

**Encadré par**  
M.Elyes OUTAY et M.Mehdi HANZOUTI

---

# DOCUMENTATION LINUX

---

Elaboré par  
Ahmed MEMNI  
[Memniahmed20@gmail.com](mailto:Memniahmed20@gmail.com)  
June 2024





## TABLE DE MATIERE

### Contents

I. INTRODUCTION :	4
II. EMBEDDED LINUX:	4
III. BOOTING SEQUENCE :	5
IV. BUILDING :	7
1. TOOLCHAIN :	7
2. BOOTLOADER :	12
3. KERNEL:	15
4. ROOT FILESYSTEM:	18

## TABLE DE FIGURES

Figure 1 : Boot sequence .....	5
Figure 2 : booting sequence and memory ( beaglbone) .....	6
Figure 3 :list samples crosstool .....	9
Figure 4:list samples crosstool part 2 .....	10
Figure 5 :commande de recherche dans list samples.....	10
Figure 6:output de recherche de list-samples.....	10
Figure 7:commande de confuguration par default .....	11
Figure 8: "/ct-ng menuconfig" menu configuration .....	11
Figure 9:build commande.....	11
Figure 10:commande pour definir la toolchain.....	12
Figure 11:commande d'accé au fichier .....	12
Figure 12:commande d'accé au fichier.....	12
Figure 13:commande pour lister les configuration de default .....	12
Figure 14:list de u-boot bootloader par default.....	13
Figure 15:commande de recherche.....	13
Figure 16:commande pour definir confuguration par default .....	13
Figure 17:commande pour ouvrir l'interface de confuguration .....	13
Figure 18:bootloader menu configuration .....	14
Figure 19 Kernel interactions entre application et hardware .....	15
Figure 20: commande pour accé au repertoire de source noyau.....	16
Figure 21:configuration de l'environemnt.....	16
Figure 22:commande pour afficher les configuration par default .....	16
Figure 23:commande pour charger un configuration par default .....	16
Figure 24:commande pour ouvrir la menu de configuration.....	16
Figure 25:menu de configuration.....	17
Figure 26:commande de construction .....	17

# I. INTRODUCTION :

Au cours de mon parcours dans le domaine de la programmation, j'ai eu l'opportunité de m'immerger profondément dans le développement sous Linux embarqué. Le développement pour Linux embarqué est une compétence essentielle pour créer des systèmes logiciels qui doivent fonctionner de manière fiable dans des environnements aux ressources limitées et souvent critiques. Contrairement au développement logiciel standard, le Linux embarqué exige une compréhension approfondie du matériel, des systèmes d'exploitation en temps réel et des techniques de codage efficaces pour répondre aux contraintes spécifiques des applications embarquées.

Ce rapport récapitule mon parcours d'apprentissage, en mettant en lumière les concepts fondamentaux du développement sous Linux embarqué. En maîtrisant ces techniques, je suis désormais mieux équipé pour contribuer au développement de systèmes embarqués sophistiqués, qui forment la colonne vertébrale de la technologie moderne.

## II. EMBEDDED LINUX:

Le Linux embarqué, ou "Embedded Linux" en anglais, est une version personnalisée du système d'exploitation Linux conçue pour fonctionner sur des appareils et systèmes avec des ressources limitées, comme les systèmes embarqués. Ces systèmes sont souvent utilisés dans des produits industriels, automobiles, de télécommunications, de consommation et bien d'autres.

Les principales caractéristiques du Linux embarqué incluent :

- **GRATUITE**
- **SANS DANGER** La transparence du code source ouvert permet à une large communauté de développeurs d'auditer et d'améliorer continuellement la sécurité du noyau Linux. La modularité et la possibilité de personnaliser les systèmes pour inclure uniquement les composants nécessaires réduisent la surface d'attaque. Les mises à jour régulières, y compris les correctifs de sécurité, sont essentielles pour maintenir la sécurité à jour
- **Modularité** : Possibilité de configurer le noyau et les logiciels pour inclure uniquement les composants nécessaires, réduisant ainsi l'empreinte mémoire et les besoins en ressources.
- **Support matériel** : Large support pour diverses architectures matérielles (ARM, MIPS, PowerPC, etc.), ce qui le rend très flexible pour une multitude de dispositifs

