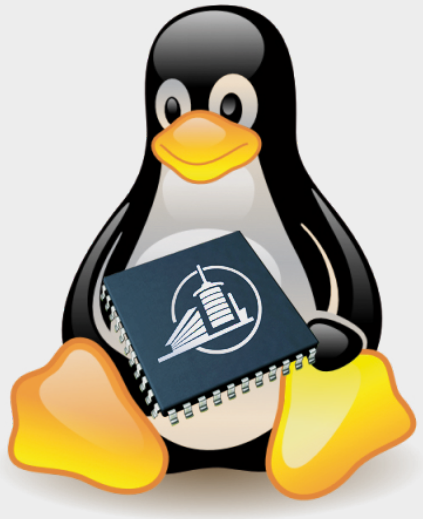


Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

MA-CSEL1 – Construction Systèmes Embarqués sous Linux

Environnement Linux embarqué

HES-SO//Master TIC/TIN 2020-21

Daniel Gachet – HEIA-FR – Télécommunications

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz

Contenu

- Architecture générale de systèmes embarqués pour Linux
- Infrastructure HEIA-FR
- Environnements de développement
- Debugging d'applications
- Boot-loaders
- Systèmes de fichiers
- Distributions Linux embarqués
- Mise en production d'une cible
- Chaîne d'outils GNU

MASTER OF SCIENCE IN ENGINEERING

MSE

Architecture générale de systèmes embarqués sous Linux

MASTER OF SCIENCE IN ENGINEERING

MSE

Architecture générale d'un système Linux embarqué

Machine de développement

chaîne d'outils de développement: compilateur, debugger, ...

Système embarqué

Application 2

Application 1

Application 3

Bibliothèque A | Bibliothèque B | Bibliothèque C

Bibliothèques standards (C, Posix)

Noyau Linux

Amorceur / Boot-loader

Hardware

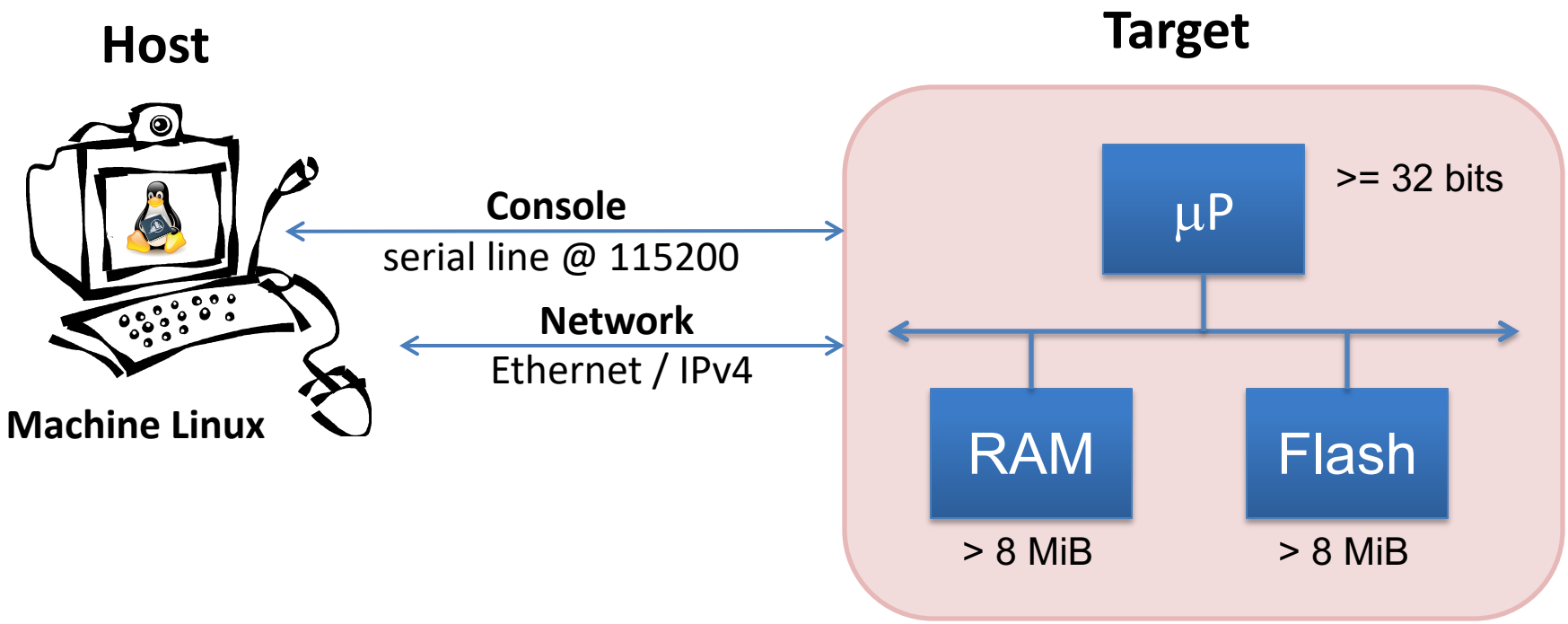
MASTER OF SCIENCE IN ENGINEERING

MSE

Composants du système Linux embarqué

- **Applications**
  - Applications logicielles implémentant la fonctionnalité spécifique du système
- **Bibliothèques**
  - Boîtes à outils logiciels propriétaires ou libres (open-source) offrant toute une gamme de fonctions facilitant le développement d'applications
- **Bibliothèques standard (C, Posix, …)**
  - Interface standardisée entre les services du noyau Linux et les applications fonctionnant dans l'espace utilisateur
- **Noyau Linux**
  - Système d'exploitation contenant des fonctions de gestion des processus, de mémoire, suites de piles de protocoles réseau, pilotes de périphériques et fournissant une large palette de services pour les applications logicielles dans l'espace utilisateur
- **Boot-loader**
  - Logiciel démarré par le processeur, responsable d'initialiser le matériel et d'amorcer le système d'exploitation Linux
- **Hardware**
  - Infrastructure électronique générique ou spécifique du système embarqué

Infrastructure hardware minimale d'un système Linux embarqué



MASTER OF SCIENCE IN ENGINEERING

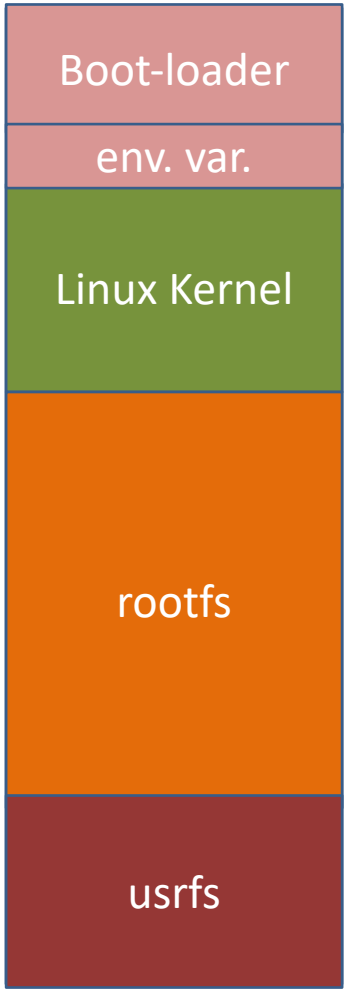
MSE

Structure d'un système Linux embarqué

Un système embarqué sous Linux est constitué de trois éléments principaux stockés dans la mémoire Flash de la carte:

- **Boot-loader (p.ex. U-Boot)**
  - Application en charge d'initialiser le système (horloges, mémoires, chip select logic, …)
  - Application permettant de gérer et de booter le système d'exploitation Linux et ses applications
  - Quelques secteurs de la flash sont réservés pour stocker des variables d'environnement/de configuration du boot-loader
- **Linux Kernel**
  - Noyau Linux exécutable souvent sous une forme compressée (zImage → uImage pour l'U-Boot)
  - Configuration du noyau Linux (Flattened Device Tree / FDT)
- **Linux root file system (rootfs)**
  - Système de fichiers principal (root *file system*) contenant les répertoires et logiciels de base du système Linux ainsi que les logiciels d'applications.
- **Linux user file system (usrfs)**
  - Suivant les applications, le rootfs peut être étendu avec d'autres partitions (usrfs)

