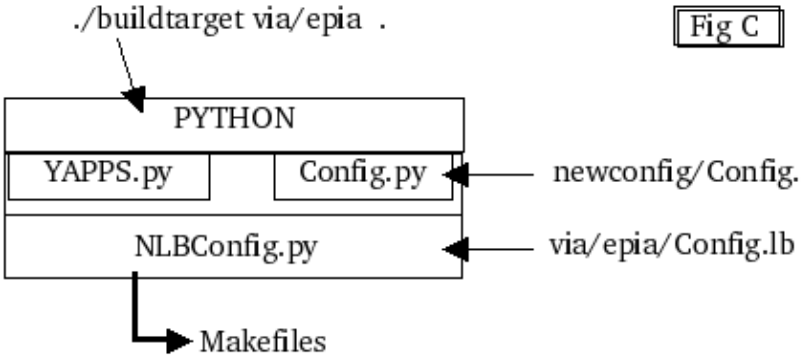
Theses beneath instructions are indeed compilation directives, they aren't directly understood by Make but they are by Buildtarget.

Buildtarget is a shell script located in target/. When executed it parameters and generates Make file as well as crt0.S and ldscript.ld files in order everything is right to compile.

```
[root@ge targets]# ./buildtarget
```

```
usage: buildtarget target [path-to-Linuxbios]
```

For instance, target for VIA EPIA is via/epia (base directory is mainboard/). Although second parameter is facultative it is strongly recommended for you shoudn't build targets in sources to prevent corruption. Buildtarget call Python interpreter with NLBConfig.py, both of them understand configuration dialect code.



Buildtarget triggering scheme

NLBConfig.py scours config.lb file, analyzes syntax, and check used keywords in comparison with Options.lb dictionnary file located in src/config. This file assigns default values to each keyword. It look like this (truncatured) :

```
          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
```

"Father" Config.lb file calls reccursively every "son" Config.lbs which self instructs unit C file compiling by the 'object' keyword (as "son" file boot/config.lb shown it previously). You can retrieve exhaustive instruction list from Stefan Reinauer's AMD64 Port Guide available at openbios.org or at Linuxbios.org.

I have spotted in my Config.lb some landmarks for you to compass :

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

uses ARCH                               |
uses CONFIG_COMPRESS                    |  Each Variable declaration *compulsory*
uses CONFIG_IOAPIC                      |   before before use
uses CONFIG_ROM_STREAM                  |

[...]
```

```
# Comm settings
option DEBUG=1                          |  Debugging mode activation &
option TTYS0_BAUD=115200                |  serial port speed config.
option CONFIG_CHIP_CONFIGURE=1
option MAXIMUM_CONSOLE_LOGLEVEL=8       |  Verbosity level for debug
option DEFAULT_CONSOLE_LOGLEVEL=8       |  string displaying to console

# Graphics settings
option CONFIG_CONSOLE_SERIAL8250=1      -- Activation of UART8250 dialog protocol for serial port
option CONFIG_CONSOLE_VGA=0             -- Ggraphic mode  VGA deactivated (if activated serial po
option CONFIG_CONSOLE_BTEXT=1           -- Btext mode activated

# Misc options
option CONFIG_COMPRESS=1                -- Enables gz'compressed kernel handling
option MAX_REBOOT_CNT=2                 -- Max reboot count in case of error before bailing out
option CONFIG_SERIAL_POST=1             -- Display POST phases debug
option HAVE_HARD_RESET=1                -- System hardware reset native handler
option AUTOBOOT_CMDLINE="hda1:/vmlinuz root=/dev/hda3 console=ttyS0 "
                                        -- Boot command line and Kernel passed parameters :
                                        -- kernel is called vmlinuz and is located on hda1
                                        -- Root partition is hda3, main console is redirected to
[...]
```

Difference between every "son" config.lb and "father" config.lb will become more evident to you after editing many time various one. Primarily, Config.lb are names after that sense guiding you to edit and modify their content in order to make things working properly. But be careful because in a unclever, confusing manner they also named another file Config.lb, located in sources (!) src/mainboard, which you don't (normally!) have to modify at all. If it's true it "configures" mainboard like their false friend located in src/target and it used the same dialect code, that is over for the rest. Indeed, it is a default base file stuffed with critical values for your mainboard, thus if it is supported, it got to be all right here without your tweakings. I insist heavily, wrong statements in there will immediatly be paid back at "buildtarget stage" !

Equally, be careful with options you use, some of them just exclude each others. In some cases buildtarget script will stop in error while in some other buildtarget won't shout but results are undefined. **Every solution that is chosen thus must be activated in father Config.lb as well as in its various, dedicated or not, son Config.lbs in order to be effectively build.**

When buildtarget works correctly you should have something like this below.

```
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
```

That is clear enough, this has created standard make files, fallback and normal ROM make files. All's right, makefile is done but before launching a reccursive long Make, you can check out if options are finely reported

in Makefile.setting file. By the way, you can notice that crt0 and ldoption are also created here. The later file presents even simplier chosen options, compiling and linking ones.

Compilation

Change directory to target/_vendor_/_product_/_target_dir and launch "make". In fact, it will runs two differents makes : one for fallback mode and this other for normal mode. By the way, there is another way to get linking and compiling options displayed by typing "make echo". Here are some main make targets : Linuxbios.rom, echo, build_opt_tbl, Linuxbios, Linuxbios.strip, romcc, build rom, clean.

Remark : buildrom, romcc, build_opt_table are compiled upon compiler call represented via $HOSTCC variable. In other codes, it is represented via $CC variable. This is because Linuxbios is build not necessary on the same host/build machine. If it is (false :) then both of theses variables should have "gcc" value.

Targets references self and others in a pointer storm. Here are a overview of main dependencies :

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

Initobjects are primarily C source files, content files if I can write so. At the opposite, crt0.o and ldscript.ld are the application backbone. Initobjects are designated as such because of the "Object" dialect directive as we've seen in the end of upper part Generalites. Precompilation parameters are mostly located in theses InitObjects. Concretely, precompilation directives are present typically under a '#define' or '#ifdef' form. You can also find them back while compiling as gcc commandlines shows their '-D_XXX' strings which was listed in ldscript file. Here is an exhaustive compiling steps ordered list :

1. Linuxbios version file is compiled
2. romcc file is compiled
3. crt0.s file is assembled, failover.c and auto.c are compiled with romcc
4. initobjects objects are all compiled
5. initobjects references are gathered into Linuxbios.a
6. Linuxbios_c object is compiled with Initobject objects, Linuxbios.a and libgcc
7. Linuxbios_c exec. code file is done.
8. symbol map of this file is recovered into Linuxbios_c.map
9. linubios boot code is generated with ldscript.ld and crt0.o
10. Linuxbios objects are copied into Linuxbios_payload
11. rv2b upon compilation is applied to Linuxbios_payload by compression. New code
12. Linuxbios boot code is regenerated with ldscript.ld and crt0.o
13. Linuxbios symbol list is stored Linuxbios.map
14. Linuxbios objects are copied into Linuxbios.strip
15. payload is compiled if exists.
16. buildrom is compiled and called with Linuxbios.strip, Linuxbios.rom and payloa

Refer to target building dependencies synoptic flowchart to have a visual reference.

You now have a Linuxbios.rom file generated in target/_vendor_/_product_/_target_dir_ directory. This is a binary code adapted to your cpu and all your hardware specifications according to what was configured in mainboard/Config.lb. You can have a glance at ROM internal structure. Now, it can be flashed in a EEPROM

using a flashing solution.

Before swiching to this stage, maybe you haven't noticed a fundamental thing about Linuxbios compile process that we have just partly discovered. *Gcc has compiled a lot of C code file while other files have been processed by another special compiler Romcc. Why ? Let's penetrate Linuxbios bowels...*

Linuxbios advanced features

Another Linuxbios big feature is that it's admitted it must run as a standalone program. It mustn't have any external references except for previously considered config. files. That's why you can find a 'memset' code (memory mapping) in src/lib directory : in Linuxbios context it is impossible to use Linux kernel external memset code which supposes a BIOS service at this moment still not set up. I can't help exposing the simplicity of this code which BIOS will use several times such as in caching and shadowing itself. What else could be expected ?