### III. BOOTING SEQUENCE :

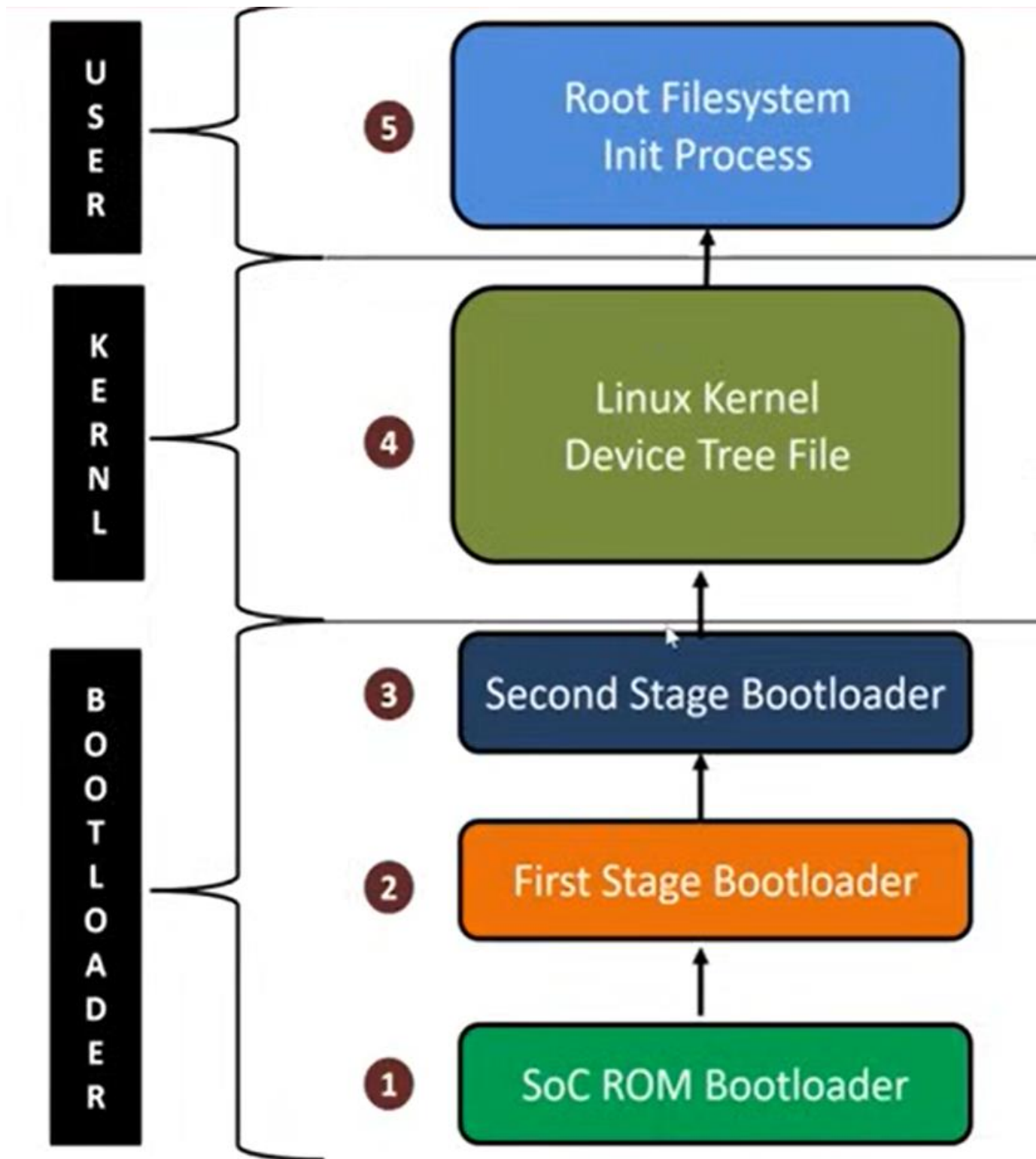


Figure 1 : Boot sequence

Lorsqu'un système embarqué démarre, la ROM de démarrage est le premier code exécuté. Il réside dans une mémoire non volatile sur puce et sa principale responsabilité est d'initialiser le matériel, de rechercher d'éventuelles erreurs ou dysfonctionnements, puis de charger le chargeur de démarrage de premier étage (FSBL) en mémoire.

Le FSPL est responsable de l'initialisation des composants hardware essentiels et les mémoires (external ram) nécessaires pour charger le chargeur d'amorçage principal U-Boot. Il est généralement plus petit et plus simple que le U-Boot principal et réside dans une zone spécifique du périphérique d'amorçage (par exemple, les premiers secteurs d'une mémoire NAND ou d'une carte SD).

U-Boot propre, ou le chargeur d'amorçage de deuxième étape, poursuit le processus d'initialisation initié par le FSPL. Il offre un environnement plus riche en fonctionnalités pour configurer, démarrer et gérer le système embarqué :

- Il Trouve et charge le noyau Linux et le binaire de l'arborescence des périphériques (Device Tree) en RAM.
- Configure les arguments de démarrage du noyau.
- Puis passe le contrôle au noyau qui utilise les arguments de démarrage et l'adresse de l'arborescence des périphériques pour s'initialiser lui-même et initialiser les périphériques matériels.

Enfin, le noyau du système d'exploitation, généralement un RTOS (Real Time Operating System), est chargé en mémoire et exécuté. Le noyau est le cœur du système d'exploitation et est responsable de la gestion des ressources matérielles et logicielles de l'appareil. Il fournit des services tels que la gestion de la mémoire, la gestion des processus, les interfaces de pilotes de périphériques et les interfaces d'appel système aux applications au niveau utilisateur.

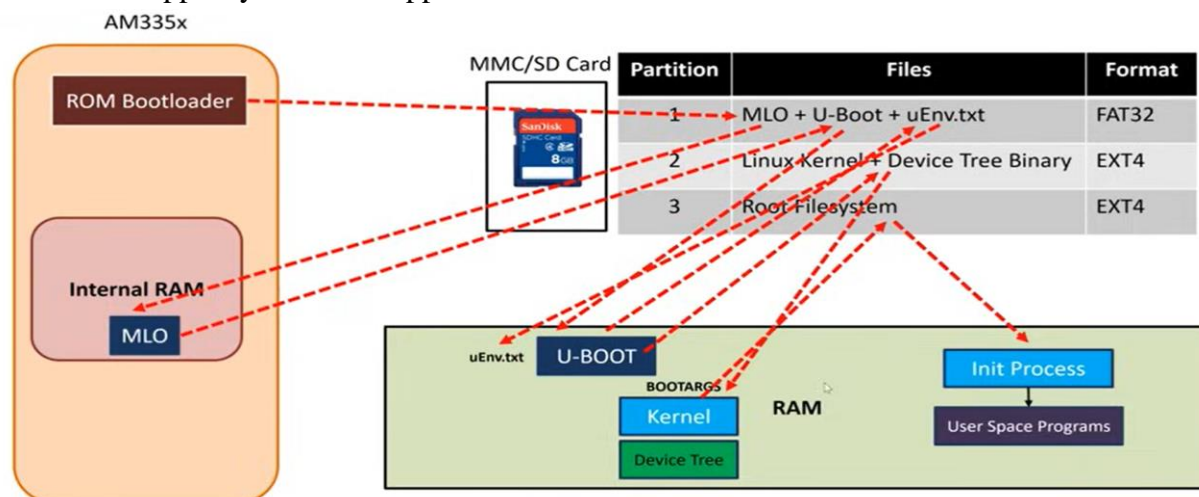



Figure 2 : booting sequence and memory ( beaglbone)



**uEnv.txt** permet de définir divers paramètres de démarrage que U-Boot utilise lors du processus de démarrage. Ces paramètres incluent notamment les arguments de ligne de commande du noyau (bootargs), le délai de démarrage et le périphérique de démarrage par défaut.

