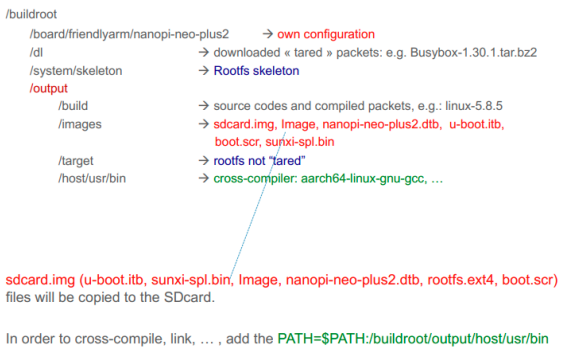
1 Buildroot

1.1 Principaux répértoires



Ce qui est manquant dans le répértoire dans le dossier output sera recompilé lors de la commande `make` ou en précisant le paquet : `make <package>-rebuild` à refaire.

1.2 Principe de fonctionnement

Basé sur des fichiers `make kconfig`, buildroot est un outil qui automatise le process de construction d'un système Linux embarqué en utilisant la cross-compilation. Lors de la commande `make` il va s'occuper de compiler l'entier du système et préparer une image complète et prête pour l'utilisation.

1.3 Configuration pour un hardware donné

Il y a un répertoire situé dans `board/friendlyarm/nanopi-neo-plus2` qui contient plusieurs fichiers intéressants. On y retrouve notamment le fichier `nanopi-neo-plus2/nanopi-neo-plus2.dts` qui contient le FTD ("Flattened Device Tree") qui décrit le hardware. On indique donc à buildroot l'emplacement de ce fichier avec la commande `menuconfig`. Autres fichiers intéressants :

```
root →/buildroot/board/friendlyarm/nanopi-neo-plus2 (main ✗) $ ls
boot.cmd              genimage.cfg         nanopi-neo-plus2.dts  rootfs_overlay
busybox-extras.config linux-extras.config  post-build.sh         ses_linux_defconfig
extlinux.conf         my_patches           readme.txt            uboot-extras.config
```

1.4 Patch buildroot

Il faut spécifier le dossier des patchs `make menuconfig` nous on a fait dans `/nanopi-neo-plus2/my_patches` et un dossier par package à patcher (il faut aussi éditer `ses\-config` pour garder le chemin des patches). Ensuite la technique consiste à profiter de l'outil git et se mettre dans `/buildroot/output/build/uboot-2020.07` et de faire:

```
git init --initial-branch=main
git add .
cd common/
git add .
git commit -m "1st commit"
```

Faire ensuite toutes les modifications à faire, les stager et faire la commande qui créé le patch au format voulu par buildroot:

```
git diff --cached --stat -p > /buildroot/boa⌋
↪ rd/friendlyarm/nanopi-neo-plus2/my_patch⌋
↪ es/uboot/0001_stack_protector.patch
```

On peut ensuite supprimer le paquet en question dans build et il se téléchargera et patchera automatiquement au prochain `make`.

1.5 Configuration de buildroot, u-boot, kernel

Pour buildroot, on utilise `make menuconfig` ça se sauvegarde dans `.config` ou dans `configs/ses_defconfig` (seulement les changement sont dans ce fichier, qui est à la base une copy de `friendlyarm_nanopi_neo_plus2_defconfig` qui se situe au même endroit).
Pour u-boot, on va dans `make menuconfig` catégorie bootloaders et ça va se sauver dans

```
/buildroot/output/build/uboot-xx/configs/nan⌋
↪ opi_neo_plus_defconfig
```

Et dans un fragment files qui ne contient que des petites modifications (save old, make new, use diff, add it to this fragment file):

```
/buildroot/board/friedlyarm/nanopi-neo-plus/⌋
↪ uboot-extras.config
```

Pour le kernel, `make linux-menuconfig`, ça save dans `/buildroot/output/build/linux-xx/defconfig` et nous on le copie et le met dans `ses_linux_defconfig`

1.6 Génération de la carte SD

Pour créer le fichier `sdcard.img` utilise le script `genimage.sh` qui va aller aussi utiliser le fichier `nanopi-neo-plus2/genimage.cfg` qui spécifie les différentes partitions à créer et les images (venant de `output/images`) à mettre dedans.