```
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;
}
```

Nevertheless it is small it couldn't be called in real mode. Written in C, it requiers a stack and a heap which are defined only while protected mode (If you wish take a look to boot process flowchart to observe it).

All the same, there is no what a pity since memset needn't being called in real mode. But for instance, a cousin code such as RAM summing calculus code must be called in real mode (no heap, no stack). Indeed it must count how long RAM is to future memset mapping. Finally, Gcc shouldn't and can't compile this code, even if it's written in C.

Romcc is a compiler which generates from C, a peculiar byte code for ROM execution. This indicates no writing, no swapping in memory so no stack and heap mechanisms and endly no RAM needed to process. Romcc was bring with V2 version, before in V1 all 'real' codes was in assembly. Major advantage of C upon ASM: portability. RAM summing calculus is thus uniq for all motherboards. Romcc emulates stacks and heaps with CPU registers in processing swaps. Romcc generated code tends to be longer than if it was compiled with Gcc granted, but boot time shouldn't be that plundered.

At boot time, another advanced features is involved. Linuxbios creates tables which was (cleverly called Linuxbios tables. It resides in low memory and holds infos such as active motherboard, as BIOS shadow address field (in a RAM non volatile part) as well as other various Linuxbios parameters. *Natively, Linux kernel preserve this table at OS start up and there on. In source tree maybe can you find a utility called 'lxbios' which recovers and reads theses tables in order for instance to validate under OS a checksum.*

Linuxbios can thus use this non volatile memory part to store persistant data. Infos like boot time counter (CNT_BOOT) are defined in a structure called CMOS table. Still "visible" from OS, here is what it looks like.

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
387	1	?	?	?
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

Linuxbios CMOS tables

You can retrieve this file in sources mainboard/ directory. Cmos_xxx utilities are also available in source tree, it enables you to modify CMOS and Linuxbios options real time under OS. Useless to mention that you should know what you do in this...

Upgraded with V2, chained devices handling is another Linuxbios feature. Chips and devices are transparent while it is made lot easier than in V1 to call theses structured functions. On the same design, PCI data structures has been modified too comparing to V1 or Linux kernel ones. Take a look to what it now looks like:

Chip structure (static.c) :

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

Devices strcuture and its device pointers :

```
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            resources;
                struct bus                      link[MAX_LINKS];
                unsigned int            links;
                unsigned long           rom_adress;
                struct device_operation         *ops;
                struct chip                     *chip;
        }
```

Just from theses structures you can imagine the chainned organisation : **it is said to be PCI centric**.

At last, one of the most interesting feature is the ability to detach part of the Linuxbios payload onto a network machine. Indeed via DHCP or TFTP service you can retrieve a distant kernel. In case of clustering, this functionnaly is decisive. *It is also possible to imagine a distant BIOS update procedure with no fiction at all.* You will find fine details about part detaching and network booting in AMD64 Port Guide. Payload and boot principe are to come further in booting solution study.

You got a Linuxbios.rom file compiled, so what about flashing it in an EEPROM ? But if you didn't touch anything about ROM configuration you will only be able to burn in 256Kb chips (default). Maybe you already know what EEPROM and what software you are going to use. But in case you just don't, let's step forward to flash solution study.

4.2 Flashing solutions study

Flash or burning process which consits basically in writing byte code from a Linuxbios.rom file into a BIOS compound such as CMOS chip is the ultimate achievement. Because flashing solutions depends on chip compound type, theses are at the beginning of this study.

There on, you will remark that there are two ways to get a BIOS burnt : "cold and hot" flashing, plugging. Also I describe a hardware cold flashing and explain why I prefered a software hot flashing. The study is conclude by a scheme which describe what is flashed in Linuxbios ROM.

Virtually you can burn and program any memory based electronical compound. But some dependencies are mostly bound to deadlock. Indeed Linuxbios base solution can concretely identify some chips and I had to find in theses a chip that would be also identified by the flashing solution.

EEPROM chips, "flash" memories

They are plenty of EEPROMs that can host a byte code such as a BIOS. Let's consider the BIOS chip, a CMOS linked via a bus with its protocol. Grosso modo and according to previously described structures this is clearly designating a device. *Well, Linuxbios is only capable to read from internal bus (CPU-RAM exchange circuit).*

They are plenty of EEPROM on the market from different size, capacity, format. Everything is made clearer thanked to JEDEC standard (cf. IEEE standards site www.ieee.org) which is a regroupment of standards concerning chip casing and instruction set. **Well endly, Linuxbios is JEDEC compliant.**

To sum up, here is an overview available EEPROM :

- Common 128KB to 512KB EEPROM

For my experiments, I used SST chip such as SST39SF020A, as SST29EE20. I had also tested Winbond chip such as W49F002UP12B, as W29C020CP90B. I had tested a AMD AM29F0140B too. They had not same capacity but they were all of PLCC casing. There are many formats : PDIP or DIP chips have long rectangulars casings; PLCC chips have square casings (take a look at this scheme from SST datasheet); a TSOP chip is a mix from the two latter, it looks like a DIP rectangle chip whose pins would have been moved onto shorter edges. It is strongly recommended to refer to vendor datasheet to make sure that your chip is JEDEC compliant if you have a doubt.

- Disk On Chip (DOC) or Disk On Module (DOM) 1MB-8MB-16MB-32MB-64MB-etc..

  These are so large storage solution that you can even regroup BIOS and OS in the same casing. I have previously described in the Linuxbios kernel short howto how to configure a DOC. I have explained the point in "abilities" part, point "reduction" of the feasability study. You should be able to find some testimonies on this in Linuxbios maillist (cf. quick landmarks in Linuxbios maillist archive).
- Compact Flash (CF) 128MB to several GB

  Just like DOC or DOM it is even larger in terms of capacity. In there you can store your BIOS, your OS, but also your applications : a complete full features systems with all useful services. I have not tested anything about it but Linuxbios maillist archives surely contains valuable infos (cf. quick landmarks in Linuxbios maillist archive)

Cold/Hot flashing and plugging considerations

Primarily, I did try to cold flash and plug because hot flash and plug was said to be delicate to realize, even very risky to too dangereous. I say, you don't have to fear to be electrocuted, nor to loose your noze by smelting, or whatsoever. Truely this was really too exagerated... except if you are badly equipped to "operate" or still if you are utterly clumsy and you don't pay attention to what you do. But obviously you do if you read this g.

Hot way : It consists to unsolder original CMOS from the mainboard while running and to remplace it by a spacefree one. Then you burn it using a program. At last, you reboot and look at the result.

Cold way : It consists to unsolder original CMOS from the mainboard while its turn off. Generally, this requiers an external EEPROM programer card.

In order to do this carefuly and properly, be equipped with the adequate tweezers, remember chip notch location while you unsolder it, paid attention to use adapted pin converter if you may use any.At last, upon any doubts just download and read hardware specification properly. Your first burn must be CMOS save one !

Hardware and software

At first I try to do it the cold way and I came to use a Willem EEPROM Programmer v3.1 20-03-2002. This is a "starter" tool, an external device which consits in an epoxy card with a DIP EEPROM slot to setup with jumpers and switchs. The lonely two pages available documentation, if I dare call it like that, and DIP2PLCC adapter botch-up had wasted my time as well as for win32 card device driver which forces me to install a proprietary OS just for testing purposes. It was so confuse I just couldn't write to the chip, only have I managed to read it.

Therefore, I gave up cold way. Actually, it force an external device adjunction thus adding more intermedia, and slowness. It is pityful to setup with its out-of-date jumpers programing. It was hot that summer, I bent

dozing over my desk and dreamt I was punching cards at input. Seriously, if you have a relatively recent motherboard, prefer ease of the hot way.

Then I have tested Openbios project's DevBios tool (cf. Openbios site ref.) which reads and writes the hot way. In fact, it is a elegant solution which enables Linux to see CMOS as any other peripherical device such as a disk or a console. That is to say you can write your into it like that : "cat linuxbios.rom >/dev/Devbios". Or save your bios like this : "cat /dev/bios >save.rom". Symetric, simple this solution in a early release version presented nonetheless some powerful cons. First, this early version (first or second "official" release) support a little number of different chips. Second, it couln't write to CMOS because of my C3 Ezra cpu which is not supported by this Devbios version. I had to forget about Devbios too.

The most simple way was what I didn't remark in Linuxbios source tree, a chip flasher. It is a simple to use flash_rom tool as this below overview attest it and it supports most common chips.