## IV. BUILDING :

La construction d'un système Linux embarqué implique la création d'une configuration personnalisée du système d'exploitation adaptée aux exigences spécifiques du matériel embarqué.

Pour construire un système Linux embarqué, il faut suivre ces étapes essentielles adaptées à votre matériel spécifique :

1. Étudier la Carte
2. Construire ou Télécharger le Toolchain
3. Construire le Chargeur d'Amorçage(bootloader)
4. Construire les Modules du Noyau (kernel) et le Blob de l'Arbre de Périphériques (DTB)
5. Construire le Système de Fichiers.
6. Construire l'application
7. Intégration Plug-and-Play

Ce processus méthodique garantit que votre système Linux embarqué soit bien configuré, optimisé et prêt à répondre aux exigences spécifiques de votre application et de votre plateforme matérielle.

La répartition de la carte SD doit prendre en compte :

- **En lecture seule** : Bootloader, image du noyau et DTB.
- **En lecture-écriture** : Système de fichiers racine.

Cette structure garantit que les composants système critiques (bootloader, noyau et DTB) restent stables et protégés contre les modifications accidentelles, tout en permettant la flexibilité et les changements dynamiques dans le système de fichiers racine pendant le fonctionnement du système.

### 1. TOOLCHAIN :

Toolchain est un ensemble d'outils de programmation utilisés pour créer des exécutables pour un environnement cible spécifique, tel qu'un système embarqué. Une chaîne d'outils typique

comprend plusieurs composants clés, chacun jouant un rôle crucial dans le processus de développement et de construction du logiciel. Voici un aperçu des principaux composants d'une chaîne d'outils :

**A. Compilateur**

- **GCC (GNU Compiler Collection)** : Le compilateur traduit le code source écrit dans des langages comme le C, le C++ et d'autres en code machine exécutable par le processeur cible.
- **Clang** : Une alternative à GCC, souvent utilisée pour sa conception modulaire et sa meilleure intégration avec les systèmes de construction modernes.

**B. Assembleur**

- **Binutils (GNU Assembler)** : Convertit le code en langage assembleur en code machine. Il traite également les fichiers objets pour produire l'exécutable final ou la bibliothèque.

**C. LINKER**

- **Binutils (GNU Linker)** : Combine les fichiers objets générés par l'assembleur en un seul exécutable ou une bibliothèque. Il résout les références de symboles entre différents fichiers objets.

**D. Bibliothèques**

- **Bibliothèque standard C** : Fournit des fonctions standard pour les entrées/sorties, la gestion de la mémoire, la manipulation des chaînes et d'autres opérations de base.
  - **glibc** : La bibliothèque C GNU, couramment utilisée dans de nombreuses distributions Linux.
  - **uClibc** : Une bibliothèque C plus petite, conçue pour les systèmes embarqués.
  - **musl** : Une autre bibliothèque C légère, optimisée pour le lien statique et la compatibilité.

**E. Débogueur**

- **GDB (GNU Debugger)** : Permet aux développeurs de déboguer leur code en définissant des points d'arrêt, en inspectant les variables et en contrôlant le flux d'exécution.

**F. Outils de construction**

- **Make** : Un outil d'automatisation de construction qui utilise des Makefiles pour définir comment compiler et lier le programme.
- **CMake** : Un générateur de système de construction multiplateforme qui crée des scripts de construction pour divers environnements.

**G. Utilitaires binaires**

- **Binutils** : Une collection d'outils binaires qui comprend l'assembleur, l'éditeur de liens et d'autres utilitaires comme objdump (pour inspecter les fichiers objets), nm (pour lister les symboles) et strip (pour supprimer les symboles inutiles).

On peut utiliser les quatre méthodes suivantes pour construire une toolchain : crosstool-NG, Yocto Project, Buildroot et la méthode manuelle. Chacune de ces méthodes offre ses propres avantages et est adaptée à différents niveaux de complexité et de besoins de personnalisation. Cependant, **crosstool-NG** se distingue comme la meilleure méthode pour plusieurs raisons. Il s'agit d'un outil spécialement conçu pour la construction de toolchains, offrant une interface conviviale pour la