1.7 Génération du rootfs

Il est copié depuis `/buildroot/system/skeleton` pour le mettre dans `/buildroot/output/target`. Il est ensuite possible d'ajouter des fichiers et des répértoires avec le `/nanopi-neo-plus2/rootfs_overlay`. Après la commande `make` le pseudo rootfs est créé et placé dans un des fichiers images de sorties `rootfs.xxx`

1.8 rootfs_overlay

On peut préprarer la structure des fichiers de la cible dans ce dossier

1.9 Installer un nouveau package dans buildroot

Pour utiliser un package, il faut faire `make menuconfig` et aller choisir le package à aller utiliser. Sinon il faut ajouter le nouveau package "foo" dans le dossier `/buildroot/package`. Il doit contenir au moins:

- Config.in: in kconfig language : on pourra y accéder à travers `make menuconfig`

- foo.mk: fichier qui décrit ou prendre les sources et comment les installer
- optional foo.hash: pour vérifier l'intégrité des fichiers à télécharger
- optional Sxx_foo: le startup script pour le package foo

2 Uboot

2.1 Démarrage du NanoPi

1. Lorsque le uP est mis sous tension, le code stocké dans BROM va charger dans ses 32KiB se SRAM interne le firmware "sunxi-spl" stocké dans le secteur n°16 de la carte SD/eMMC et l'exécuter
2. Le firmware "sunxi-spi" (Secondary Program Loader) initialise les couches basses du uP, puis charge l'U-Boot dans la RAM du uP avant de le lancer
3. L'U-Boot va effectuer les initialisation hardware nécessaires (horloges, contrôleurs, ...) avant de charger l'image non compressée du noyau Linux dans la RAM, le fichier Image, ainsi que le fichier de configuration FDT (flattened device tree).
4. L'U-Boot lancera le noyau Linux en lui passant les arguments de boot (bootargs).
5. Le noyau Linux procèdera à son initialisation sur la base des bootargs et des éléments de configuration contenus dans le fichier FDT (sun50i-nanopi-neo-plus2.dtb).
6. Le noyau Linux attachera les systèmes de fichiers (rootfs, tmpfs, usrfs, ...) et poursuivra son exécution

2.2 Principales commandes de Uboot durant le boot

Pour entrer en uboot mode, il faut appuyer sur une touche durant le boot et on arrive dans le prompt u-boot. Ensuite on peut taper help pour avoir une liste des commandes. Les commandes princpales sont

- booti - boot Linux kernel 'Image' format from memory
- ext2load - load binary file from a Ext2 filesystem
- ext2ls - list files in a directory (default /)
- ext4load - load binary file from a Ext4 filesystem
- ext4ls - list files in a directory (default /)
- ext4size - determine a file's size
- fatinfo - print information about filesystem
- fatload - load binary file from a dos filesystem
- fatls - list files in a directory (default /)
- fatmkdir - create a directory
- fatrm - delete a file
- fatsize - determine a file's size
- fatwrite - write file into a dos filesystem
- mmcinfo - display MMC info
- printenv - print environment variables
- setenv - set environment variables

Le fichier `boot.cmd` (tranformé en `boot.scr` par une commande montrée plus bas) contient les commandes que u-boot effecture :

```
setenv bootargs console=ttyS0,115200
↪  earlyprintk root=/dev/mmcblk2p2 rootwait
ext4load mmc 0 $kernel_addr_r Image
ext4load mmc 0 $fdt_addr_r
↪  nanopi-neo-plus2.dtb
booti $kernel_addr_r - $fdt_addr_r
```

La commande `booti $kernel_addr_r - $fdt_addr_r` spécifie l'adresse de l'image, de du fdt et le - précise qu'il n'y a pas de initrd. Voir plus tard avec initramfs où on fera autrement
Création d'un fichier `boot.scr` à partir d'un `boot.cmd`

```
mkimage -C none -A arm64 -T script -d board/
↪  friendlyarm/nanopi-neo-plus2/boot.cmd
↪  output/images/boot.scr
```

2.3 Configuration de uboot

`make uboot-menuconfig` pour configurer (cette config se sauve dans output ubuild uboot-2020.07 .config), `make uboot-rebuild` ou `rm output/build/uboot-xx/.stamp-built` et `make` pour compiler. Après le make 2 fichiers sont créés: `u-boot.itb` et `boot.scr` et se trouvent dans `output/images/`

2.4 Sécurisation de uboot

2 choses sont à faire :

- Retirer les informations de debug, c'est à dire l'option -g lors de la compilation
- Ajouter l'option `-fstatck-protector-strong`. Qui permet d'éviter les attack en buffer overflow, un programme va donc s'arrêter en cas de détection de stack smashing.