```
[root@ge utils]# ./flash_rom
usage: ./flash_rom [-rwv] [-c chipname] [file]
-r : reads flash and save into file
-w : write file into flash (default when a file is specified)
-v : verify flash according to file
-c : probe only for specified flash chip
 If no file is specified, then all that happens
 is that flash info is dumped
```

Another solution is available but I had no time studing it. It is "lbflash" command (cf. cmos commands).

Linuxbios ROM structure.

To conclude this study on a graphical resource, you can take a look to Linuxbios ROM memory layout. Lower addresses are at top.

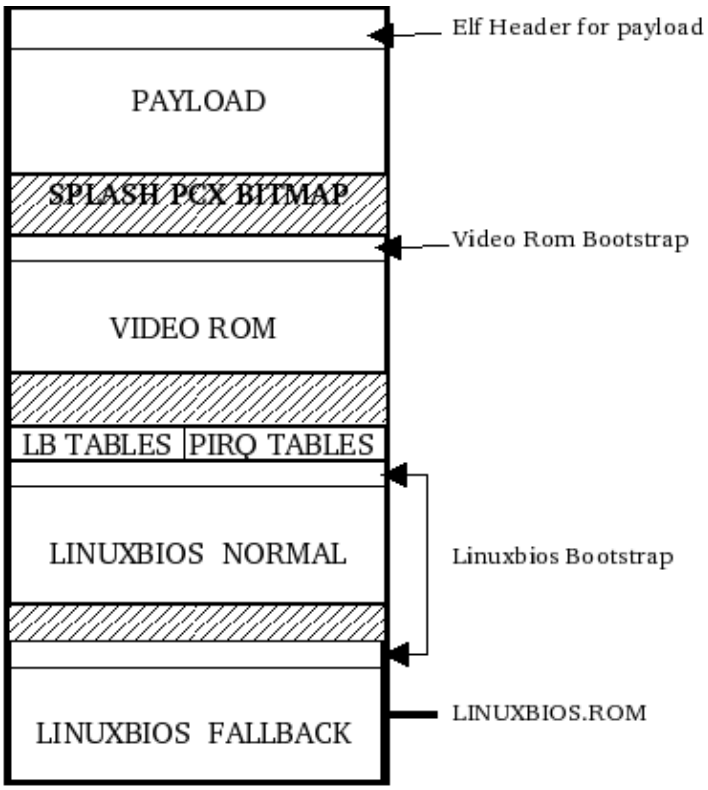


Fig E

You have achieved to configure, build and flash your free bios Linuxbios. Although I just don't know whether it will be working or not. What you need is a easy to setup and to use logging facility. This is exactly what it is about into next part.

4.3 Logging solutions study

A successful built and flashed bios ROM should allow you to think it will work perfectly. But while no display solution has been set up, neither VGA nor BTEXT mode, sole remaining solution is to checkout debug strings with the resources and tools part described experiment appareil. By the way, also check out your cable with cable pin description.

If DEBUG option and SERIAL_CONSOLE have properly been set in Config.lb, if cable is properly made up, if transmission protocol is correctly set up, normally debug messages must been sent to ttyS0 serial port. To fetch them back I studied "minicom" which is a program for modem to emit and receive packets on serial port. After several tries and bad protocol setting up, minicom appeared to be too much of a pain to be used for this study propose. Menu commands have to be triggered by several CTRL holding while multiple key hitting. Menu bars are badly refreshed, etc.. This is a recommanded tool but believe me, in fact, it does badly more that it is requiered.

In that case, what does it needs specificly ? Just to retrieve theses strings from ttySO and to display it to console. I've made up a command called Hear! bash script, which is simple to use and to configure. Let's have a synoptis :

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

Logging Procedure : Howto to proceed with testing and debugging your Linuxbios:

First of all, set up your appareil as evocated just beneath. Linuxbios build machine is now fully qualified bacause you had inserted your Linuxbios CMOS chip into freed CMOS slot while flash procedure.

Second, on your other machine you just have to start Hear script pending.

Endly, just reboot Linuxbios build machine and gather debug strings into Hear logfile.

Here is typical Linuxbios startup truncatured logfile created with Hear!

```
**************************************************
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.$
```

Still pending there is two ways of closing script : classical by sending a end-of-file (for instance send /dev/null) or interactivly by pressing CTRL+C. At the end, Hear.sh propose to record your commentars, best is to add there experiments conditions, specific tested configuration options, some remarks/feelings. It will be placed at end of logfile as well as boot time measure.

Here is the end of this same logfile