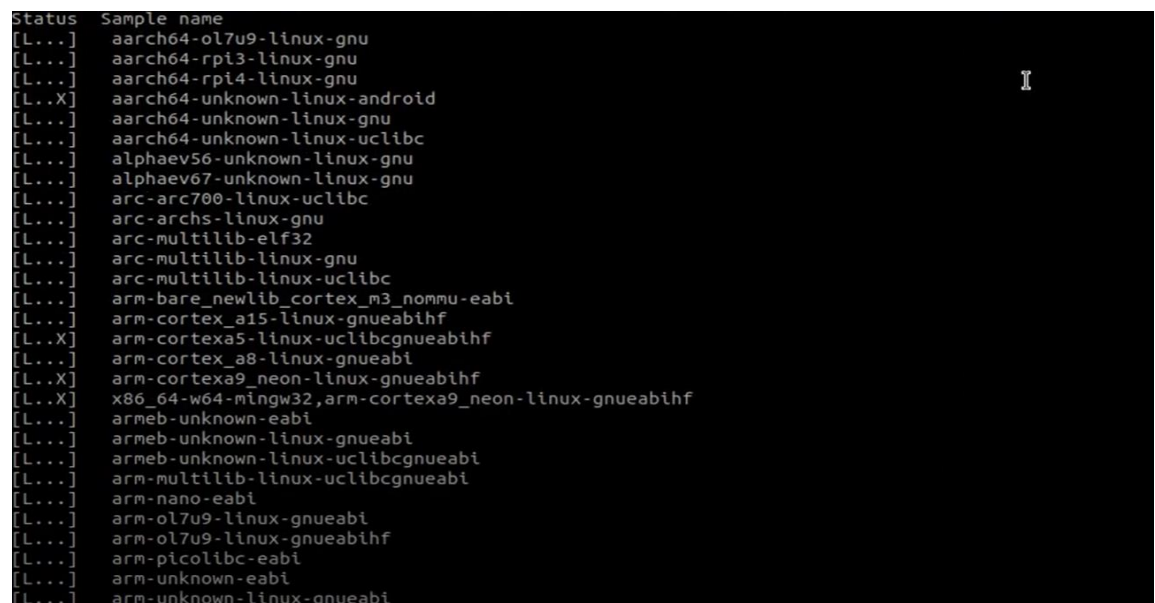


configuration et la gestion de tout le processus. Crosstool-NG permet une grande flexibilité dans le choix des versions de GCC, binutils, et autres composants, tout en automatisant de nombreuses étapes fastidieuses. Sa documentation détaillée et sa large communauté d'utilisateurs facilitent également la résolution des problèmes et l'optimisation des toolchains pour des architectures spécifiques. Grâce à ces avantages, crosstool-NG est souvent considéré comme le choix privilégié pour les développeurs cherchant à créer des toolchains robustes et personnalisées de manière efficace.

## CROSSTOOL-NG:

Une fois installé, l'outil ct-ng permet de configurer et de construire un nombre arbitraire de toolchains.

- Son système de configuration est basé sur kconfig, comme le système de configuration du noyau Linux.
- La configuration de la toolchain à construire est stockée dans un fichier .config.
- Des configurations d'exemple sont fournies avec Crosstool-NG :
  - Liste : `./ct-ng list-samples``



```
Status Sample name
[L...] aarch64-ol7u9-linux-gnu
[L...] aarch64-rpi3-linux-gnu
[L...] aarch64-rpi4-linux-gnu
[L..X] aarch64-unknown-linux-android
[L...] aarch64-unknown-linux-gnu
[L...] aarch64-unknown-linux-ucLibc
[L...] alphaev56-unknown-linux-gnu
[L...] alphaev67-unknown-linux-gnu
[L...] arc-arc700-linux-ucLibc
[L...] arc-archs-linux-gnu
[L...] arc-multilib-elf32
[L...] arc-multilib-linux-gnu
[L...] arc-multilib-linux-ucLibc
[L...] arm-bare_newlib_cortex_m3_nommu-eabi
[L...] arm-cortex_a15-linux-gnueabihf
[L..X] arm-cortexa5-linux-ucLibcgnueabihf
[L...] arm-cortex_a8-linux-gnueabi
[L..X] arm-cortexa9_neon-linux-gnueabihf
[L..X] x86_64-w64-mingw32,arm-cortexa9_neon-linux-gnueabihf
[L...] armeb-unknown-eabi
[L...] armeb-unknown-linux-gnueabi
[L...] armeb-unknown-linux-ucLibcgnueabi
[L...] arm-multilib-linux-ucLibcgnueabi
[L...] arm-nano-eabi
[L...] arm-ol7u9-linux-gnueabi
[L...] arm-ol7u9-linux-gnueabihf
[L...] arm-picolibc-eabi
[L...] arm-unknown-eabi
[L...] arm-unknown-linux-gnueabi
```

Figure 3 :list samples crosstool

```

[L...] pru
[L..X] riscv32-hifive1-elf
[L..X] riscv32-unknown-elf
[L..X] riscv64-unknown-elf
[L..X] riscv64-unknown-linux-gnu
[L..X] s390-ibm-linux-gnu
[L..X] s390-unknown-linux-gnu
[L...] s390x-ibm-linux-gnu
[L...] s390x-unknown-linux-gnu
[L...] sh-multilib-linux-gnu
[L...] sh-multilib-linux-uclibc
[L...] sh-unknown-elf
[L...] sparc64-multilib-linux-gnu
[L...] sparc-leon-linux-uclibc
[L...] sparc-unknown-linux-gnu
[L..X] tic6x-uclinux
[L...] x86_64-centos6-linux-gnu
[L...] x86_64-centos7-linux-gnu
[L...] x86_64-multilib-linux-gnu
[L..X] x86_64-multilib-linux-musl
[L...] x86_64-multilib-linux-uclibc
[L..X] x86_64-w64-mingw32,x86_64-pc-linux-gnu
[L...] x86_64-ubuntu14.04-linux-gnu
[L...] x86_64-ubuntu16.04-linux-gnu
[L...] x86_64-unknown-linux-gnu
[L...] x86_64-unknown-linux-uclibc
[L..X] x86_64-w64-mingw32
[L..X] xtensa-fsf-elf
[L...] xtensa-fsf-linux-uclibc
L (Local) : sample was found in current directory
G (Global) : sample was installed with crosstool-NG
X (EXPERIMENTAL): sample may use EXPERIMENTAL features
B (BROKEN) : sample is currently broken
O (OBSOLETE) : sample needs to be upgraded

```

Figure 4: list samples crosstool part 2

-Pour afficher les échantillons disponibles avec Crosstool-NG, vous pouvez utiliser la commande suivante avec grep pour filtrer les résultats. Par exemple, pour trouver des échantillons spécifiques au Raspberry Pi, vous pouvez utiliser :

```
ct-ng list-samples | grep rpi
```

Figure 5 :commande de recherche dans list samples

```

[G...] aarch64-rpi3-linux-gnu
[G...] aarch64-rpi4-linux-gnu
[G...] armv7-rpi2-linux-gnueabihf
[G...] armv8-rpi3-linux-gnueabihf
[G...] armv8-rpi4-linux-gnueabihf

```

Figure 6:output de recherche de list-samples

- Charger un exemple : `./ct-ng <nom-de-l'échantillon>`, remplace le fichier .config