Pour mettre l'option de proctection de stack de base dans uboot, il faut faire un patch en éditant le Makefile de uboot de cette manière:

```
KBUILD_CFLAGS += $(call
↪  cc-option,-fstack-protector-strong)
#KBUILD_CFLAGS += $(call
↪  cc-option,-fno-stack-protector)
```

Et ajouter `stackprot.o` qui contient une fonction `__stack_chk_fail` dans le `/common/` et ajouter la ligne `obj-y += stackprot.o` dans le Makefile du common.

2.5 Etapes pour la création de l'image de u-boot.itb

L'image `u-boot.itb` contient 3 parties:

- `u-boot-nodtb.bin` qui est le code de uboot qui a été créé à partir du `elf` de u-boot avec la commande `aarch64-linux-objcopy --gap-fill=0xff` qui fait attention à ne garder que le strict nécessaire (sans debug, symbol, relocation)
- `b131.bin`: trust zone ou ARM Trusted Firmware
- `sun50i-h5-nanopi-neo-plus2.dtb` : Device Tree blob (voir plus bas)

La commande qui fait ça est :
`mkimage -f u-boot.its -E u-boot.itb` (est c'est dans `u-boot.its` (fichier texte) qu'est spécifié les différentes parties de l'image `u-boot.itb`

2.6 Commande strip sur un elf

La commande `aarch64-linux-strip u-boot` (la commande `aarch64-linux-objcopy` le fait automatiquement) supprime les information de symbole et de debug : voici la différence sur le fichier `u-boot.elf`

Diassemble with symbols

```
arm-linux-gnueabihf-objdump –d u-boot
   43e00058 <reset>:
   43e00058:       eb0004f8        bl      43e01440 <save_boot_params_default>
   43e0005c:       e10f0000        mrs     r0, CPSR
   43e00060:       e3c0001f        bic     r0, r0, #31
   43e00064:       e38000d3        orr     r0, r0, #211    ; 0xd3
   43e00068:       e129f000        msr     CPSR_fc, r0
```

Diassemble without symbols

```
arm-linux-gnueabihf-objdump –d u-boot
   43e00058:       eb0004f8        .word   0xeb0004f8
   43e0005c:       e10f0000        .word   0xe10f0000
   43e00060:       e3c0001f        .word   0xe3c0001f
   43e00064:       e38000d3        .word   0xe38000d3
   43e00068:       e129f000        .word   0xe129f000
```

2.7 Création de uImage

uImage est l'image Linux au format uboot elle est créée lors du make. Dans le process du boot.cmd, uboot va spécifier son emplacement dans la variable d'environnement `$kernel_addr_r=0x40080000`

2.8 Flattened Device Tree

Le FTD contient les informations sur le hardware que linux va utiliser pour sa configuration. Le fabricant du uP a surement écrit `sun50i-h5-nanopi-neo-plus2.dts` et on le passe en `.dtb` avec la commande `dtc board.dts {o board.dtb`. Lors du boot.cmd on précise l'adresse de cd `.dtb` dans la variable d'environnement `$fdt_addr_r=0x4FA00000`

2.9 Mapping de la SDCard

La carte SD contient ces éléments

- `sunxi-spl.bin`: Secondary Program Loader
- `u-boot.itb`: selon plus haut
- `boot`: image `.ext4` qui contient 3 autres choses: `Image`, `nanopi-neo-plus2.dtb` et `boot.scr`
- `rootfs`: le filesystem de linux

2.10 boot.scr

u-boot au démarrage, sans pression d'une touche, va chercher le fichier `boot.scr` (format boot script) et l'exécuter. Il contient enfait les instructions du fichier `boot.cmd`.

3 Kernel

3.1 Principaux répertoire du noyau linux

Les principaux répertoires du kernel sont listés:

- arch Hardware dependent code
- block Generic functions for the block devices
- crypto Cryptographic algo. used in the kernel
- Documentation Documentation about the kernel
- drivers All drivers known by the kernel
- fs All filesystem know by the kernel
- include kernel include files
- init Init code (function start_kernel)
- ipc Interprocess communication
- kernel Kernel code, scheduler, mutex, ...
- lib different libraries used by the kernel
- mm Memory management
- net Different protocols, IPv4, IPv6, bluetooth, ...
- samples Different examples, kobject, kfifo, ...
- security Encrypted keys, SELinux, ...
- sound Sound support for Linux kernel
- virt Kernel-based virtual machine

3.2 Principales méthodes pour sécuriser le noyau

Pour configurer linux : `make linux-menuconfig` ou `make linux-xconfig`

- ne pas inclure le Linux kernel `.config` dans le kernel : `Kernel .config support`
- protéger la stack en active la détection de buffer overflow : General architecture-dependant options =¿ [*] Stack Protector buffer overflow detection, [*] Strong Stack Protector