```
114      found PCI IDE controller 1106:0571 prog_if=0x8f$
115      primary channel: native PCI mode$
116      Waiting for ide0 to become ready for reset... ok$
117      Testing for hda$
118      Probing for hda$
119      LBA mode, sectors=40021632$
120      Init device params... ok$
121      hda: LBA 20GB: Maxtor 32049U3                        $
122      Partition 1 start 63 length 2040192$
123      Unknown filesystem type$
124      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
```

Results and Remarks :

Indicated '13.67' seconds boot time is actually a relatively long boot time. But remember that in debug mode, tests and displays are very time consuming. Plus in normal mode, you should not need any stream redirection which is also slowed by protocol's baud rate. The 'unknow filesystem' indicates that my payload (none = elfboot) don't accept hda part1 filesystem which is an ext3 partition.

You surely have to multiply tests and reconfiguration. For instance, in beneath exposed example my filesystem is not supported and I had to modify partly my payload and thus my kernel configuration (ext3 native support), to buid it again. I also did try to modify Linuxbios configuration to activated any ext3 option, but I couldn't find any. Maybe it is not just a payload matter then I had to study other boot solution.

4.4 Boot solutions study

This study will clarify a very important notion : the payload. It is the pure useful load, typically the OS. Although Linuxbios ROM layout represents it at top, payload is executed at the end of bios start up as logfile tail can attest. In previous bios boot example , nevertheless there wasn't any payload, bios was allthesame instructed to boot elf payload (default is ELFboot) on hda part. 1.

This clumsy permissiveness design forces to explain what's in a name to be more precise what is to boot ? *A bootloader is a program which boot a payload that is an OS. A bootloader which boots an application program should be just called a loader.*

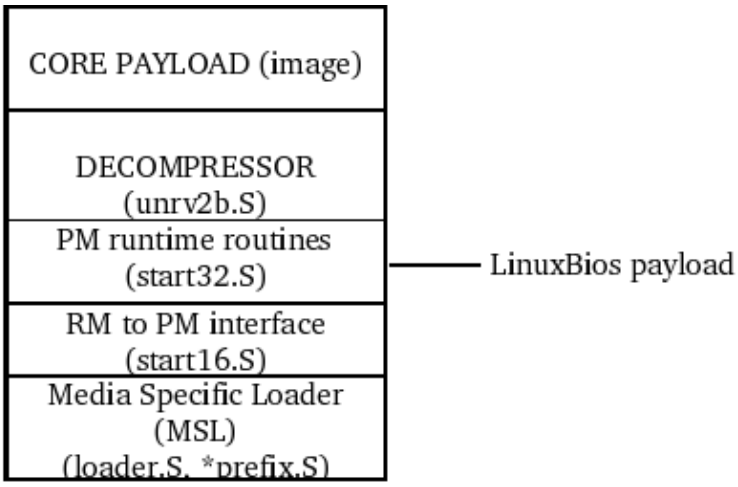
I take a concrete case to explain further. For instance, to start with loading a simple payload, you can attach a funny program which would only display "Hello tragic world !" and stops. In this case, you will load from i.e hda1 a program whose format must be understood by the loader. Linuxbios no payload choice induce the activation of default bootloader or loader ELFboot which boots ELF format.

Now if you want to boot an entiere OS, previous loader must mutate into a bootloader such as FILO. But this complex distinction will become clearer further on. Anyway, simplest is to remember both bootloaders and loaders are boot solutions.

There are two conditions limitating payload program constitution. First, payload parts must fit in free and adjacent memory spaces. Second, of course, its bootloader or loader must be payload's format compatible and it must be link statically (no external dependancies, standalone).

Here is represented Linuxbios complete payload layout:

Fig F



The Media Specific Layer (MSL) is core payload's decoder. Decompressor code exists if core payload is compressed (well, Linux kernel is). RM to PM interface code handles real mode to protected mode interface. For instance, if you want LILO as bootloader, payload will consist in liloprefix.S MSL (cf. ALDO source tree). Make a do-it-yourself payload takes time but Linuxbios offers supports for various ones.

There are two different boot solutions : Linuxbios native one and external ones. According to this, either a boot solution is simple and bios probes first 8KB to look for a ELF image, either it is said advanced and it could even analyse filesystem to check for specified image. Therefore, concerning simple boot solution image file must resides at ROM beginning.

Now let's list how you could design a boot architecture with theses boot solutions. Note that the two first have been internalized :

- Linuxbios ELFboot (simple)

  This is the default base solution activated if no other is and whether a payload is actually specified or not. It can boot from IDE devices an ELF format payload. Typically it is a Linux kernel. It can also be any other program which if needed have to be A.OUT to ELF reformatted. In order to make an ELF kernel payload from your previously prepared and compiled kernel (cf. howto part from a kernel built for Linuxbios part) you can refer to howto make a elf image appendix.
- Linuxbios FILO (advanced)

  This prior external solution gained success quickly thus it has been internalized. Nonetheless infos I had gathered was rather related to external FILO version. Therefore below featuring functions can belong either to one, to the other or even both of them. It can :

  - ♦ Boots from IDE hard disks and Cdroms disks.
  - ♦ Boots standard ROMs linked to internal bus.
  - ♦ Boots from raw devices with a specific offset such as CF and DOC/DOM
  - ♦ Supports theses filesystems : ext2, fat, jffs, reiserfs, xfs, iso9660
  - ♦ Supports theses payload formats : ELF, zImage, bzImage
  - ♦ Supports natively VGA console, keyboard and serial port

  This solution has extendly been tested. Written in C code it is of course completely standalone. I had chosen this solution from my Linuxbios, it seems that it has to remplace ELFboot. I had chosen this solution from my Linuxbios it seems that it has to remplace ELFboot.
- USBboot (simple ?)

It is a recent solution. It would add a USB support for Linuxbios bios. It would be a ELFboot based code solution. More infos are to be retrieved in Linuxbios maillist and archives.

- Etherboot (simple but some advanced versions would exist)

  Etherboot is a stable and famous project (cf. ref on Etherboot project site). It enables to boot a payload from a network machine. On one hand, it builds a small network bootstrap according to network controler chip. Actually, Linuxbios needs to be told to integrate/load it. On the other hand, a TFTP server have to be ready to respond and deliver the kernel upon client request.

  It is also possible to Ether-boot with an IDE patch. Indeed, with the same solution, gain resides in enabling IDE boot while keeping the ability to network boot.
- BOCHS and ADLO (?)

  This solution used to be very implemented more than it is now. ADLO is based upon BOCHS, a kind of emulator with abstracts motherboard hardware. It offers you to build entierely your own payload. It also provides a VGA BIOS implementation which I used to test a Linuxbios display prototype. It seems that this solution is left abandoned. To my mind, it is worthy and you should check out V1 sourcetree.

When you finish testing bootloading mechanisms with ELFboot or FILO, it might not be enough if you want to sketch an advanced boot architecture. Linuxbios ables you to "mount" a complex one provided you do some more coding or some enhancements. I didn't gone that far in, theses are proposals or examples considering some Linuxbios forum posts about boot architecture successfull tests. Theses are called bootchains.

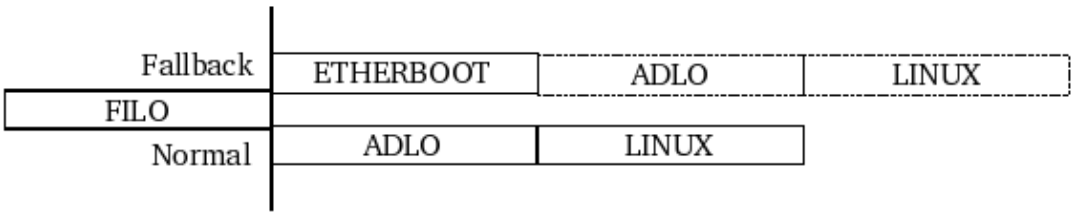
Here are some differents bootchains, from simplest to more complex ones.

Simpliest : Linuxbios uses ELFboot to load kernel from hard disk. *

Simplicity/Adaptativty compromise : Linuxbios uses Filo to load a kernel from hard disk or from other IDE device such as a CDROM. *

Linuxbios uses ALDO as a booter. ALDO then loads Lilo as bootloader which will load Linux kernel. *

An advanced architecture : Etherboot is set onto Linuxbios. It download from a server ADLO booter which will load Lilo bootloader. If Etherboot has been IDE patched, it becomes a local booter. It is not more efficient than previous architecture but it is more adaptative.



Another one, finest of the finest : One branch retrieve distant payload as an update while the other one just exploit that updated payload.

(*) : Theses are trusted solutions.

Core payload Local payload Distant payload

Remark : It is obvious ROM/flash memory isn't at will expendable. Therefore in theses boot chains, could space may become a problem notably for longest ones ? Not at all because never a currently running payload ever gives hand back to previous one (which has launched it). In other words, a payload has no execution return, it is useless, isn't it ? Therefore theorically, current running payload can occupy its previous launcher memory, that is to say that theses multibooters/bootloader architectures are not memory spendthrift. If practically feasable, this capacity would although need a strong code enhancement to enable this.

Boot solutions are one of the most active effort in Linuxbios project. But user's boot need are versatile, so remind to keep it as flexible as possible. Keep as simple as possible while not killing Linuxbios fallback mechanism. To start with, best is to try simplest chains. If you make your boot solution to work with enough resilience (flexible and safe mostly) you can consider your free, open, Linuxbios functionnal. Indeed when Linux kernel will take hand over it should (re)initialize and in fact set your display. But you have to admit that before this stage, this wouldn't be considered that resilient without any display configured. Yes I guess what you could be thinking "if all is fine, who cares to see that bios dialog at startup and to have it displayed ?". I do, and you also when you will have some bios prompt or

troubles. Anyway, do you really believe you would like to keep and load yourself down with such a strange wired appareil ? Indeed no, you don't. So let's see how to get a valid diplay solution.

4.5 Display solutions study

Experiments don't need any graphic display to get Linuxbios working and to attest it works. But it is not possible in "an product approch", in other words, your users will want a graphic display. You know that Linux initializes a console and sets up a terminal, it also sets up the VGA driver enabling a graphic console and a graphic mode terminal.

Noticing this matter of fact, Linuxbios developpers thus implemented minimal code to initialize graphic card registers into the chipset. Roughly they send a ready-to-display byte stream to the card according to its chipset. It has been realized without a lot of code yet it isn't that simple because they had to know precisely that chipset as well as its specific initialization. *Theses infos are either given by the card manufacturer either it is not. If it is then all is right.*

If theses infos are declared secret, it clearly set a barrier on designing a video code. Anyway, who ever said you need to do this ? No one and there is always something that's available, the original video code stored into the graphic card ROM. It is called VGABIOS. *Simply, it will be copied into Linuxbios ROM (cf. Linuxbios ROM layout).*

All the same, a problem remains : theses intializations primarily lie on 16 bits mode BIOS calls whilst Linuxbios is fully 32 bits since a while up to there. To attest this is true, just consider that the first thing to be displayed when a system is turned on is the graphic card name and its manufacturer. This is made possible with 16 bits codes for at this early stage RAM has not been summed up and analysed. (cf. check out boot process to see by yourself).

To make it through Linuxbios developpers have taken the last resort : to use an external emulator x86emu. Sure, this is not a panacea but until Linuxbios gains more support from manufacturers it is efficient. Anyway at Linux kernel graphic driver set up, it will be replaced by vesa (vesafb) or by a frame buffer device (fbdev) or even better a X server (X11). Theses drivers just need chipset to respond to managed I/O.

Stack showing different software layers to handle graphics in Linuxbios :

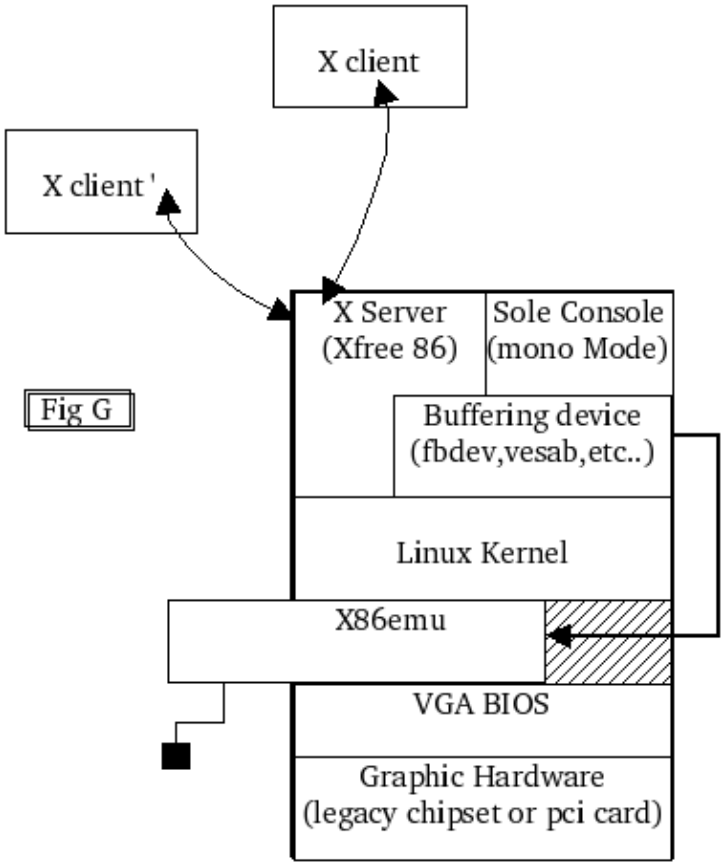


Fig G

Of course, nothing stops you from not desactivating SERIAL_DEBUG while building your next Linuxbios. I let you know that it won't prevent this VGA solution to work anyway. Linuxbios source tree provides testbios to check and validate your video ROM. Then you can test direct writing to stress I/O frame buffers and validate your X server. Endly, you can test graphic modes. Theses two tests I've made quickly are graphic tests scripts.

To conclude this section and all the study part I would say that I've reveleaved a lot of Linuxbios globally and peculiarly. This study which is very Linux oriented and very server-client oriented still stays general. As I've stated in the disclaimer remember it is a specified Linuxbios project, an instance if you prefer.

I think you now have all the keys to even develop from Linuxbios project if you have coding knowledges. Linuxbios project like other FLOSS communities is always pleased to welcome all contributors.

Next section is a pure HOWTO configure and combine studied solutions for VIA EPIA-M motherboard.

Next Previous Contents

5. HOWTO VIA EPIA-M motherboard case

The objective of this part is to describe how to boot a kernel with a Linuxbios. In order to realize it are deployed Linuxbios studied solutions. Mainly, it is and howto for developpers to deploy quickly a Linuxbios prototype using theses "solutions". Thus it condenses at most previous work behind this document. If you get yourself directly parachuted in here, consider reading some part of the study if you are in trouble.

It was a choice to make the distinction between study and all that is not what is you're going to read g, Study explains while describing functionnalities, while demonstrating how it works, while theorically conjecturing about. This pure HOWTO implement what is bound to this study and well really sets practically what I was enable to achieve onto VIA EPIA-M based machine. (Without any other conjecture) Therefore alternatively this part also express this gap and go further to put Mangrove Linuxbios on a derivated development branch.

5.1 How to get started

Primarily it is vital to gather specifications of your build machine and your target machine hardware as I did (cf. my build machine). Launch a `lspci -vv > myconfig` to get your config. and dump it to file because if you go for help in forum, you could then send it to basically express what type of config. you have.

Check out how up all your hardware is supported, look for motherboard support page on Linuxbios project site (cf. ref. Linuxbios project). If you have a VIA EPIA-M 1000 note that it is almost fully supported. For each type of solutions, it is another matter so try to early answer to that Skakespearian "supported or not" question, scour the site forum. (cf. landmarks to forum maillist archive). If not also I let you know you always could then realize a port from a supported motherboard or/and ask for help to do it.

Check out Project's tools and resources part for some details.

Finally get your development/experiement platform up to date in order to prevent any versionning trouble :

o There you'll find the toolchain to build a Linux kernel

o There you'll find the toolchain to build a Linuxbios.

Personally I used a Red Hat 9.0 because it complies with all the above referenced requierements. Note that by the time I worked on Linuxbios, I found not ext3 filesystem support, so checkout if ext3 is supported and for what boot solutions it is or consider booting on an ext2.

This consideration is bound to how you prepare your Linux kernel payload. Also you should know what is your Linuxbios specific needs. Check out theses innerdoc links to draw out yours :

o Know what are Linuxbios pros and cons, there in feasability section of Project part.

o Know what is your boot wish, there in boot solution study.

Ok let's go. Edit your mainboard/via/epia (main) Config.lb. Tweak everything like you want, take a glance to following option explanations