Flash organisation

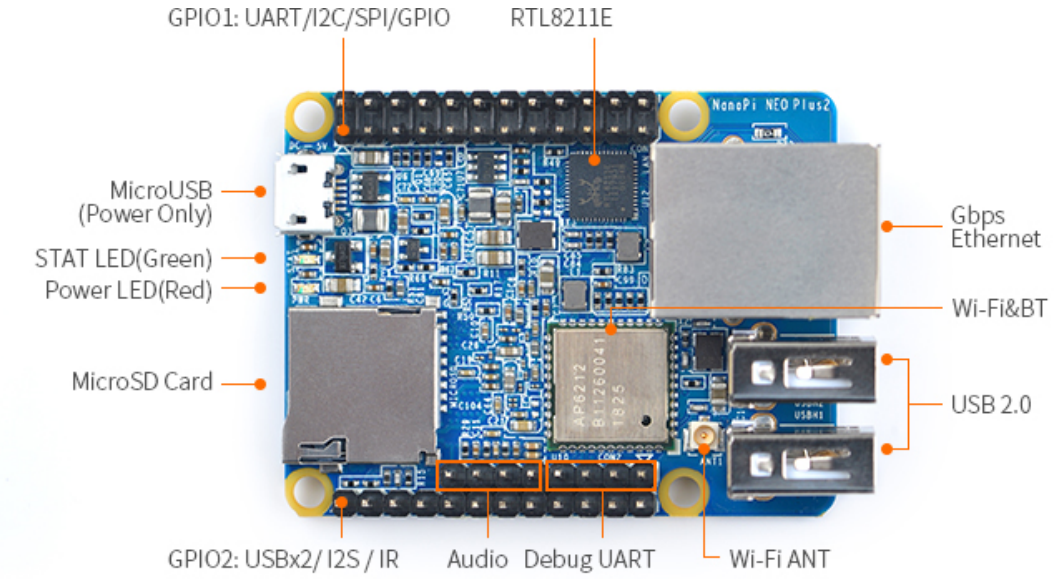


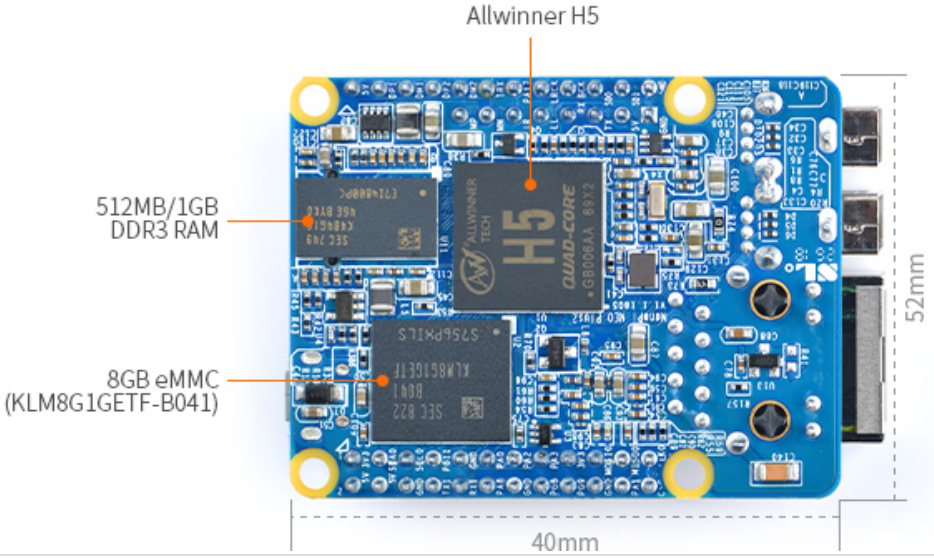
Infrastructure HEIA-FR

MASTER OF SCIENCE IN ENGINEERING

MSE

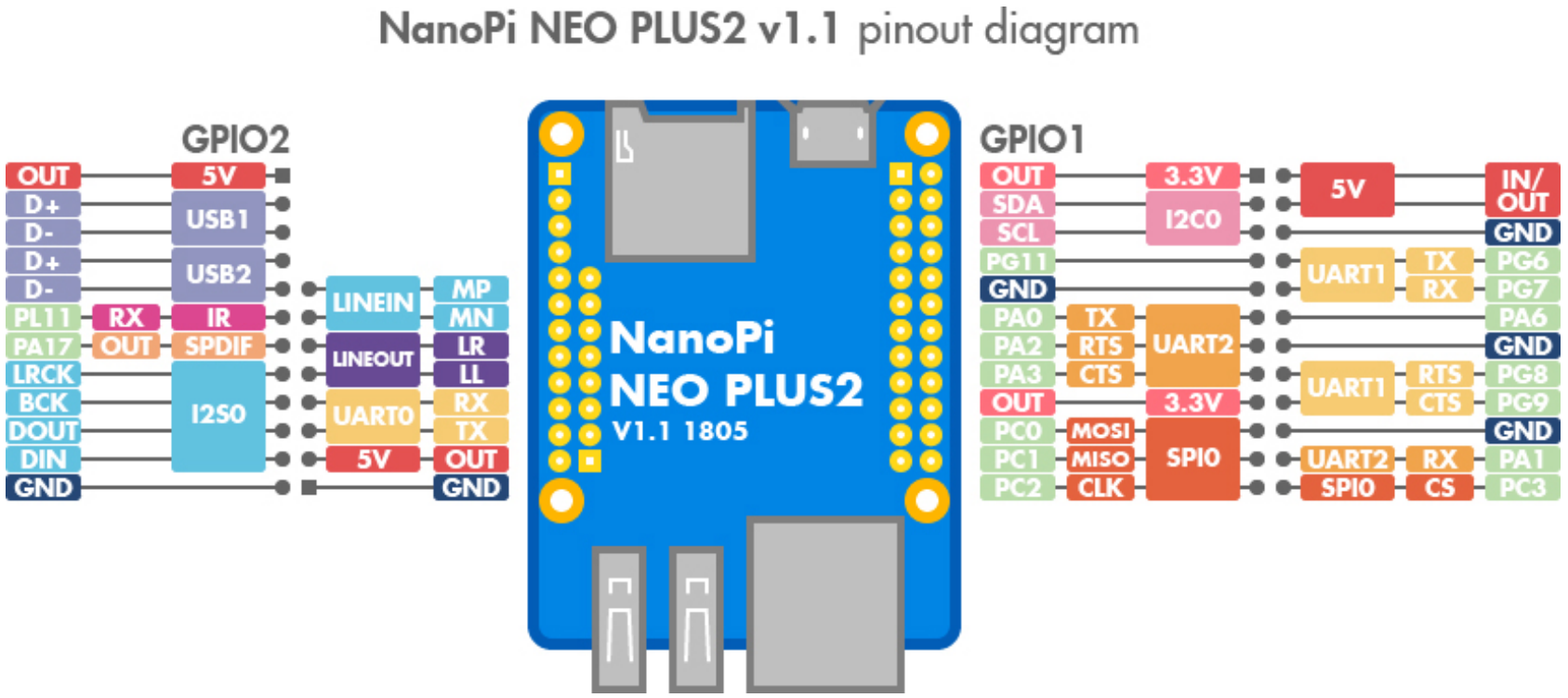
NanoPi NEO Plus2





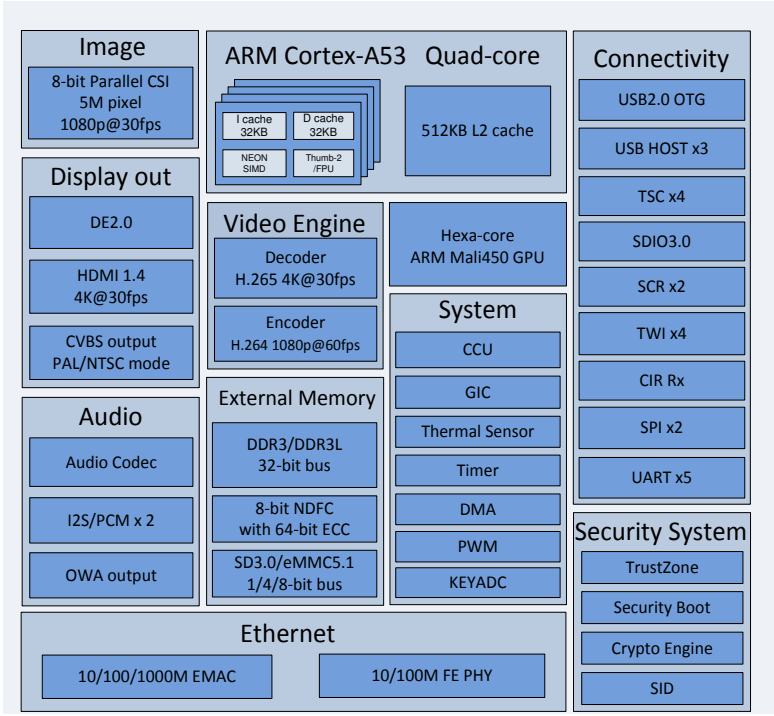
Ref.: http://wiki.friendlyarm.com/wiki/index.php/NanoPi_NEO_Plus2

NanoPi NEO Plus2 – Pinout



NanoPi NEO Plus2 – Main Characteristics

- SoC: Allwinner H5, Quad-core 64-bit high-performance Cortex A53
- DDR3 RAM:1GB
- Storage: 8GB eMMC
- Network: 10/100/1000M Ethernet based on RTL8211E-VB-CG
- WiFi: 802.11b/g/n
- Bluetooth: 4.0 dual mode
- MicroSD: 1 x slot supporting system booting
- Audio Input/Output: 5 pins
- MicroUSB: power input
- Debug Serial: 4 pins
- GPIO1: 24 pins (UART, SPI, I2C and IO)
- GPIO2: 12 pins (USB, IR receiver, I2S and IO)
- Power Supply: DC 5V/2A
- PCB Dimension: 40 x 52mm
- PCB Layer: 6-Layer