- Par exemple : `./ct-ng armv8-rpi4-linux-gnueabihf`

```
ct-ng armv8-rpi4-linux-gnueabi
```

Figure 7: commande de configuration par défaut

- Si aucun échantillon n'est chargé, la configuration par défaut de Crosstool-NG est une toolchain bare-metal pour l'architecture CPU Alpha

- La configuration peut ensuite être affinée en utilisant soit :

- `./ct-ng menuconfig`

- `./ct-ng nconfig`

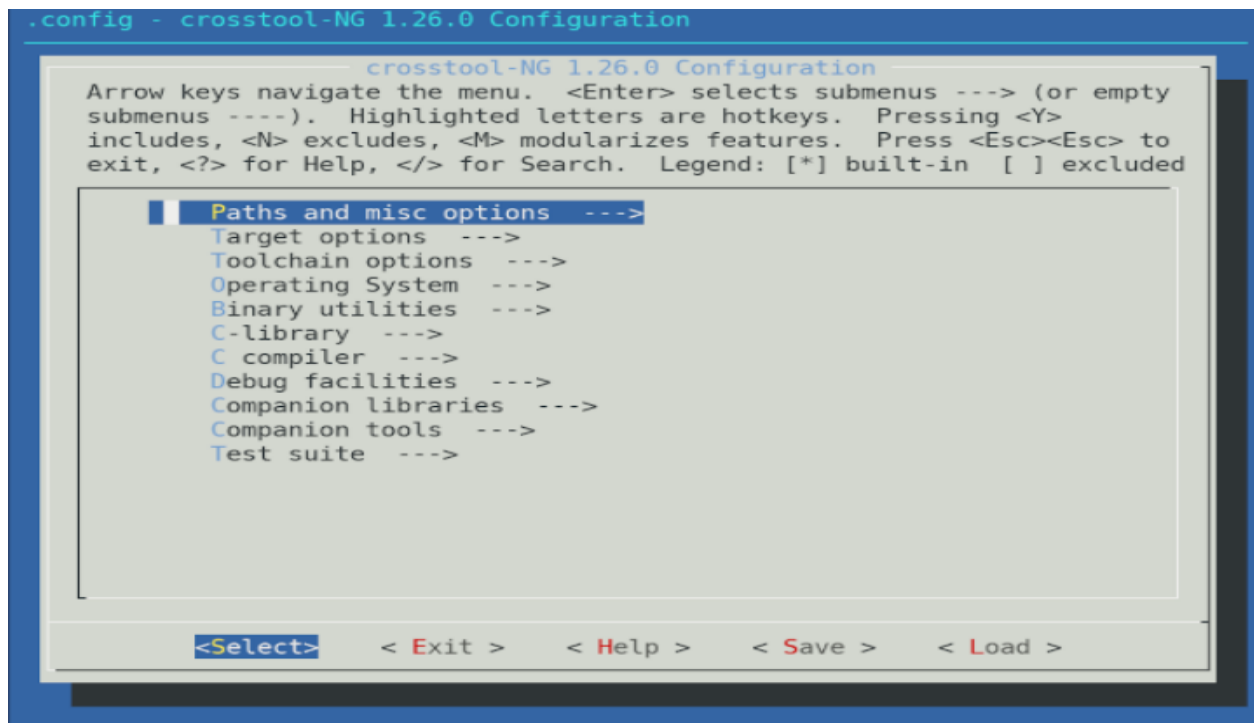


Figure 8: "/ct-ng menuconfig" menu configuration

To build the toolchain

```
ct-ng build
```

Figure 9: build commande

## 2. BOOTLOADER :

Un bootloader dans les systèmes embarqués est un logiciel crucial qui initialise le matériel dès le démarrage d'un système informatique et charge ensuite le noyau principal du système d'exploitation ou un autre programme en mémoire. Il assure le bon fonctionnement du démarrage en gérant les premières étapes critiques, telles que l'initialisation du processeur, la configuration de la mémoire (RAM) et la gestion des périphériques essentiels comme les UARTs et les dispositifs de stockage. Parmi les bootloaders utilisés, U-Boot se distingue particulièrement pour sa polyvalence et sa robustesse, adaptée à une variété d'architectures telles que ARM, MIPS et PowerPC, fréquemment rencontrées dans les systèmes embarqués. U-Boot offre une gamme étendue de fonctionnalités avancées, notamment la configuration flexible du démarrage, la gestion des paramètres de boot, la possibilité de démarrer plusieurs systèmes d'exploitation et une interface utilisateur permettant d'interagir avec le processus de démarrage. En résumé, U-Boot est largement apprécié dans le domaine des systèmes embarqués pour sa fiabilité, sa capacité à s'adapter à différents matériels et sa robustesse lors de la gestion du processus de démarrage.

Avant d'utiliser une configuration par défaut d'U-Boot, il faut s'assurer d'avoir correctement défini la variable `CROSS_COMPILE` pour pointer vers votre chaîne d'outils de cross-compilation. Cette variable spécifie le préfixe des binaires de la chaîne d'outils qui vont construire U-Boot pour votre architecture cible, tel que `arm-linux-gcc` pour les systèmes basés sur ARM. Définir `CROSS_COMPILE` correctement est essentiel pour garantir que U-Boot se construise correctement pour la plateforme visée : Il faut utiliser la commande `export` pour définir la variable `CROSS_COMPILE`. Par exemple, si notre binaires de chaîne d'outils de compilation croisée sont préfixés par `arm-linux-`, il faut la définir ainsi :

```
export CROSS_COMPILE=arm-linux-
```

*Figure 10:commande pour definir la toolchain*

Pour accéder aux fichiers de configuration par défaut (`*_defconfig`) dans U-Boot :

```
cd /chemin/vers/le/source/uboot
```