```
DEBUG=1/0
      Activates debugging mode.

TTYS0_BAUD={124800,76800,48000,28800,19200,9600,1200,0}
      Transmission speed on the (crosswired) RS232 cable DB9 pins. Faster is best...

LOGLEVEL=number(dflt:6)
      Levels are described in arch/i386/include/printk.h.
      1 : EMERG       4 : ERR         7 : INFO
      2 : ALERT       5 : WARNING     8 : DEBUG
      3 : CRIT        6 : NOTICE      9 : SPEW        10 : everything

MAX_REBOOT_CNT=number(dflt:3)
      Number of boot tries

CONFIG_SERIAL_POST=1/0
      Activates POST debuging if you have a POST compliant device...
      or not anyway you'll have a hex decimal POST indication telling what stage of
      the boot sequence you're at.

CONFIG_COMPRESS=1/0
      Indicates to the booter/bootloader that ROM image is compress or not.

HAVE_HARD_RESET=1 (better not to touch this)
      Hot reset handling.

i686=1 or i586=1
      Set them to 1. It's architecture specific registers parameters.

CPU_FIXUP=1
      Usefull in some case (which one ?). Patch for CPU, just activate it
      if it wrecks everything turn it off, if it still wrecks all plus your
      head... go have some aspirin.

INTEL_PPRO_MTRR=1
      Graphic MTRR registers.

HAVE_OPTION_TABLE=1
      Part of the Linuxbios table.

HAVE_FALLBACK_MODE=1
      -"The J option"
      Unset it if you really believe there are spiders on Mars.

ROM_SIZE=number*1024(dflt:256)
      BIOS compound/CMOS chip you have to store your Linuxbios (ROM_SIZE value in bytes)

FALLBACK_SIZE=number*1024(dflt:192)
      Fallback mode size in Linuxbios ROM (number value is expressed in bytes)

_ROMBASE=0x0004000
      Begin of Linuxbios code into the CMOS.

CONFIG_IDE=1
      Activates boot on IDE.

CONFIG_FS_STREAM=1
      Open a stream onto the filesystem. By default also activates builtin booter/bootloader FI

CONFIG_ROM_STREAM=0
      Open a strean onto a linear memory structure. (ROMBOOT)
```

```
AUTOBOOT_CMDLINE="hda1:/vmlinz root=/dev/hda3  console=tty0,ttS0"
       Booter/Bootloader boot command line and parameter passing to kernel.
       IF NO STREAM IS OPENED, Native buildin booter/bootloader ELFBOOT is in charge.

CONFIG_LEGACY_VGABIOS=1
       Activates x86emu emulation for VGABIOS.

VGABIOS_START=0xfffc0000
       Pointer on the VGA Bios code in Linuxbios ROM.

CONFIG_CONSOLE_SERIAL8250=1
       Relay debugging strings to serial port.

CONFIG_CONSOLE_BTEXT=1
       Active a uniq console, the 'sole console'.

CONFIG_CONSOLE_VGA=1
       Active a graphical console supporting X11.
```

Equally, look at the *romimage* structure

```
romimage 'fallback'
       Section title : fallback or normal ROM image.

  USE_FALLBACK_IMAGE=1
       Fallback ROM in case of normal feinting.

  ROM_IMAGE_SIZE=0x10000
       Taille de la rom de secours.

  mainboard via/epia
       Mainboard which it is supposed to boot

  payload null
       Payload to boot with. Null indicates nothing of course, or
       rather that it is an internal solution ELFBOOT or FILO (elf format)

end
```

Now properly launch buildtarget on via/epia. You must have something pretty much like this.

```
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
```

Check out Makefile.settings file or Makefile.rules... err check out makefile config files to compare with this :

```
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
```

Unless "you know what you're doing" (Pat.:) if you don't have this values... umm... figure out why, because it is going to waste your time compiling before reaching an error.

Then it's time to launch make, type it. You have your linuxbios.rom, so get it saved somewhere and prepare to flash. Replace with care, unsolder the bios chip with a adapted tweezer (yes, when your mainboard is online ! If you're not feeling confident with this read flash solution study.)

Place your new erased chip in (your target machine CMOS slot of course). Find the flash_rom utility into the Linuxbios source tree. Basically you need to launch a `flash_rom linuxbios.rom`. If your chip is supported this utility will find it. If not, change tool... still not change compound... still not ? you must be kidding... If you want to erase your chip, there must be a switch to do this... Anyway, consider creating a zero filled file to flash with, use a `dd if=/dev/zero of=zerofile bs=1024 count=rom_size_in_KB` command to create that file.