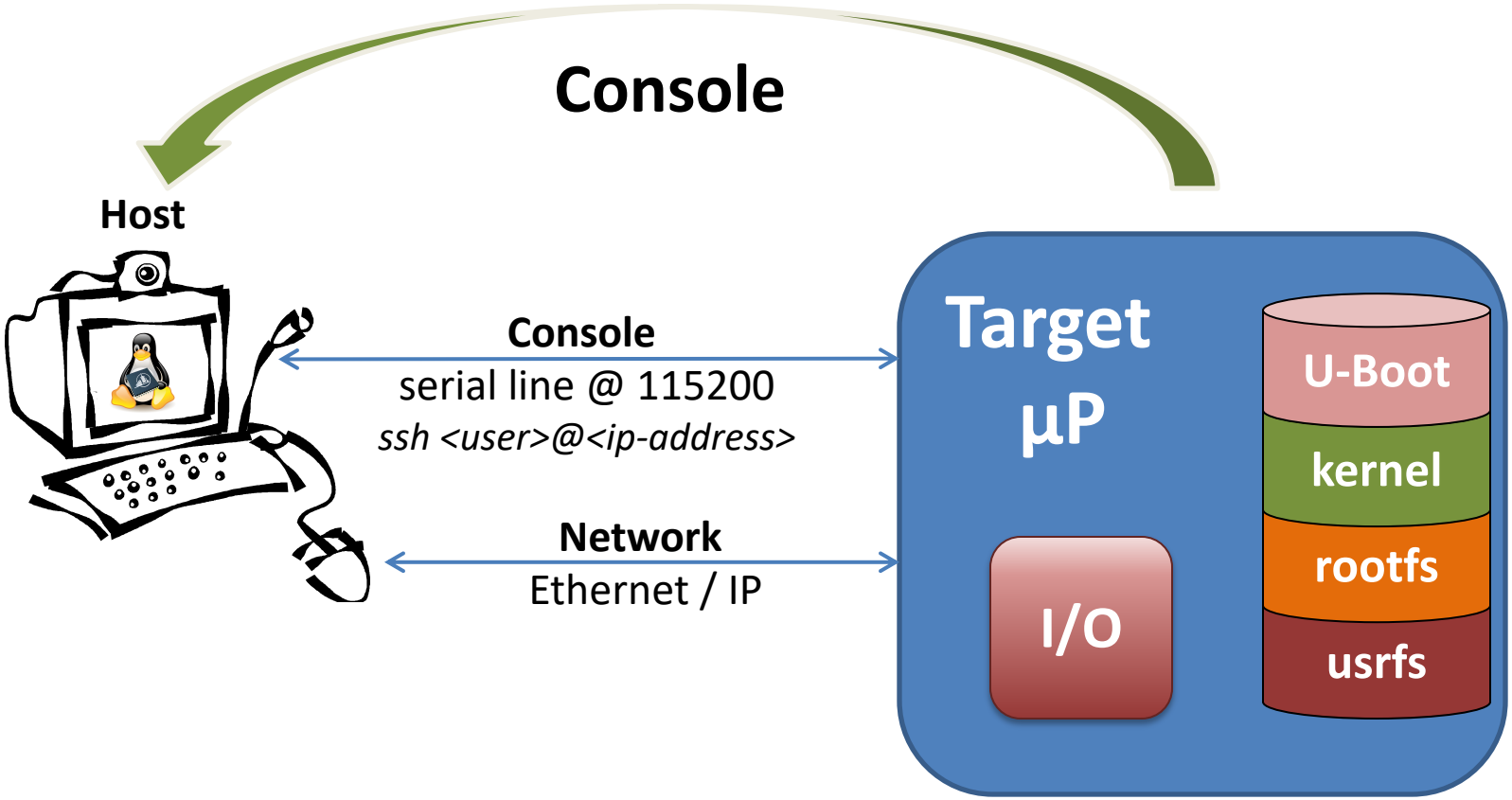
Ref.: http://wiki.friendlyarm.com/wiki/images/d/de/Allwinner_H5_Datasheet_V1.0.pdf

Environnement de développement

MASTER OF SCIENCE IN ENGINEERING

MSE

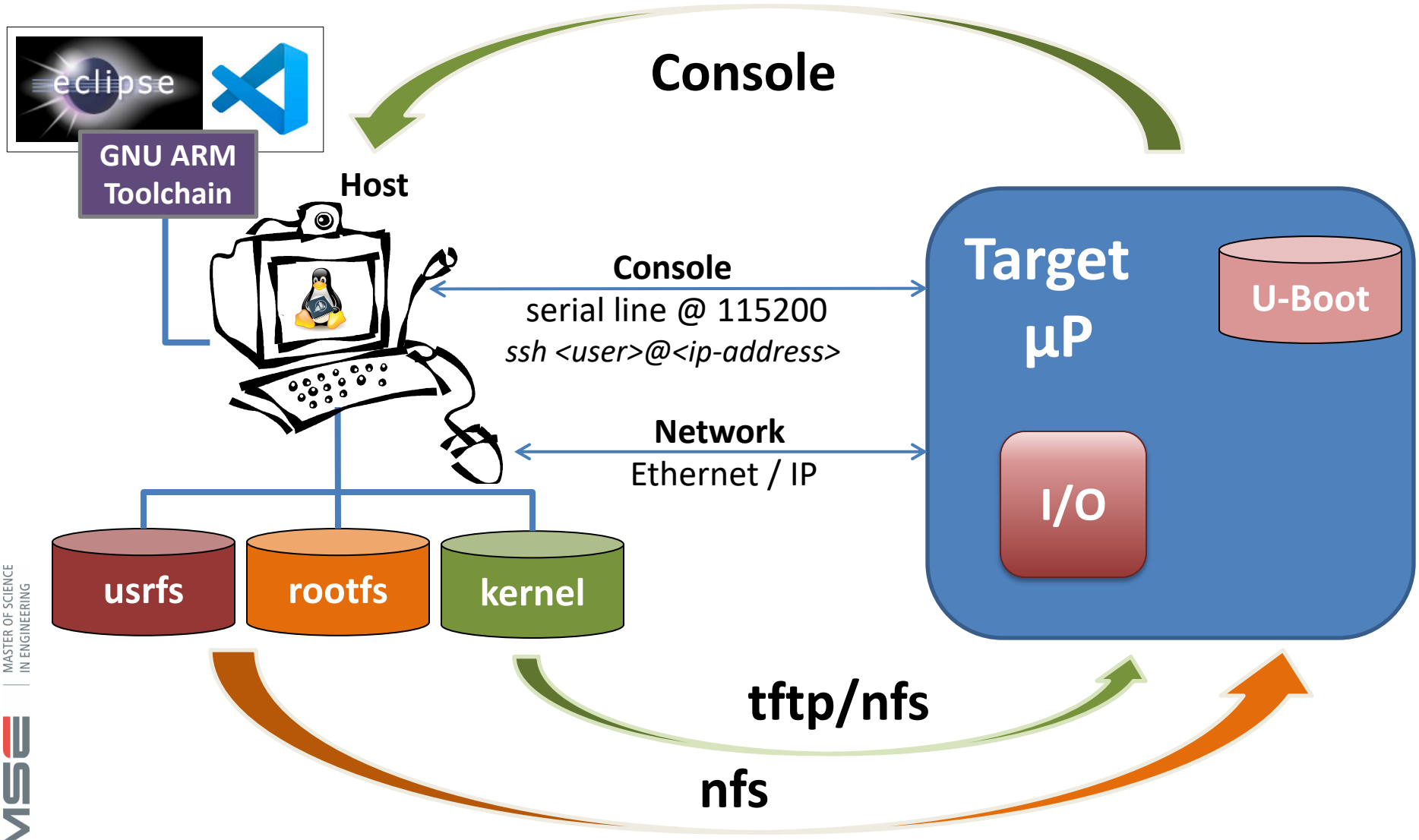
Environnement de production sous Linux



MASTER OF SCIENCE IN ENGINEERING

MSE

Environnement pour le développement du noyau Linux



Outils de développement

- **Une multitude d'outils sont disposition pour le développement de logiciel sous Linux. Dans le cadre de ce module, nous avons fait le choix d'utiliser que des outils libres « open source ».**
  - Pour la génération du noyau Linux, des modules et des applications, nous utiliserons la chaîne d'outils de GNU
  - Pour le debugging d'applications, nous utiliserons le debugger GDB de GNU, mais afin d'obtenir une interface graphique selon l'état de l'art nous emploierons soit le VS-Code, soit l'IDE d'Eclipse

Bootloader

Rôle d'un bootloader

- **Les bootloaders ont généralement trois fonctions principales:**
  - Initialisation du processeur
    - *Après la mise sous tension du processeur, certaines composantes matérielles doivent impérativement être initialisées avant qu'un programme, même élémentaire, puisse s'exécuter. Les horloges, la RAM et son contrôleur en sont des exemples typiques.*
    - *Cette initialisation est toujours dépendante du type de processeur et de ses périphériques. Elle est généralement développée en assembleur. On n'initialisera naturellement que les périphériques nécessaires au fonctionnement du bootloader.*
  - Lancement de l'application
    - *Sur les systèmes embarqués, les applications sont généralement stockées dans des mémoires non volatiles de type flash. Cependant, pour être exécutées celles-ci doivent souvent être chargées dans la RAM.*
  - Mise à jour de l'application
    - *Certains bootloader offrent la possibilité de mettre à jour les applications chargées sur le système. Celle-ci peut s'effectuer soit localement (carte SD, interface série, …) ou à distance par l'intermédiaire de réseaux de communication (Ethernet, WiFi, …)*

Les bootloaders

- **Sur le marché, il existe un grand nombre de bootloaders commerciaux ou open source. Ces boot-loader sont souvent spécialisés soit pour un type de processeur, p. ex. en open source :**
  - Architecture Intel / PC
    - *GRUB (GRand Unified Bootloader)*
    - *LILO (Linux loader)*
    - *RedBoot (Red Hat Embedded Debug and Bootstrap firmware)*
    - *U-Boot (Universal Bootloader)*
    - *…*
  - Architecture MIPS
    - *RedBoot*
    - *U-Boot*
    - *…*
  - Architecture ARM
    - *RedBoot*
    - *U-Boot*
    - *…*