*Figure 11:commande d'accé au fichier*

Dans le répertoire source de U-Boot, on trouve généralement un dossier nommé `configs` ou similaire. Ce dossier contient différents fichiers de configuration (`*_defconfig`) pour différents cartes et plateformes prises en charge par U-Boot.

```
cd configs
```

*Figure 12:commande d'accé au fichier*

Une fois dans le répertoire `configs`, on utilise la commande `ls` pour lister tous les fichiers de configuration disponibles.

```
ls *_defconfig
```

*Figure 13:commande pour lister les configuration de default*

```
$ ls *defconfig
am335x_evm_defconfig      imx6_sabreauto_defconfig
am57xx_evm_defconfig      imx6_sabresd_defconfig
beagle_x15_defconfig      imx6_sabrelite_defconfig
beaglebone_defconfig      imx6sabre_common_defconfig
imx6_cubox_i_defconfig    myboard_defconfig
```

*Figure 14: list de u-boot bootloader par défaut*

Il faut utiliser grep pour rechercher un motif ou un nom spécifique parmi la liste des fichiers de configuration (\*\_defconfig). Par exemple, pour trouver les configurations relatives à la carte BeagleBone :

```
$ ls *defconfig | grep -i beaglebone
beaglebone_defconfig
```

*Figure 15: commande de recherche*

Il faut utiliser ces fichiers de configuration directement avec la commande make pour configurer U-Boot pour votre carte spécifique. Par exemple :

```
make myboard_defconfig
```

*Figure 16: commande pour définir configuration par défaut*

On utilise make menuconfig pour ouvrir l'interface de configuration de U-Boot :

```
make menuconfig
```