```
Programming Page: address: 0x0003f000
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
All right.
```

Get your cable set properly, between host and target. Launch the hear script. Do not forget to have in the same directory, the digest script also. Ok, if you feel low on bash scripts, you can still use minicom. But in this case, for this target, this is really too much. Maybe you're using Linuxbios for embedded devices, true guess ? Well then embedded means least "structure", so just size your soft properly and stick to this.

Well that's it, your bios is over. Congratulations ! And what about adding a VGA BIOS to your BIOS in order to take advantage of Config.lb activated parameters and of a full-featured modern bios !

5.2 How to go (a bit) further (in with the VGA display on target)

Having a target standalone display is far from being superficial. So you can wonder why I come to this at the end. Look, if you want your Linuxbios to run an appliance, a black box, for example a machine such as a router do you really need any sort of display, I mean when it'll run stable ? Do you need a VGA display ? Remember syslog facility is able to redirect a logging facility to a remote host.

Ok, you know what you're doing anyway I was just proposing...Right, you need to do some handcraft in here.

Recent Linuxbios developments must been hiding this, check for updates. All the same, what fellows surely stands still behind.

You need first to recover original video bios (check display solutions study to know why). It is maximum 64KB long, EPIA's one is 44KB long. Let's assume it begins in 0xC0000 and ends in 0xD0000, because it is often true (check out with a hexdump tool if it is coherent). The beginning has a ELF header which size is 0x01000. You'll have to cut from 0xC1000 because that header will be rebuilt. As a result, pure VGABIOS code is 0xC1000-0xD1000 that is to say from 790528th byte to the 856064th byte (clue : conversion powered by `bc`)

Kcore is a peculiar entry in the /proc filesystem which is by default compiled in ELF format in the Red Hat 9 kernel. Check if you have something relevant by doing something like : `readelf -e /proc/kcore` . Kcore express all the physical mapped memory by the kernel, thus it fluctuates in "real-time". The next command with fetch back the VGABIOS in kcore.

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

Resulting video.bin is your original VGABIOS. Heh, remember this better be done with the original BIOS plugged onto target machine, EPIA mainboard CMOS slot. Then you could safely test your video.bin file with *testbios* from V1/util/vgabios/testbios. To do so, you'll need a proper graphical framebuffer driver to get insmoded.

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

I've hacked up in speed some graphical test scripts to test video display, since they actually stress framebuffer and framebuffer driver actually do VGABIOS calls, it should somehow test VGABIOS. If you want to measure a more specific performance, you'll consider finding a real tool to do so.

Endly you can concatenante your VGABIOS with linuxbios.rom file after the rom making stage, look at bios's ROM structure to see where to operate precisely. So you basically needs to do this : `cat linuxbios.rom video.bin >linuxbios`

Flash the linuxbios resulting file as explained in the upper howto section. If you have something stable, remember disabling all debug features and compiling a new Linuxbios...theses slown down boot time. If you spared so much space that it is just remaining finally too much space, know that you can add a bootsplash

screen bitmap at boot startup !

Pardon ? Yes true, VGABIOS is a binary image copy. So if you want a compiled from source VGABIOS note that OpenGraphic project development aims towards provinding a generic graphic support.

There you have a Linuxbios CMOS running ? well you know it's not over. You're happy-happy yet still wondering when or why, that super feature you dream of, will be implemented and how it will be. Then think in 3d : derivate, deploy, develop !

Next Previous Contents

Next Previous Contents

6. Conclusion

Can you realize what happened ? You have gained control over the lowest boot code of your machine, that "sacred-closed" thing became free and you even can keep on developping. True, that just a matter of specifing to your hardware needs, boot needs, power and security needs, a matter of thinking of your booting architecture : just matters you couldn't get your nose in before ! Yet already, you imagine of a lot of boot scripting possibilities. *Boot A if B fails, until C boots. Upon 3 if C fails, bios-netupdate C if A not boot. Can theses letters be machines or programs ?* **This sounds like a beginning of IO streams of BIOS, isn't it ?**

Next Previous Contents

7. Glossary

- Cluster : Grape(s) of heterogeneous machines which runs enslaved beneath a Master machine,a server, centralizing memory, access requests, etc.. Use in grid computation or in massive calculsus.
- Flash : Action aiming to write data into a chip such as EPROM or EEPROM (Electronic (Erasable) Programmable Read Only Memory). In this project all of them are EEPROM.
- BIOS : BIOS stands for Basic Input/Output System. Kind of minisystem before the Operating System (OS).
- CMOS : EEPROM/EPROM Chip that stores the BIOS binary code.
- Cross Compiling : Useful notably when resources are low on a machine (constraint milieu, embedded device) this technic enables another machine to 'tune' its parameters (such as memory quantity, cpu type and so on) to the former and to emulate its behavior. As a result and because the later have higher capacities, compiling goes faster and so does tests and validation.
- Shadow RAM : Part of the RAM memory reserved by the BIOS to copy itself as a shadow mainly in order to be run and accessed more quickly.