MASTER OF SCIENCE IN ENGINEERING

MSE

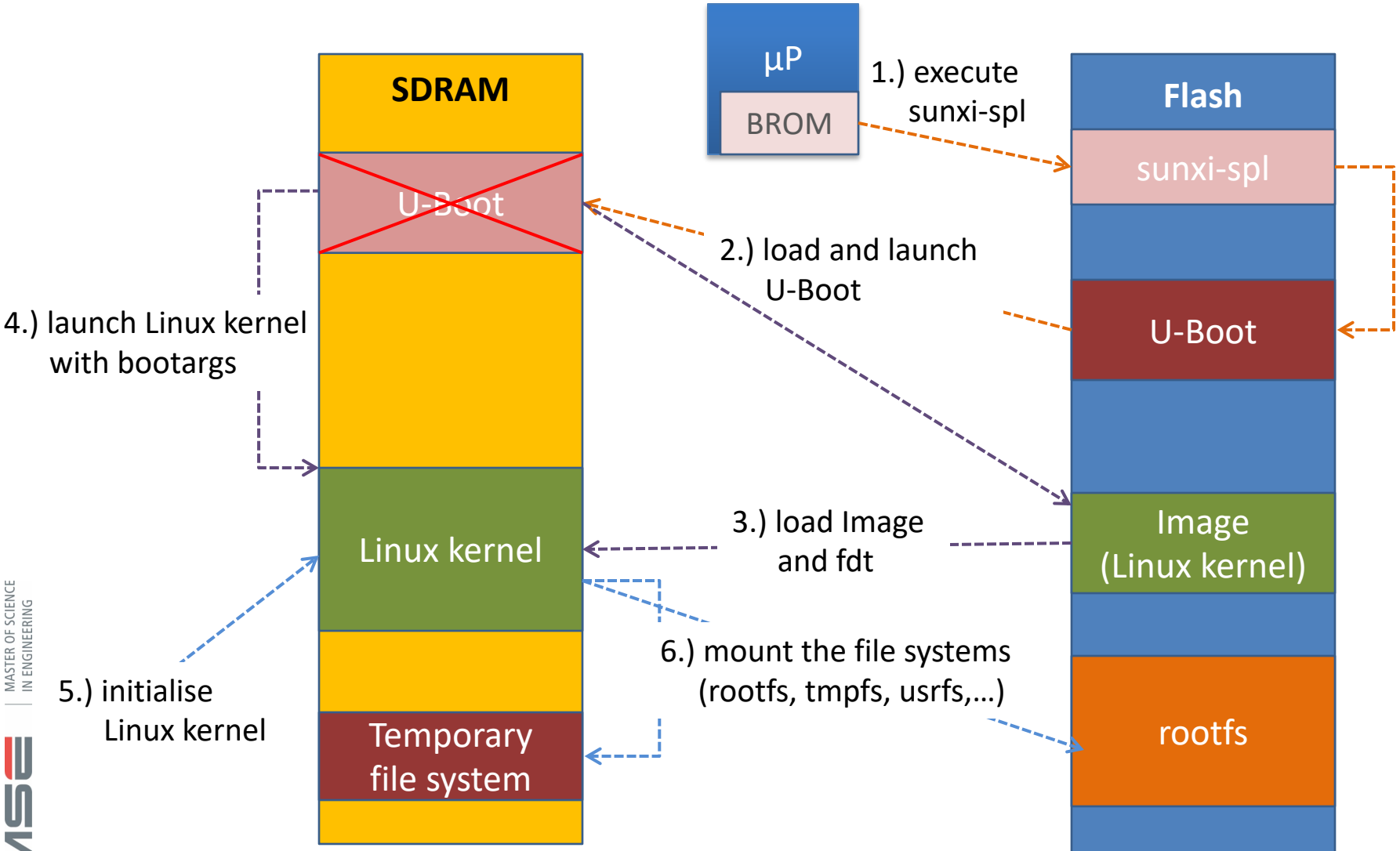
Choix d'un bootloader

- **Le choix d'un bootloader adapté à vos besoins est une tâche ardue. Ci-dessous quelques facteurs à considérer lors du choix :**
  - Est-ce qu'il supporte mon choix de processeur ?
  - A-t-il déjà été porté sur une carte similaire à la mienne ?
  - Est-ce qu'il supporte les fonctions dont j'ai besoin ?
  - Est-ce qu'il supporte les périphériques que j'ai prévu d'utiliser ?
  - Existe-t-il une large communauté d'utilisateurs d'où je pourrai obtenir de l'aide ?
  - Existe-t-il des fournisseurs commerciaux dont je pourrai acheter du support ?

Das U-Boot

- **Dans les systèmes embarqués, «Das U-Boot» est probablement l'un des bootloaders des plus populaires.**
- **Il est développé et maintenu par la maison DENX Software Engineering à Munich (www.denx.de).**
- **La première version a été libérée en juillet 2000 pour des processeurs PowerPPC. Aujourd'hui, il supporte une très large palette de processeurs et offre une grande fonctionnalité .**
- **Quelques caractéristiques:**
  - Processeurs: PPC, ARM, MIPS, x86, m68k, …
  - Systèmes opératifs: Linux, VxWorks, FreeBSD, LynxOS, …
  - Interfaces: série, Ethernet, flash card
  - Protocoles: tftp, bootp/tftp, rarp/tftp, dhcp
  - Systèmes de fichiers: JFFS2, UBI/UBIFS, ext4, …
- **Source code**
  - Licence GPLv2
  - Langage: développé en C (sans OS ni interruptions)
  - Taille de l'exécutable: ~300KiB
  - Release: tous les 2 à 3 mois

Séquence de démarrage du NanoPi NEO Plus2 – Carte SD / eMMC



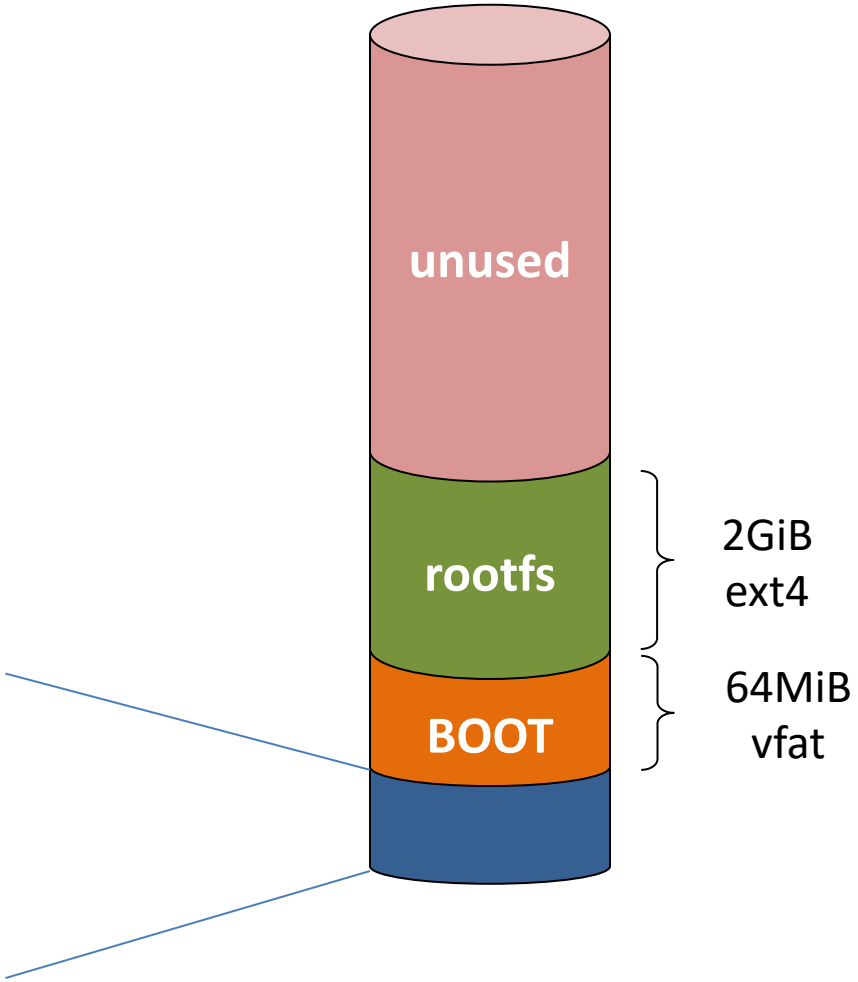
Séquence de démarrage du NanoPi NEO Plus2 – Carte SD / eMMC (II)

- **Le démarrage du NanoPi NEO Plus2 se décompose en 6 phases:**
  - Lorsque le µP est mis sous tension, le code stocké dans son BROM va charger dans ses 32KiB de SRAM interne le firmware « sunxi-spl » stocké dans le secteur n° 16 de la carte SD / eMMC et l'exécuter.
  - Le firmware « sunxi-spl » (Secondary Program Loader) initialise les couches basses du µP, puis charge l'U-Boot dans la RAM du µP avant de le lancer.
  - L'U-Boot va effectuer les initialisations hardware nécessaires (horloges, contrôleurs, …) avant de charger l'image non compressées du noyau Linux dans la RAM, le fichier «Image», ainsi que le fichier de configuration FDT (flattened device tree).
  - L'U-Boot lancera le noyau Linux en lui passant les arguments de boot (bootargs).
  - Le noyau Linux procédera à son initialisation sur la base des bootargs et des éléments de configuration contenus dans le fichier FDT (sun50i-h5-nanopi-neo-plus2.dtb).
  - Le noyau Linux attachera les systèmes de fichiers (rootfs, tmpfs, usrfs, …) et poursuivra son exécution.