*Figure 17: commande pour ouvrir l'interface de configuration*

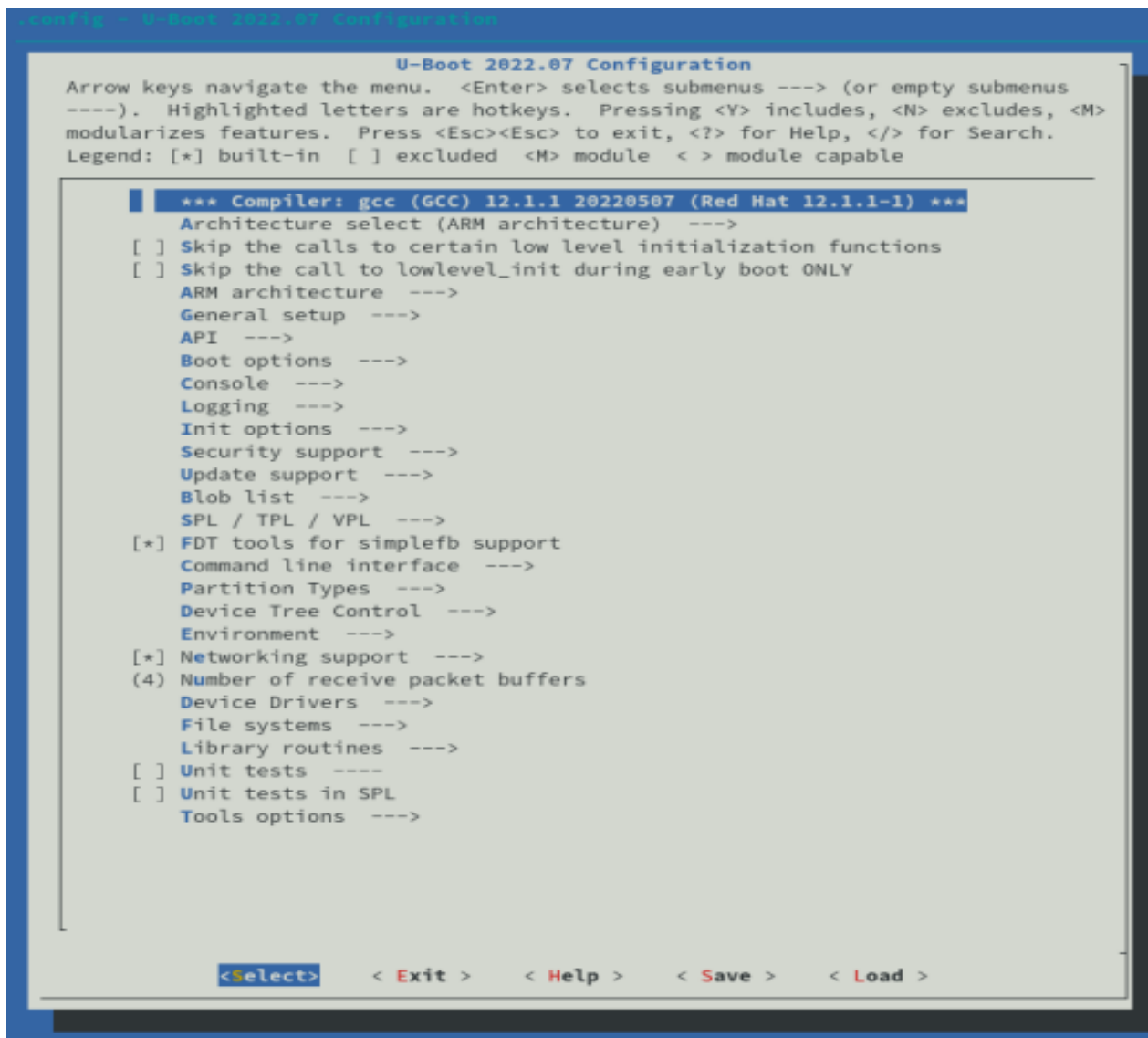


Figure 18:bootloader menu configuration

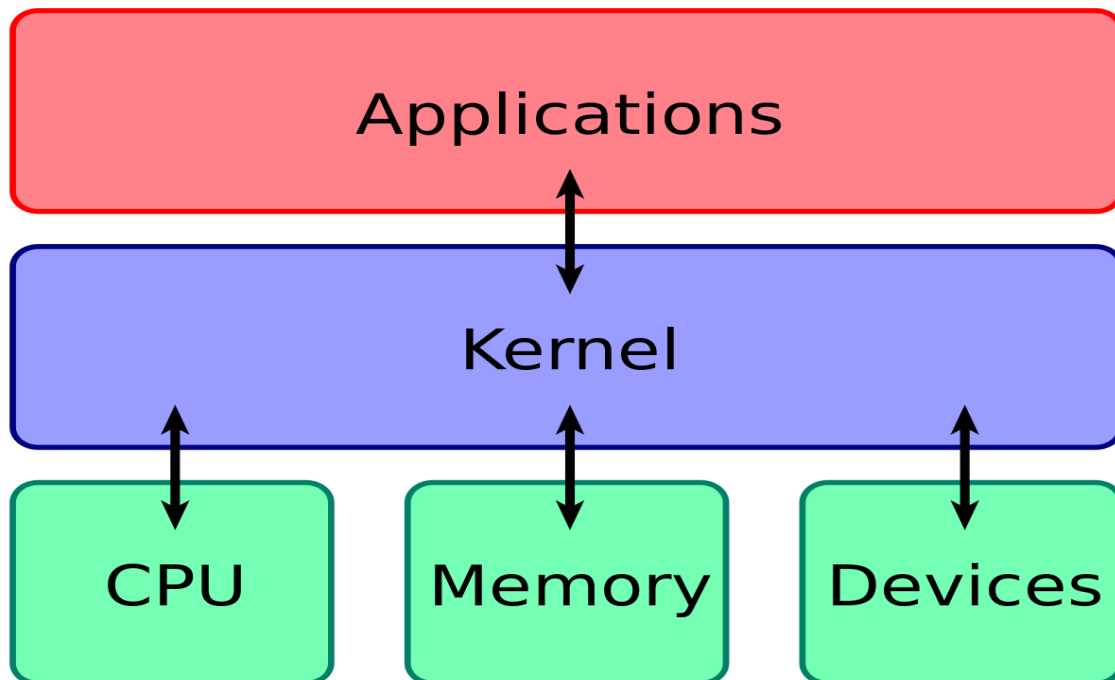
Il est essentiel de ne pas oublier de créer le fichier `uboot.env` pour la configuration de l'environnement de U-Boot. Dans le sous-menu de l'environnement, la configuration garantit la persistance des changements de variables à travers les redémarrages en désactivant "Environment is not stored" et en activant "Environment is in an EXT4 filesystem". Les autres options de stockage telles que MMC, SPI et UBI sont désactivées pour maintenir la clarté et la concentration. Spécifiez le périphérique de bloc pour l'environnement comme `mmc`, avec le périphérique et la partition configurés sur `0:4` dans le système de fichiers EXT4. Le fichier EXT4 désigné `uboot.env` est spécifié pour servir de lieu de stockage des variables d'environnement de U-Boot, assurant ainsi la fiabilité et la continuité des paramètres de configuration.

### 3. KERNEL:

Le noyau, ou kernel, est le cœur du système d'exploitation qui agit comme une interface essentielle entre le matériel et le logiciel. Il gère les ressources matérielles, assure la communication entre les composants matériels et les applications, et fournit des services essentiels tels que la gestion de la mémoire, des processus et des périphériques. En tant que gestionnaire central, le noyau prend en charge l'allocation de la mémoire, le planificateur de tâches, les pilotes de périphériques et les interfaces réseau, garantissant ainsi que toutes les parties du système fonctionnent en harmonie.

La configuration et l'optimisation du noyau sont cruciales pour le bon fonctionnement du système, car elles déterminent la manière dont les ressources matérielles sont utilisées et gérées. Un noyau bien configuré peut améliorer les performances du système, la stabilité et la sécurité.

Dans le processus de construction d'une image de noyau pour un système embarqué, le noyau représente le troisième élément clé, après la construction du chargeur de démarrage (bootloader) et la configuration de la chaîne d'outils de cross-compilation. Le chargeur de démarrage initialise le matériel et charge le noyau en mémoire, tandis que la chaîne d'outils de cross-compilation fournit les outils nécessaires pour compiler le noyau pour l'architecture cible. Le noyau, une fois compilé, utilise les informations de l'arborescence des périphériques (Device Tree) pour comprendre la configuration matérielle spécifique, assurant ainsi une initialisation et une gestion correctes des composants matériels lors de l'exécution.



*Figure 19 Kernel interactions entre application et hardware*

voici les commandes pour construire une image du noyau, en français, avec les étapes nécessaires pour préparer l'environnement avant la compilation

Accédez au répertoire de votre source du noyau

```
cd $HOME/embedded-linux-4d-labs/linux-stm32
```

*Figure 20: commande pour accès au repertoire de source noyau*

Avant de construire le noyau, vous devez configurer la chaîne d'outils et l'architecture. Cela se fait en définissant les variables d'environnement appropriées : `export ARCH=arm` pour l'architecture ARM et `export CROSS_COMPILE=arm-linux-` pour spécifier le préfixe de la chaîne d'outils de cross-compilation.

```
export ARCH=arm
export CROSS_COMPILE=arm-linux-
```

*Figure 21: configuration de l'environnement*

Ouvrir la liste des exemples de configurations

```
make list_defconfigs
```

*Figure 22: commande pour afficher les configuration par défaut*

Charger la configuration par défaut pour votre carte (STM32MP157)

```
make stm32mp157_defconfig
```