- CHS geometry : Cynlinders/Heads/Sectors 3D mapping of a harddisk. The reason why it is still used is now historical.Now Abstracted by LBA (Logical/Large Block Access) enabling more than 528Mb disks mapping.
- RAM : (Random Access Memory) Essential memory that stores info for a session. Dumped upon reset button-push (reboot) nowadays RAM's content is partly auto-sustaintive (not dumped by simple reset)
- ACPI : Advanced Configuration of Power Managment Interface. Enables the BIOS to shut down a device after a certain period. Also let the OS user monitors and tweaks theses settings.
- Table descriptors : GDT stands for Global Descriptor Table, LDT stands for Local Descriptor Table, IDT for Interrupt Descriptor Table.
- DMA Channels : Direct Memory Assignment. Programmed Input/Output (PIO) is a serious concurrent to this mode.
- IRQ : Interrupt ReQuests Handlers triggers an normal code branch interruption. A bit deprecated since PnP norm.
- Assembly code : Alpha,Aleph code, the first generation code.
- C code : second generation code (B code has never existed)
- MBR/PBR : Master Boot Record, Partition Boot Record holds 512 bytes (a standard sector).
- MTRR : Memory Type Range Register, grosso modo theses are registers reserved to initialize dialog between PCI and AGP regarding to the video card. Their correct intialization boots performance notably.
- Failsafe, Fallback : Failsafe is a stable emergency mode which enables you to repair the system to work back in normal mode. Fallback is Linuxbios stable emergency BIOS image stored in a ROM which is loaded on main BIOS image error.
- Kernel Panic, Fatal Error : Major system error, unavoidable, uncurable, often unpredictable. In short a disappointment.
- Boot Process : System Initialization sequence is standard IEEE 1275-1994.
- Boot Loader : Code residing in MBR block that reads the 55AAH signature and loads OS.
- LILO : LInux LOader.
- GRUB : GRand Unified Bootloader.
- VFS : Virtual FileSystem. Abstracted device driver layer for file systems.
- POST : Power On Self Test is mostly a boot sequence designation standard integrated in PCI card hardware or in some motherboard itself. It gives a good review but prefer Linuxbios debug string transit via serial cable.

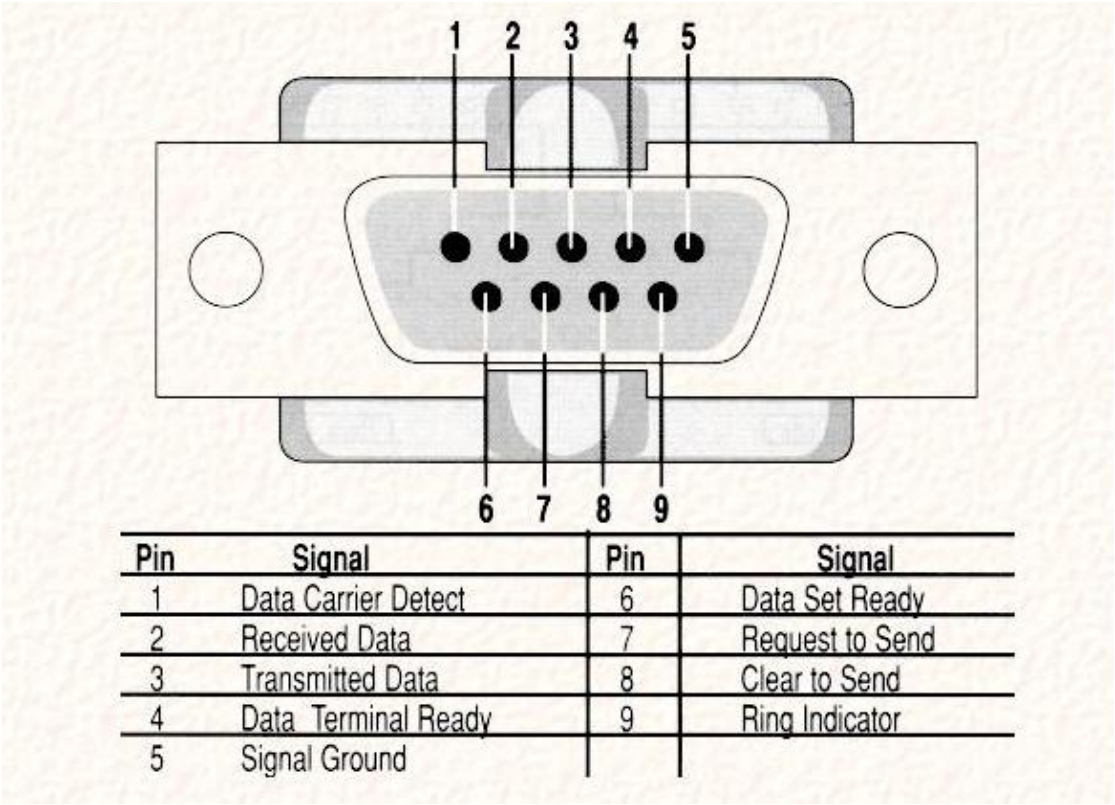
- RamDisk : Coming with RAMFS, this is a standard file systems that resides in RAM memory instead of disk memory. It is a close friend of TMPFS (TeMPorary FileSystem).
- Deamon : "background" running program. Listenning to client connection and registration to a given service. It relays service offer and request to client connection.
- VGA : Video Graphic Array. SVGA stands for Super VGA.

Next Previous Contents

8. Appendix

8.1 Serial Cable pin configuration as null cable

Here is the 9 pin(DB9) RS232 interface. Null cable is also called "crossed cable"



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		

Null cable plugs with their pin numbers

A Plug	B plug
2 and 3	3 and 2
4 and 6+1	6+1 and 4
5	5
7 and 8	8 and 7

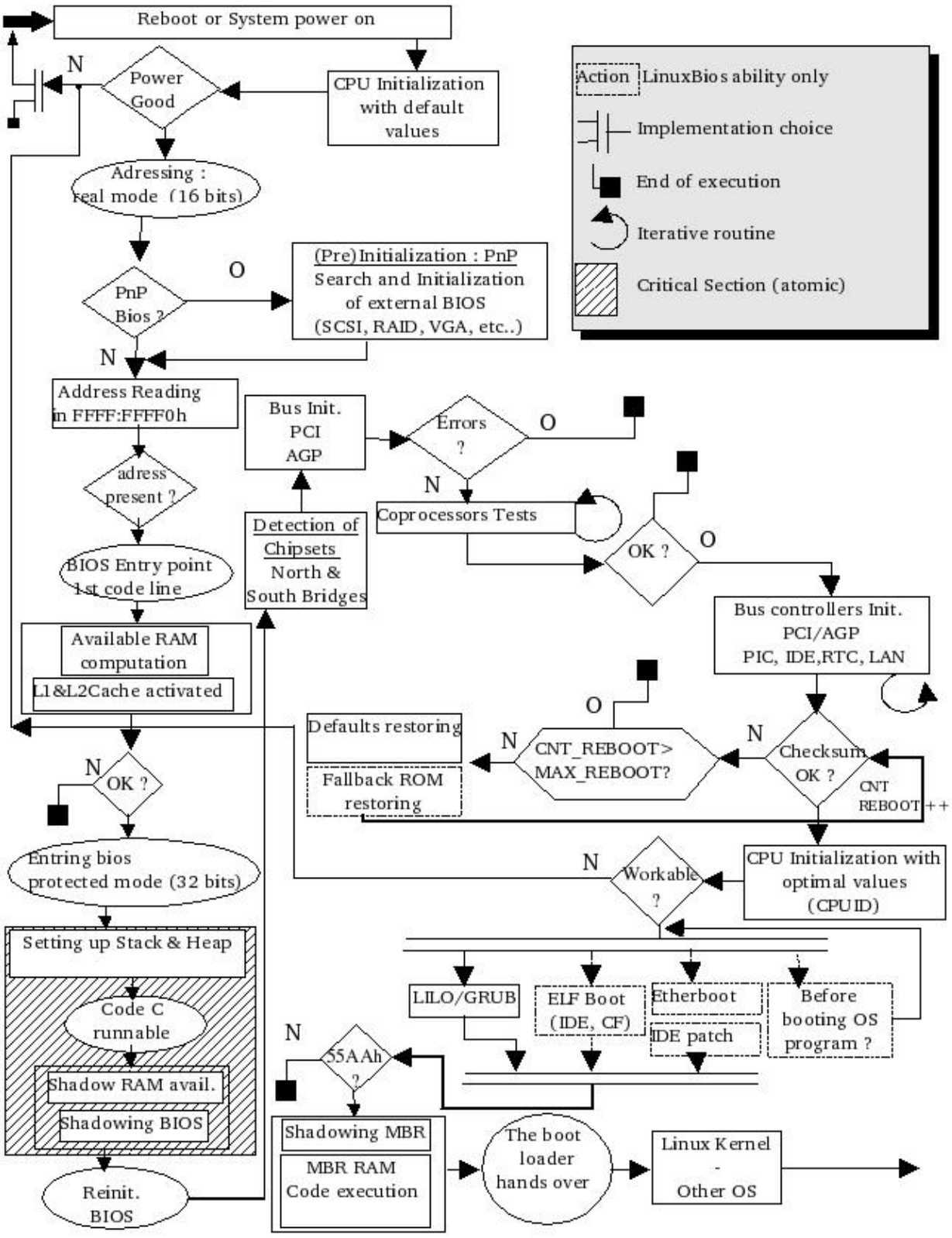
8.2 My build machine hardware specifications

- MotherBoard: VIA EPIA M(Mini-ITX) 17cm²
- CPU : VIA C3/EDEN EBGA Processor, C3 1Ghz

- Chipset : VIA CLE266 NorthBridge VT8235 SouthBridge
- PCI,ISA : VT8601, VT8231
- IDE : VT82C586 (2x UltraDMA 133/100/66 Connectors)
- Memory : 1 DDR266 DIMM socket up to 1GB
- VGA : AGP,MPEG Accelarator, Trident CyberBlade/i1
- Extension : 1 PCI
- LAN : VIA VT6103 10/100 Base-T Ethernet PHY
- Audio : VIA VT1616 6 channels AC'97 Codec
- Misc. : firewire and USB support

8.3 Bios boot sequence

This is a synpotic flowchart resuming the boot process of Linuxbios.



8.4 Kernel help! recovering procedure

In probable case you crashed your bootloader, i.e lilo (the std bootloader) freeze on "LI" display. A crashed kernel is mostly a problem if not other sane was prepared, but it is obviously not your case, isn't it ? The goal

is to reach a sane command line, this a type a 'lilo' command for example. Here is a procedure to recover your MBR when you don't have a floppy drive and you just have a bootable Linux CD such as Red Hat Linux.

Objectives : Get a shell, mount partitions, disk integrity checkout, have a recovering 'lilo' command.

Type *linux rescue* at redhat type install CD boot menu. A menu invites you to configure language and keyboard. Then another menu comes, do the same for ethernet device eth0 and eventually setup ftp if you need to recover some config. files (ie rc.sysint or lilo.conf) from network. After Ananconda starts, access via CTRL+ALT+1 to first virtual terminal to check out if your bootable device /dev/hdaxx is present, otherwise come back via CTRL+ALT+7 to let pass some menus before coming back again to first terminal.

All is right ? Ok, now remount your partition in read write mode with *mount -no remount,rw /*. Mount root partition i.e hda3 like this *mkdir /hda* then *mount /dev/hda3 /hda3*. Do same for other partitions, especially those where /etc /sbin resides. Chroot yourself : *chroot / /bin/sh*. If your keyboard is buggy type a *loadkeys en* to reset it to your language (here english). At last, you can edit your lilo.conf and launch a */sbin/lilo* before exiting for maintenance mode with CTRL+D. Synchronise partitions with *sync* and reboot *reboot*.

8.5 Kernel minimal toolchain requierement

Compound	Min. version requiered	Current version
GNU C	2.91.66 "egcs1.1.2" *	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

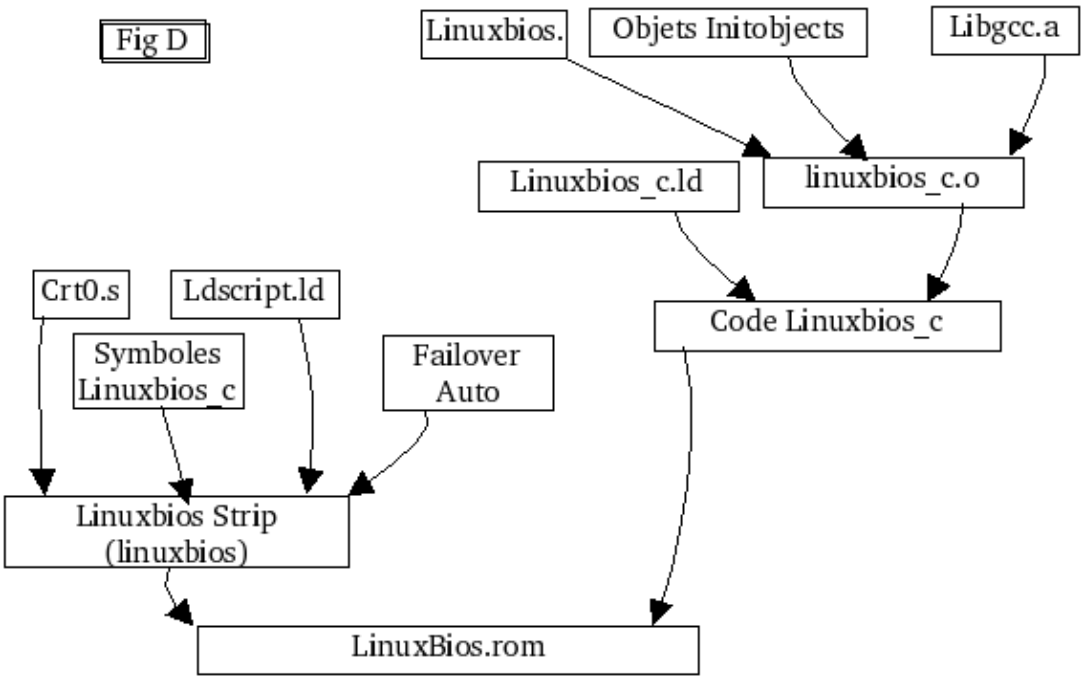
Software toolchain minimal requierements to build a Linuxbios kernel

* Note : "egcs 1.1.2" is the sole official version recommended to build a stable kernel.

8.6 Linuxbios minimal toolchain requierement

Compound	Min. version requiered	Current version
GNU C	3.2.2	gcc --version
GNU Make	3.77	make --version
Binutils	2.14.90	ld -v
CVS	1.11.6	cvs --version
Python	2.3	python -V

8.7 Targets building dependencies synoptic flowchart



8.8 Linuxbios tables, CMOS Table utilities overview