Ref. documentation sur les bootargs :
- fichier ./Documentation/kernel-parameters.txt dans les sources du noyau Linux
- sp.02.4_mas_cesl_linux_boot_commands_arguments.pdf

Structure de la carte SD du NanoPI NEO Plus2

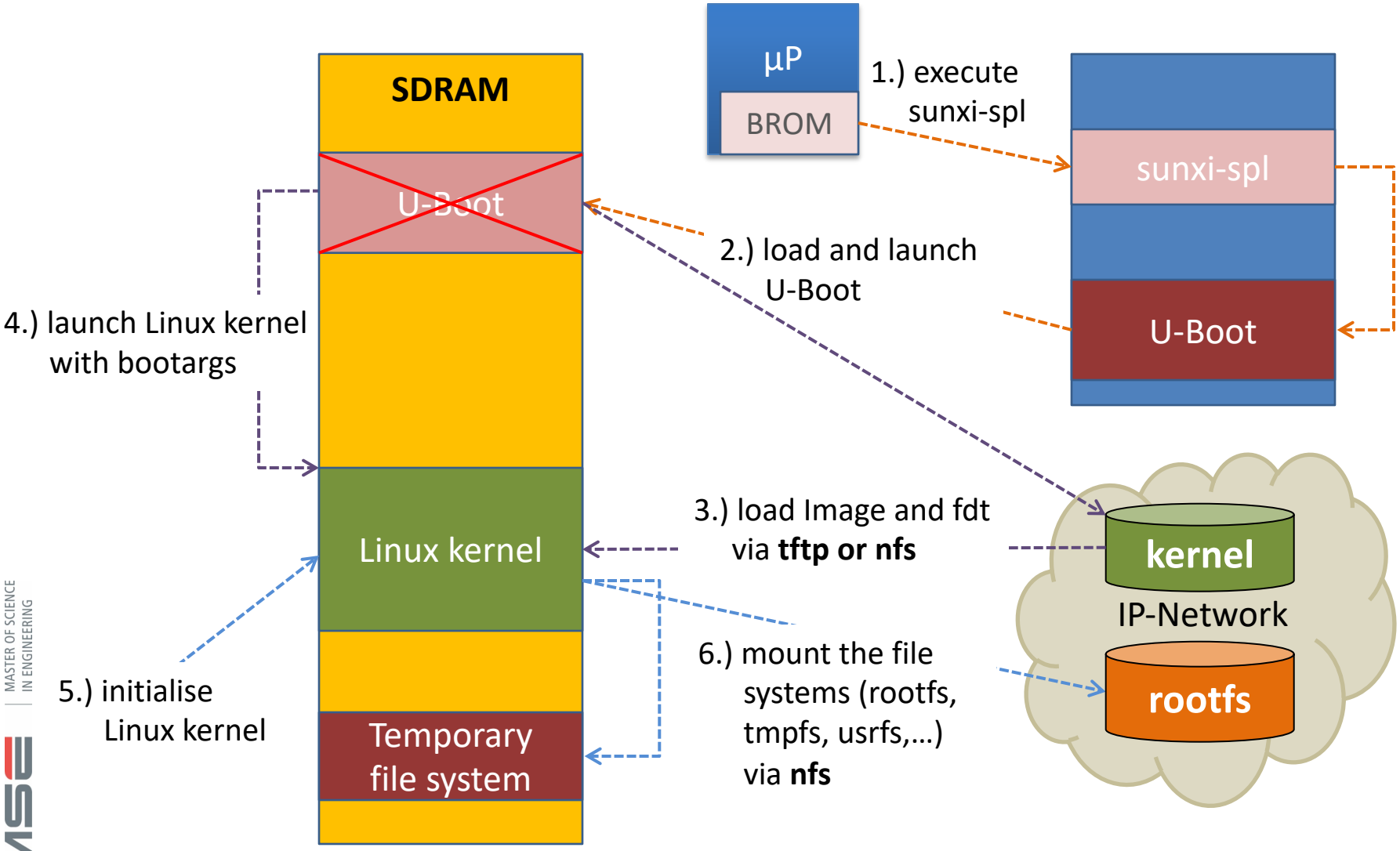
Area name	From sector	Size
U-Boot	80	1MiB
sunxi-spl	16	32KiB
MBR (partition table)	0	512B



Sector size = 512 Bytes

MSE | MASTER OF SCIENCE IN ENGINEERING

Séquence de démarrage du NanoPi NEO Plus2 – Réseau



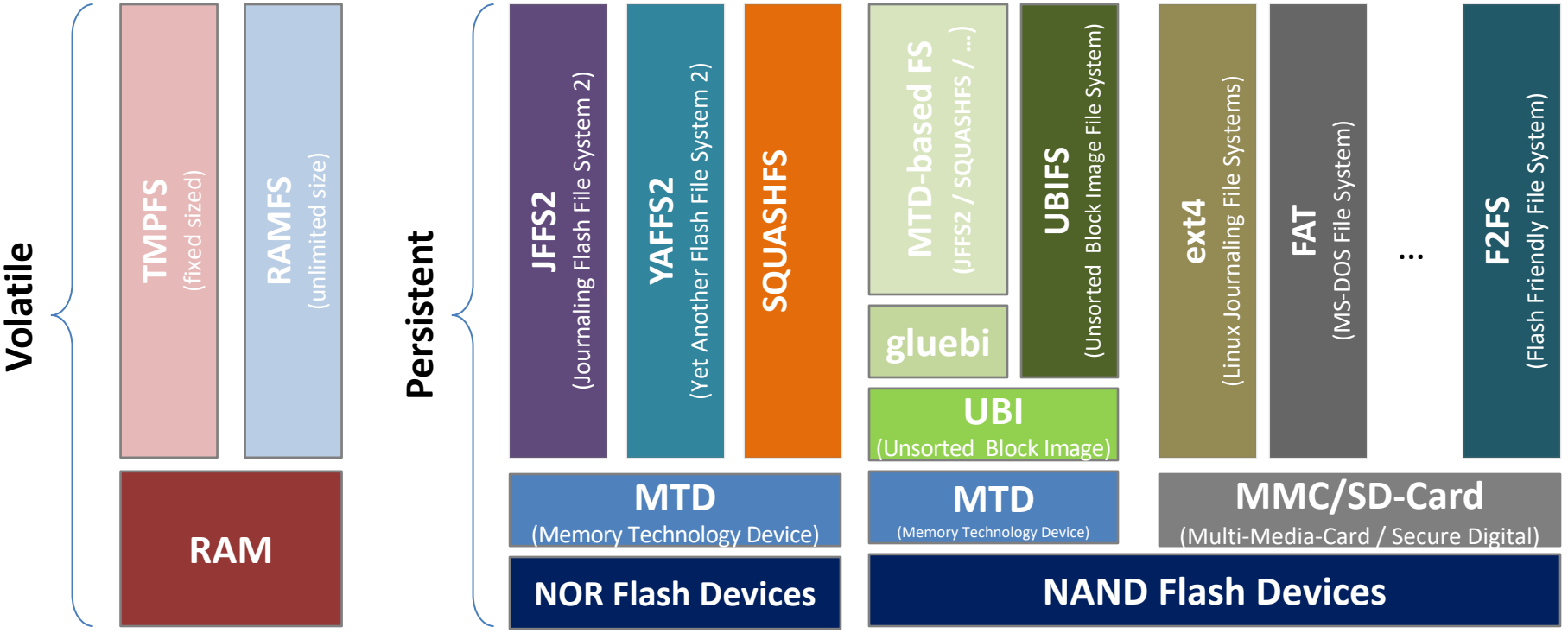
Systèmes de fichiers

MASTER OF SCIENCE IN ENGINEERING

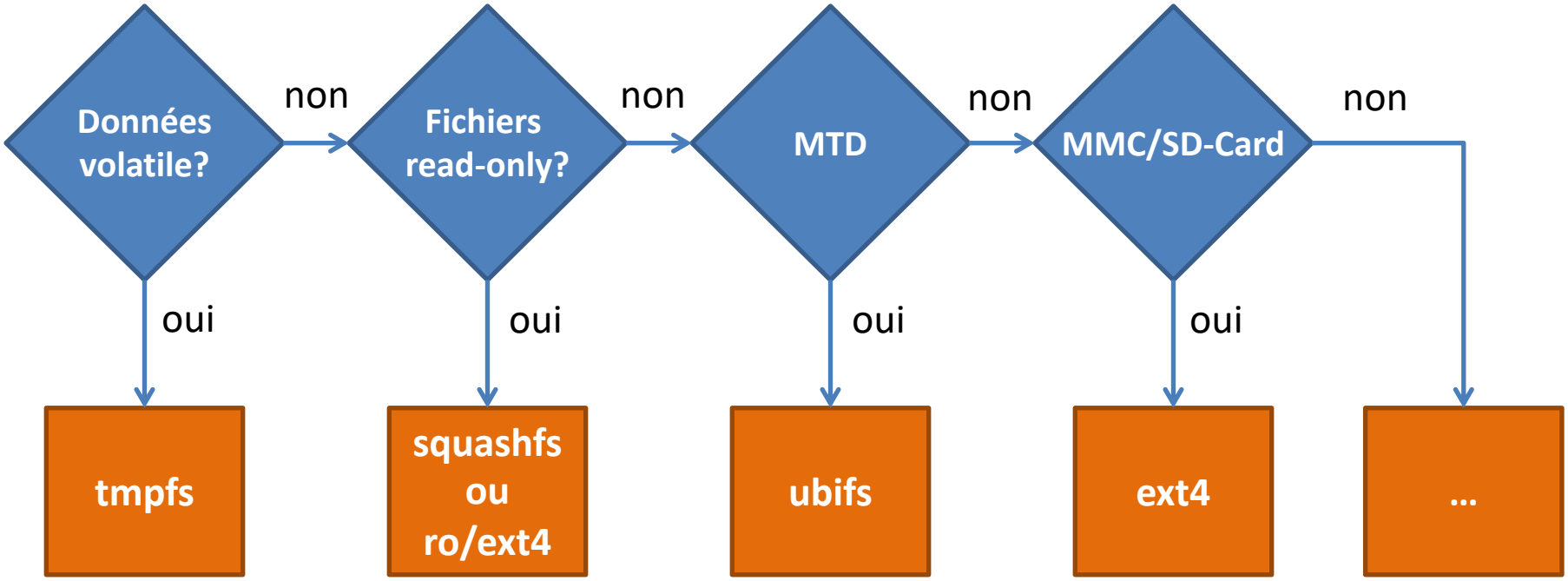
MSE

Systèmes de fichiers

- Il existe une multitude de systèmes de fichiers. Cependant, les systèmes embarqués sont plutôt équipés de mémoires Flash (le plus souvent NAND) et de mémoires RAM.
- Ces mémoires proposent des couches d'abstraction (MTD, MMC/SD-Card), lesquelles permettent de partager la mémoire en plusieurs partitions.
- Chacune de ces partitions peut contenir un autre type de système de fichiers.