*Figure 23: commande pour charger un configuration par défaut*

Optionnel : Personnaliser la configuration du noyau

```
make menuconfig
```

*Figure 24: commande pour ouvrir la menu de configuration*



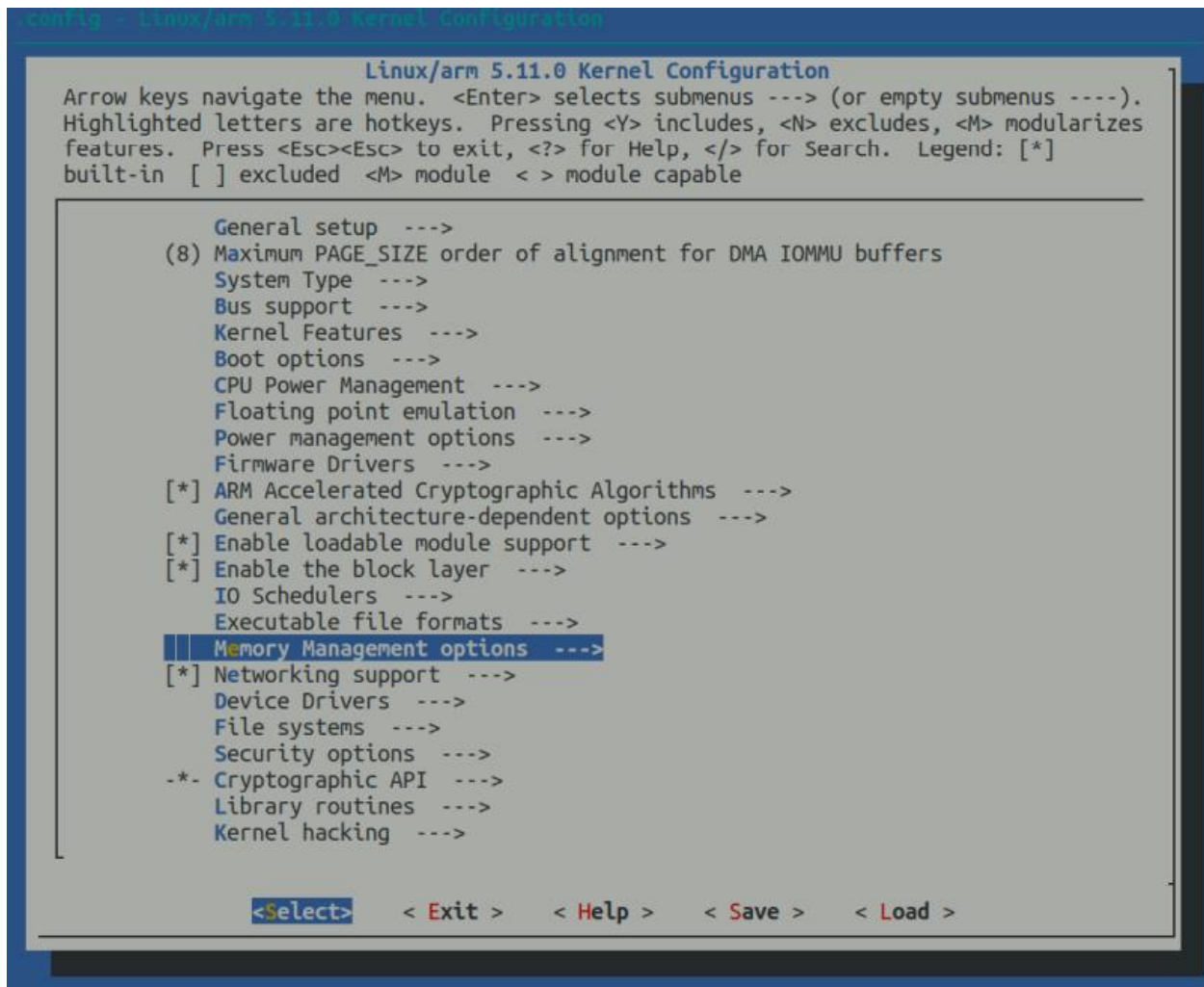


Figure 25:menu de configuration

Commande pour construire le noyau

```
make
```

Figure 26:commande de construction

## 4. ROOT FILESYSTEM:

Le noyau obtiendra un système de root filesystem, soit sous la forme d'un initramfs passé en tant que pointeur depuis le bootloader, soit en montant le périphérique de bloc indiqué par root= paramètre dans la ligne de commande du noyau. Une fois qu'il a un système de fichiers racine, le noyau exécutera le premier programme, nommé par défaut init. Ensuite, du point de vue du noyau, son travail est terminé. Il revient au programme init de commencer à lancer d'autres programmes et de faire démarrer le système. Pour créer un système de fichiers racine minimal, vous avez besoin de ces composants :

- **init** : C'est le programme qui lance tout, généralement en exécutant une série de scripts. Je décrirai plus en détail le fonctionnement de init dans le chapitre 13, Démarrage - Le programme init.
- **Shell** : Vous avez besoin d'un shell pour obtenir un invite de commande, mais surtout pour exécuter les scripts shell appelés par init et d'autres programmes.
- **Daemons** : Un daemon est un programme en arrière-plan qui fournit un service aux autres. De bons exemples sont le daemon de journalisation système (syslogd) et le daemon shell sécurisé (sshd). Le programme init doit démarrer la population initiale des daemons pour prendre en charge les principales applications du système. En fait, init est lui-même un daemon : il est le daemon qui fournit le service de lancement des autres daemons.
- **Bibliothèques partagées** : La plupart des programmes sont liés à des bibliothèques partagées, il est donc nécessaire qu'elles soient présentes dans le système de fichiers racine.
- **Fichiers de configuration** : La configuration pour init et les autres daemons est stockée dans une série de fichiers texte, généralement dans le répertoire /etc.
- **Nœuds de périphériques** : Ce sont des fichiers spéciaux qui donnent accès à divers pilotes de périphériques.
- **proc et sys** : Ces deux pseudo-filesystems représentent les structures de données du noyau sous forme d'une hiérarchie de répertoires et de fichiers. De nombreux programmes et fonctions de bibliothèque dépendent de /proc et /sys.
- **Modules de noyau** : Si vous avez configuré certaines parties de votre noyau comme modules, ils doivent être installés dans le système de fichiers racine, généralement dans /lib/modules/[version du noyau].