```
        + Restores a Rom image +
bios_restore /dev/mtd_x romimage [--vendor VENDOR] [--part PART]

        + Save Bios content in a Rom image file +
bios_save    /dev/mtd_x romimage [--vendor VENDOR] [--part PART]

        + A utility to edit CMOS 'RAM settings' configuration
cmos_util [-d|-u|-s|-h]   [--image file]      [--outpout_image file]
          [--layout file] [--new_layout file] [settings file]


lxbios  + Reads and writes Linuxbios table parameters

lbflash + a flash solution dedicated to MTD DOC chip for Linuxbios
          also make a checkout with motherboard type.
```

8.9 Logging and communication scripts

Hear script which retrieves debug strings.

```
#!/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 script which is used by Hear! to have a nice human-readable display

```
#!/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
```

8.10 SST SST39SF series EEPROM chip of PLCC casing

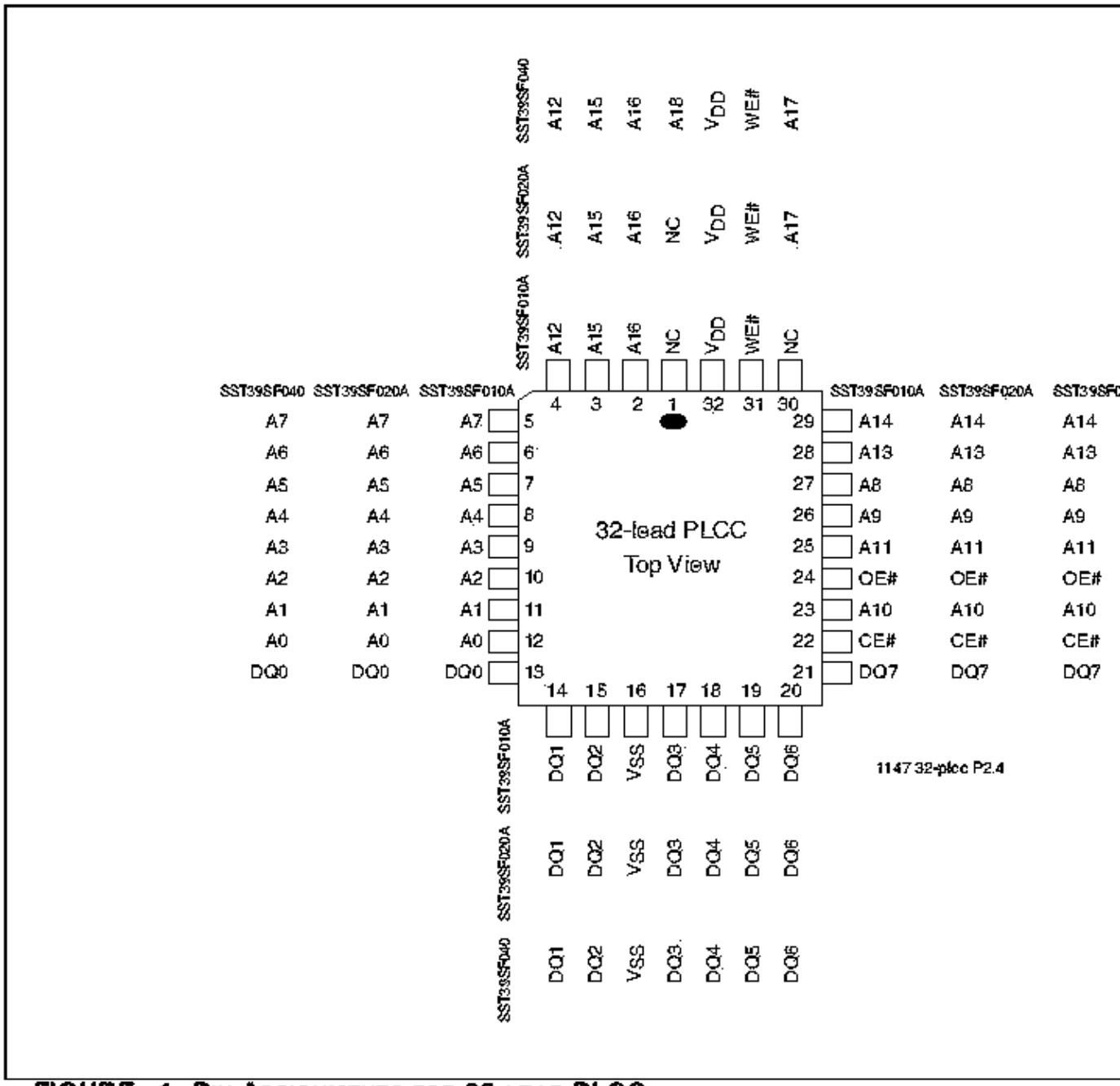


FIGURE 1: PIN ASSIGNMENTS FOR 32-LEAD PLCC

8.11 Linuxbios maillist archives compass

Pipermail Record date	Subjects or Topics

2002-October, 2002-November	ADLO
2002-December	VGA Splash screen
2003-April	Xfree, IDE boot, CF boot
2003-September	Command line boot options
2004-February	Use of CF

8.12 Howto prepare and make a ELF kernel payload with MkelfImage

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

Into kernel option, any gzip compressed Image file makes the deal. Now if you launch a 'file vmlinuz.target', it will answer not stripped. If you want to reduce it a bit more, you can strip it and Image file debug & comments sections will be dropped. Follow this :

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

Now file.elf.stripped which can be rename to file.elf if you want contains strictly want is needed to be loaded. You can check this out by yourself this way :

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

If you want it to be booted from i.g /dev/hda3 which is root partition, you need to copy the first 4Ko on it. Do it this way:

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

8.13 Graphic tests scripts

Script that tests direct writing to stress I/O to frame buffer fb0

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

Script that tests frame buffer device's graphic modes

```
#!/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
```

Next Previous Contents

9. GNU Free Documentation License (Copyright)

GNU Free Documentation License
Version 1.1, March 2000

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles

are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and

visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one

stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but

different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except

as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME.
> Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Next Previous Contents