Choix d'un système de fichiers



- **Comparaison des différents systèmes de fichiers**
  - Tristan Lelong a fait une analyse très intéressante, pour les détails: sp.02.5_mas_csel_filesystem_considerations_for_embedded_devices.pdf
  - Voir la documentation du noyau Linux pour plus de détails sur les systèmes de fichiers **`Documentation/filesystems/`**

MASTER OF SCIENCE IN ENGINEERING

MSE

Debugging d'une application

Technique de debugging

- **Il existe une multitude de chemins pour debugger une application:**
  - Try and Error
  - Revue de code (code walk through)
  - Traçage (Tracing: printf, syslog, 7-segment, …)
  - Debugging à l'aide d'un debugger
    - *Ligne de commande, p. ex. GDB (GNU Debugger)*
    - *Graphique, p. ex. DDD (Data Display Debugger) ou Eclipse*

- **Il est important de choisir le bon outil, en fonction du problème à analyser. Le traçage est probablement le plus simple et le plus facile à mettre en œuvre. Les debuggers apportent un grand confort lors du développement, mais il faut savoir qu'ils peuvent perturber le comportement du code et qu'il est souvent difficile de les utiliser lorsque les cibles ont été déployées chez les clients.**

Tracing

- **Technique très simple, il suffit d'ajouter des «printf» à l'intérieur du code aux endroits sensibles. Il permet d'afficher l'état de variables** *(…/fibonacci)*

```
int value = 100;
if (/*condition*/) {
  value++;
  printf ("if statement\n");
} else {
  printf ("else statement\n");
}
printf ("Value decimal:%d, hex:0x%x\n", value, value);
```

- **Cette technique peut être amélioré avec une compilation conditionnelle** *(…/tracing)*

```
#ifdef DEBUG
#define TRACE(x) printf x
#else
#define TRACE(X)
#endif

TRACE(("Value decimal:%d, hex:0x%x\n", value, value));
```

- **D'autres améliorations sont encore possible: log-file, exécution conditionnelle, syslog, etc.**

MASTER OF SCIENCE IN ENGINEERING

MSE

Debugging avec GDB

- Le debugger GNU, généralement appelée simplement GDB, est le debugger standard pour le système GNU, et par conséquent pour Linux. Il s'agit d'un debugger portable qui fonctionne sur de nombreux systèmes similaires à Unix et fonctionne pour de nombreux langages de programmation, tel qu'Ada, C/C++, FreeBasic ou Fortran.
- Démarrer le programme avec GDB

```
gdb <program_with_symbols>
```

- Ce type de debugging peut être exécuté indifféremment sur la cible ou sur la machine hôte.
- Note: le programme doit être compilé avec l'option de debugging (`gcc -g`).

Debugging avec GDB – quelques commandes

- **Commandes de base**
  - `list` - show the program code
  - `break` - set breakpoint (to address or source-line no)
  - `del break` - delete a breakpoint
  - `run` - start program
  - `continue` - continue execution after breakpoint
  - `step` - step (into) instruction
  - `next` - step over instruction
  - `print` - show variable contents
  - `x/i` - display memory content as instruction
  - `bt` - backtrace: show execution stack
  - `frame` - select a stack frame
  - `quit` - leave the debugger

- **Quelques guides**

  http://visualgdb.com/gdbreference/commands/

  https://cs.baylor.edu/~donahoo/tools/gdb/tutorial.html

MASTER OF SCIENCE IN ENGINEERING

MSE

Debugging à distance avec GDB / GDB Server

- **GDB propose un mode distance (remote), utilisé principalement lors de debugging de systèmes embarqués. On parle de debugging à distance, lorsque GDB fonctionne sur la machine hôte et le programme étant debuggé sur la cible. Dans ce cas, GDB communiquera avec la cible soit par une interface série, soit par une interface Ethernet/IP via le protocole série ou TCP/IP.**
- **Démarrer une session de debugging à distance est également simple**
  - Sur la machine cible

```
# gdbserver <[hostname]:port> <program_without_symbols>
```

  - Sur la machine hôte

```
$ <path>/<target>-gdb <program_with_symbols>
> set sysroot <shared_library_path>
> target remote <targetname:port>
> break main
> continue
```

- **Pour le NanoPi NEO Plus2**
  - path (toolchain) ⇔ `~/workspace/nano/buildroot/output/host/usr/bin`
  - target ⇔ `aarch64-none-linux-gnu-`
  - shared_library_path ⇔ `~/workspace/nano/buildroot/output/staging`

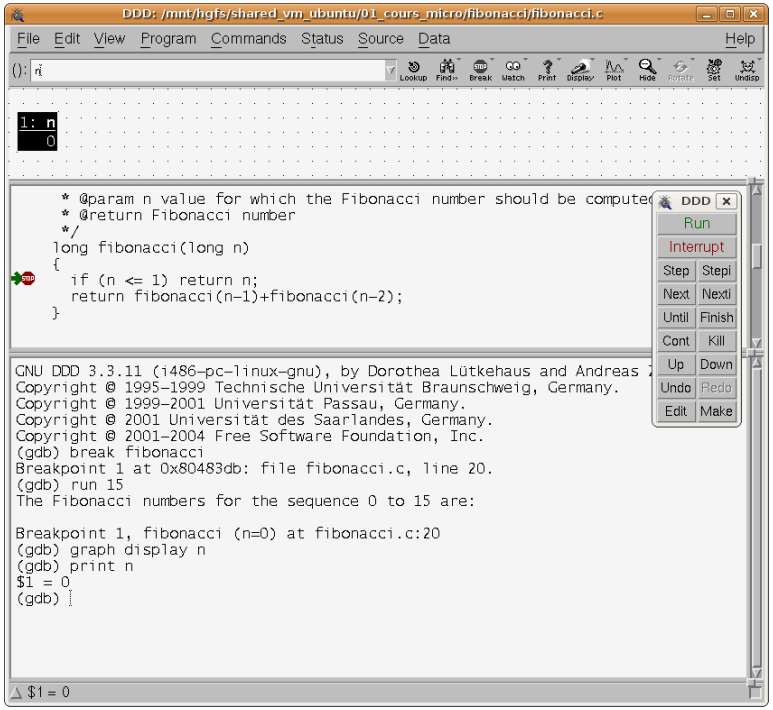
MASTER OF SCIENCE IN ENGINEERING

MSE

Debugging avec DDD

- **Étant donné que GDB ne dispose pas d'interface graphique telle que les IDE, l'utilisation d'un front-end externe est utile. Il en existe beaucoup. DDD en est un, simple et pratique.**
- **Appeler DDD:**

```
ddd <program_with_symbols>
```



- **Au démarrage, DDD affiche plusieurs fenêtres différentes, p. ex. :**
  - Data window
  - Source window
  - GDB console
- **Les commandes GDB peuvent être invoquées soit en pressant sur les boutons dans la fenêtre de commandes, soit en tapant la commande l'intérieur de la console GDB.**

MASTER OF SCIENCE IN ENGINEERING
MSE

Debugging à distance avec DDD

- **Le debugging à distance avec DDD est réalisé de manière similaire au debugging à distance avec GDB utilisant le GDB Server sur la cible.**
- **Démarrer une session de debugging à distance est également simple :**
  - Sur la cible

```
gdbserver <[hostname]:port> <program_without_symbols>
```

  - Sur la machine hôte

```
ddd --debugger <path>/<target>-gdb <program_with_symbols>
> set sysroot <shared library path>
> target remote <targetname:port>
> break main
> continue
```

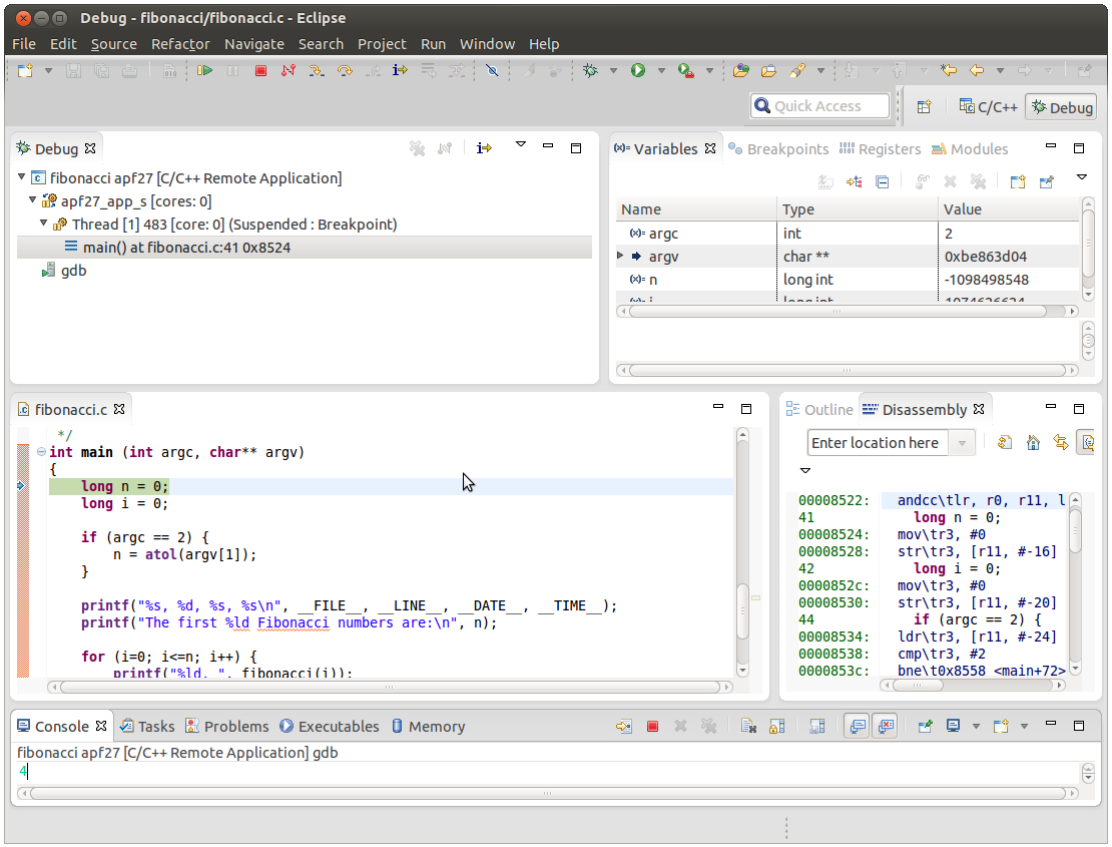
- **Pour le NanoPi NEO Plus2**
  - path (toolchain) ⇔ `~/workspace/nano/buildroot/output/host/usr/bin`
  - target ⇔ `aarch64-none-linux-gnu-`
  - shared_library_path ⇔ `~/workspace/nano/buildroot/output/staging`

