Linux pour les systèmes dédiés

Fabien Lahoudère

Consultant Linux embarqué chez Open Wide
fabienlahoudere.pro@gmail.com
aragua sur github et IRC

INSA Linux embarqué, Novembre 2015

Sommaire

Intro
Systèmes dédiés?

Un système dédié est un système concu pour ne réaliser qu'un certains nombre de tache défini.
Exemple:

- Serveur web
- Calculateur automobile
- Box multimedia

On associe un système à une fonction (ou un groupe de fonction).
On cherche à maitriser le fonctionnement du système.
Différe des systèmes génériques censés savoir tout faire sur n'importe quelle plateforme.

Intro
Systèmes dédiés GNU/Linux?

Utilisé depuis longtemps comme serveur.
Pourquoi ne pas l'utiliser pour d'autre système générique? (systèmes embarqués)
De nombreuses architectures supportées par Linux (ARM, mips, powerpc ...)
Systèmes fiables et performants
De nombreuses fonctionnalités déjà codé et validé
Nécessite un BSP (Board support package) et un peu d'adaptation.

Intro
Eléments nécéssaires

Une système GNU/Linux dédié nécéssite:

- Chaîne de compilation croisée (Gcc,as,ld,LibC)
- Un Bootloader (U-Boot, barebox, grub, isolinux)
- Noyau Linux adapté à l'archi hardware
- Outils GNU/LINUX Commandes Linux (sh, ls, cp, etc.) Applications ...
- Outil de génération (BR, OE, ...)

Toolchain

Un point très complexe !
Nécessité de construire une chaîne croisée :

- Gcc
- Binutils (as, ld, ...)
- Dépendances avec le noyau (system calls, ...) => erreur "Kernel too old"
- Choix d'une libC => Glibc, uClibc, Eglibc, ...
- GDB
- Toute autre bibliothèque utilisée => libstdc++
- Dépendances avec le compilateur hôte

Toolchain

Interaction entre la libC et le noyau Linux

- Appels systèmes (nombre, définition)
- Constantes
- Structures de données, etc.

Compiler la libC – et certaines applications - nécessite les en-tête du noyau
Disponibles dans <linux/...> et <asm/...> et d'autres répertoires des sources du noyau (include, ...)

Toolchain
compilateur binaire

Utiliser un compilateur binaire :

- ELDK: http://www.denx.de/wiki/DULG/ELDK
- Code Sourcery : https://sourcery.mentor.com/sgpp/lite/arm/portal/release1803
- Installation simple
- Support (payant) possible
- Configuration connue => support par les forums

Par contre:

- Versions des composants figées
- Non utilisation des possibilitées du CPU
- Choix libC limité

Toolchain
Compilé son compilateur

Construire un compilateur:

- Crosstool => obsolète
- Crosstool-NG => assez complexe à prendre en main
- Buildroot / OpenEmbedded

Aucun n'est "plug and play"
La mise au point peut prendre des jours, voire plus !
Binaires produits : arm-linux-* (gcc, as, ld, ar, nm, ...)

Bootloader

Premier logiciel lancé au démarrage de la machine. Initialise le matériel (RAM, stockage, ...) nécésaire au boot. Charge le noyau en RAM et lui donne la main. Permet de donner des arguments au noyau. Exemples:

- U-boot (supporte de nombreux architectures, la référence)
- barebox (u-boot v2, meilleure archi, moins de fonctionnalités pour l'instant)
- grub (x86, serveur et desktop principalement)
- isolinux (alternative x86 à grub)

Un système embarqué nécéssitera une configuration et un support matériel dans le bootloader.

Kernel

Génération d'un noyau:

1. Récupération du noyau (kernel.org)
2. Développement du support des drivers manquants.
3. Création du support de la carte (board config 3.10 device tree)
4. Configuration le noyau (make menuconfig)
5. Compilation du noyau (make uImage)
6. Installation sur la cible

Binaire utilisable dans arch/arm/boot/zImage ou bien arch/arm/boot/uImage (pour U-Boot)
Binaire vmlinux utile pour le debug
Utilisation des variables ARCH et CROSS_COMPILE.

Binaire de base

Binaire de base indispensable au système (ls, bash, dd, top, cp, mv, init,...) Trois possibilités:

- Récupérer les sources et les compiler un par un. (laborieux)
- GNU/Linux coreutils (trop lourd pour l'embarqué)
- Busybox (regroupe la majorité des commandes Linux en un seul executable) 95% des distributions Linux embarqués l'utilise
  Simple, léger, portable
  Diffusé sous licence GPLv2

Busybox

Utilisation des variables ARCH et CROSS_COMPILE. (comme le noyau)
Génération de busybox:

1. Récupération des sources
2. Configuration (make menuconfig)
3. Compilation (make uImage)
4. Installation sur la cible (make CONFIG_PREFIX=... install)

Rootfs
Peuplement

Peuplement d'un rootfs:

1. Installation de la libc et des autres librairies fournies par la toolchain
2. Installation des binaires de busybox
3. Création de la configuration de démarrage (/etc/init.d)
4. Ajout des nodes dans /dev (si pas de mécanisme automatique)
5. Ajout des librairies et applications du projet.

Deux mèthodes:

- A la main, copier ou installer chaque fichier/logiciel manuellement. (LFS)
  Inenvisageable à moyen et long terme
- Automatiser soit en scriptant soit en utilisant un système existant (OE, yocto, buildroot, uclinux ...)

BSP

Une plateforme "neuve" nécéssite des ajustements pou fonctionner:

- Adaptation de la toolchain
- Support des drivers dans le bootloader et le kernel
- Ajout des fonctionnalités dans le bootloader et le kernel (réseau, NAND, HDD, ...)

Fourni par le constructeur de la plateforme éléctronique. Si on est le constructeur, il faut développer le support. Outils à disposition (gdb, sonde JTAG, ftrace, oscilloscope, multimetre ...)

User space

Lorsque le BSP est OK, la mise au point en espace utilisateur peut commencer.
De nombreux outils permettent de mettre au point des applications:

- printf: outil simple à mettre en oeuvre et connu de tout le monde
- syslog: utile pour vérifier le bon fonctionnement ou detecter des anomalies
- valgrind: vérifie la bonne utilisation de la mémoire
- strace/ltrace: affiche les appels systèmes et librairies
- gdb: debugguer GNU/LINUX
  permet d'acceder à la memoire, controler l'execution, monitoré les threads ...
  gdbserver pour les cibles embarqués pour déporter le debug