MASTER OF SCIENCE IN ENGINEERING

MSE

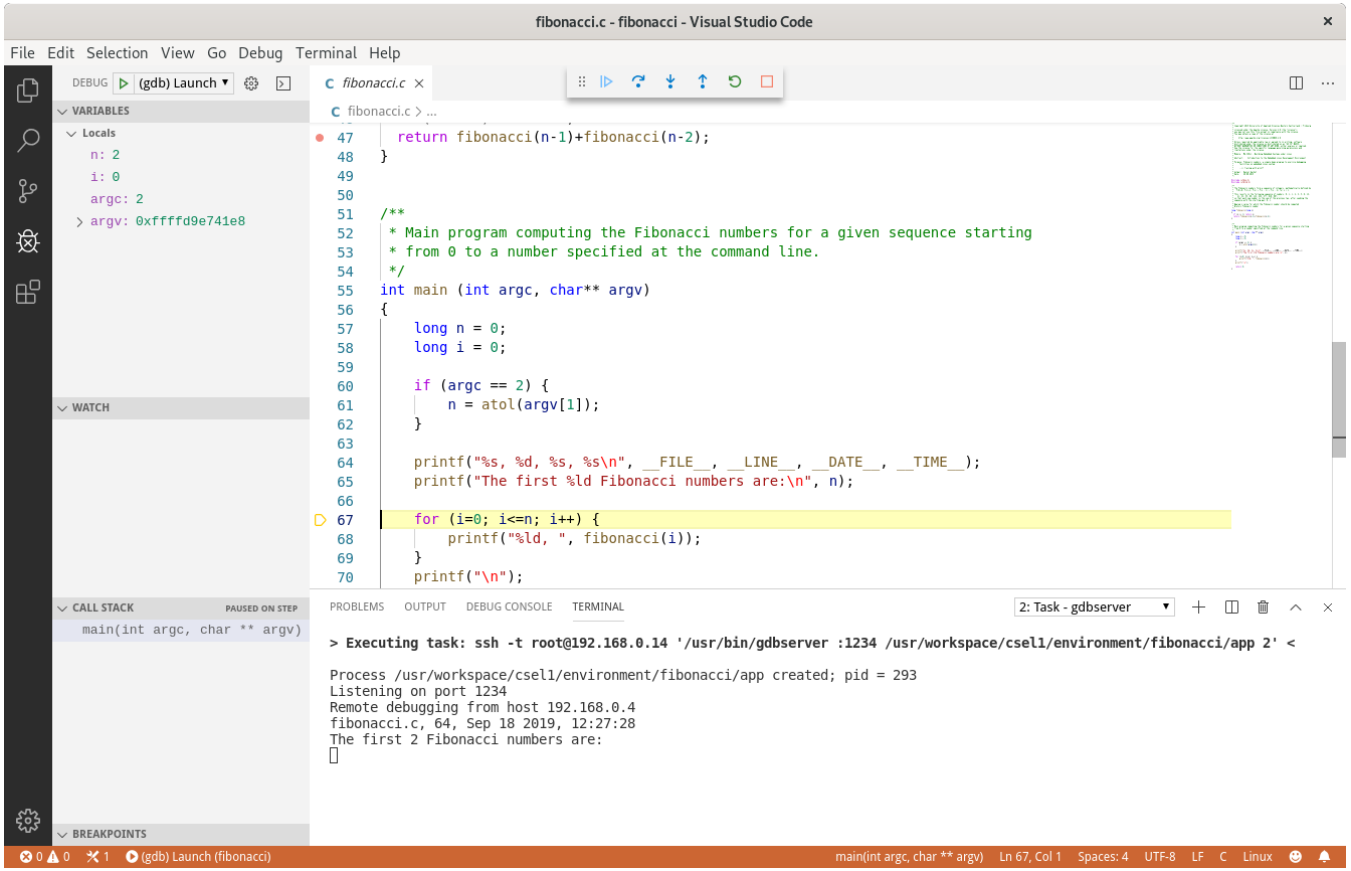
Debugging avec Eclipse

- **Eclipse est un environnement de développement intégré Il propose toute une série d'outils très puissants pour le développement d'applications. Il offre entre autres une interface graphique très performante pour le debugging d'applications native et distante.**

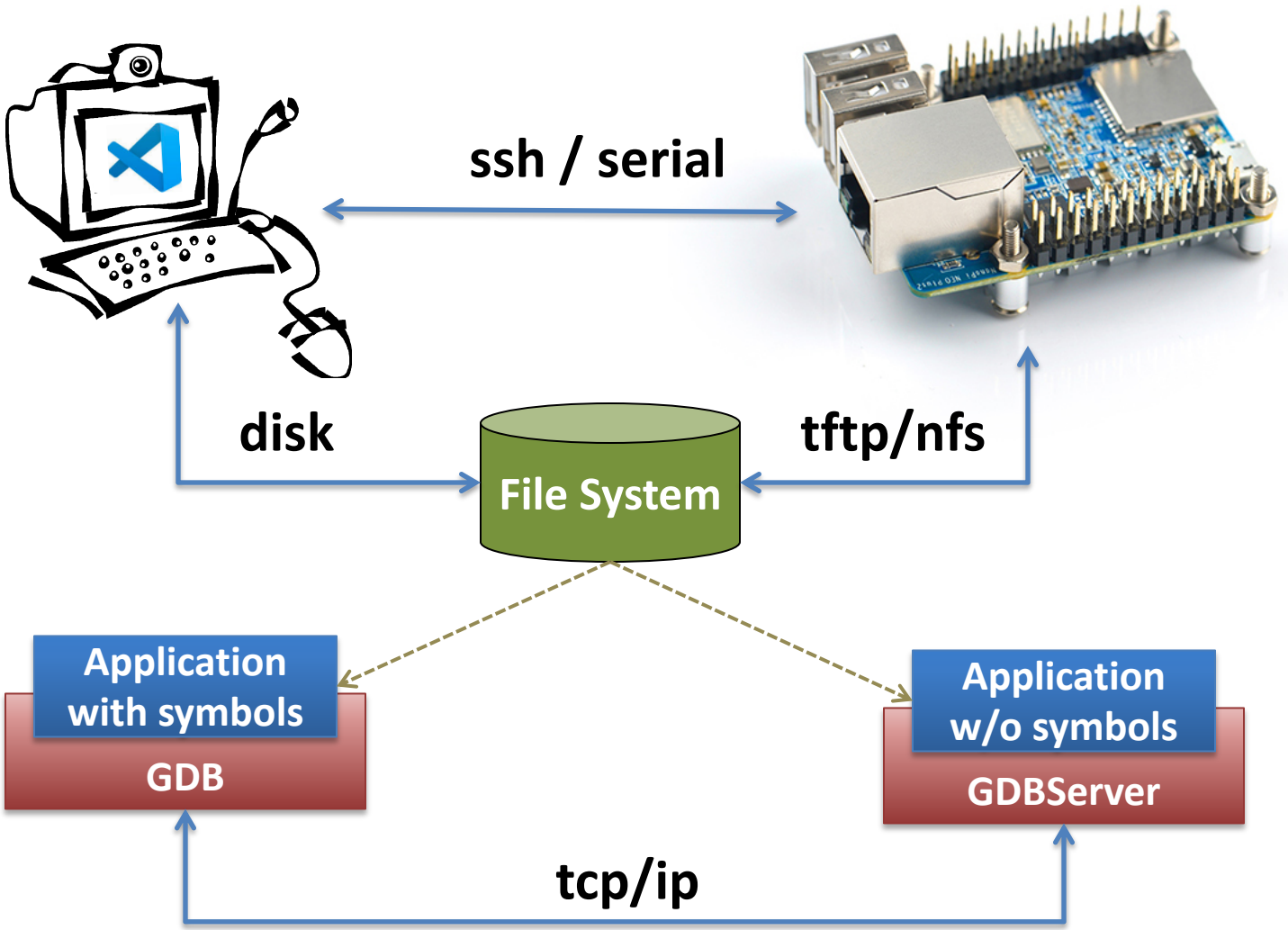


Debugging avec VS-Code

- **VS-Code est un éditeur très puissant. Il propose toute une série d'extensions pour le développement de logiciels. Dans ce catalogue, l'extension «C/C++» offre des services pour le debugging d'applications à distance via GDB, laquelle s'intègre avec le GUI de VS-Code.**



Configuration de l'infrastructure de debugging



MASTER OF SCIENCE IN ENGINEERING

MSE

Core dumps

- **Un « core dump » est un fichier contenant une image de la mémoire du processus à l'instant du crash. Il est généré par le noyau Linux lorsque l'exécution d'un programme lève un signal pour lequel aucune routine de traitement n'a préalablement été enregistrée.**
- **Le core dump est enregistré dans le répertoire et sous le nom de fichier spécifié dans `/proc/sys/kernel/core_pattern`, généralement `"core.<pid>"`**
- **La création de core dumps doit être autorisée. Pour cela, il suffit d'utiliser la commande :**

```
# ulimit -c <size|unlimited>
```

- **Les core dumps sont un désastre pour les applications, mais ils peuvent être une bénédiction dans la recherche du problème pour le développeur. Il suffit d'appeler GDB pour debugger un core dump :**
  - Sur la cible:

```
# gdb <program-name> <core-file>
  > bt (backtrace)
```

  - Sur la machine hôte:

```
$ sudo <path><target>-gdb <program-name> <core-file>
  > bt (backtrace)
```

MASTER OF SCIENCE IN ENGINEERING

MSE

Core dumps (II)

- **Appel du programme** *(…/core_dumps)*

```
# ./app
Segmentation fault (core dumped)
```

- **Informations fournies par le debugger**

```
$ sudo ~/workspace/nano/buildroot/output/host/usr/bin/aarch64-none-linux-gnu-gdb app core
...
Core was generated by `./app'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x0000000000400534 in access_data () at core_dumps.c:31
31                  *p=10;
(gdb) bt
#0  0x0000000000400534 in access_data () at core_dumps.c:31
#1  0x000000000040056c in call (n=0) at core_dumps.c:37
#2  0x0000000000400568 in call (n=1) at core_dumps.c:36
#3  0x0000000000400568 in call (n=2) at core_dumps.c:36
#4  0x0000000000400568 in call (n=3) at core_dumps.c:36
#5  0x0000000000400568 in call (n=4) at core_dumps.c:36
#6  0x0000000000400568 in call (n=5) at core_dumps.c:36
#7  0x0000000000400568 in call (n=6) at core_dumps.c:36
#8  0x0000000000400568 in call (n=7) at core_dumps.c:36
#9  0x0000000000400568 in call (n=8) at core_dumps.c:36
#10 0x0000000000400568 in call (n=9) at core_dumps.c:36
#11 0x0000000000400568 in call (n=10) at core_dumps.c:36
#12 0x0000000000400588 in main () at core_dumps.c:43
(gdb)
```

Backtrace

- Un backtrace est une impression de la pile d'exécution d'un thread ou processus. Il peut être initié à tout moment en appelant les fonctions `backtrace(…)` et `backtrace_symbols_fd(…)`.
- Ces fonctions seront généralement incluses dans la routine de traitement des erreurs de segmentation.
- Lors de l'impression de la pile d'exécution, les adresses absolues correspondant aux de lignes de code des fonctions appelées sont affichées. Pour convertir ces adresses en nom des fichiers sources et numéros de lignes, il suffit d'appeler l'outil `addr2line` :

```
$ addr2line -e <name-of-executable-with-debug-symbols> <absolute-address>
```

Note: ces fonctions ne sont pas disponibles dans toutes les bibliothèques, p.ex. µClib

Backtrace (II)

- **L'exemple de backtrace affiche les infos suivantes lors du crash** *(.../backtrace)*

```
# ./app
backtrace() returned 17 addresses
./app[0x40070c]
linux-vdso.so.1(__kernel_rt_sigreturn+0x0)[0xffffb43f9578]
./app[0x400748]
./app[0x400780]
./app[0x40077c]
./app[0x40077c]
./app[0x40077c]
./app[0x40077c]
./app[0x40077c]
./app[0x40077c]
./app[0x40077c]
./app[0x40077c]
./app[0x40077c]
./app[0x40077c]
./app[0x4007ac]
/lib64/libc.so.6(__libc_start_main+0xe4)[0xffffb4281fac]
./app[0x40062c]
#
```

- **Pour obtenir le nom du fichier et la ligne de code correspondant à une adresse, il suffit d'appeler l'outil addr2line comme suit :**

```
[lmi@lmi backtrace]$ ~/workspace/nano/buildroot/output/host/usr/bin/aarch64-none-linux-gnu-addr2line -e app 0x400704
/home/lmi/workspace/master_csel1/05_code/01_environment/backtrace/main.c:37
[lmi@lmi backtrace]$ ~/workspace/nano/buildroot/output/host/usr/bin/ aarch64-none-linux-gnu-addr2line -e app 0x400740
/home/lmi/workspace/master_csel1/05_code/01_environment/backtrace/main.c:46
[lmi@lmi backtrace]$ ~/workspace/nano/buildroot/output/host/usr/bin/ aarch64-none-linux-gnu-addr2line -e app 0x400774
/home/lmi/workspace/master_csel1/05_code/01_environment/backtrace/main.c:52
```

System calls

- `strace` est un outil pour tracer les appels système du noyau Linux.
- Pour obtenir la liste des appels, il suffit de lancer l'application sous le contrôle de `strace` :

```
strace <program-name>
```

- Un appel typique peut donner : *(…/system_calls)*

```
# strace ./app_a
execve("./app_a", ["./app_a"], [/* 17 vars */]) = 0
...
open("/sys/class/thermal/thermal_zone0/temp", O_RDONLY) = 3
read(3, "46000\n", 50)                          = 6
close(3)                                        = 0
...
+++ exited with 0 +++
#
```

- Étant donné que les informations affichées peuvent être très nombreuses, il est parfois bon de les limiter en spécifiant un ensemble des appels système intéressants:

```
strace -e trace=<set of calls> <program>

# strace -e trace=openat,close,mmap,munmap ./app_a
```

- Pour plus de détails, voir « `man strace` » ou http://en.wikipedia.org/wiki/strace

Memory leaks

- `valgrind` est un outil pour debugger, effectuer du profilage de code et mettre en évidence des fuites mémoires.
- Pour analyser le code, il suffit de lancer le programme sous le contrôle de `valgrind` :

```
valgrind --leak-check=full <program-name>
```

- Pour plus de détails, voir «`man valgrind`» ou la homepage `valgrind` ou http://en.wikipedia.org/wiki/Valgrind

MASTER OF SCIENCE IN ENGINEERING

MSE

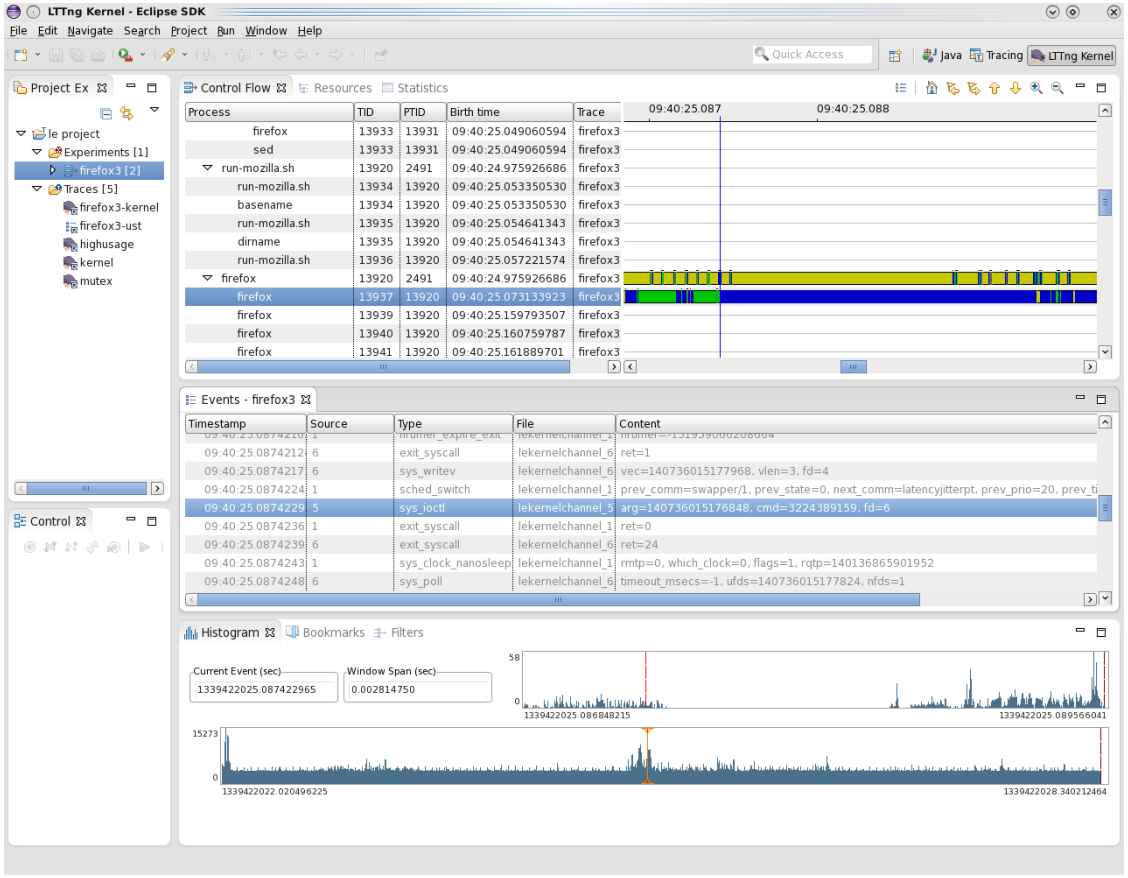
Memory leaks (II)

- **Un appel typique peut donner:** *(…/memory_leaks)*

```
# valgrind --leak-check=full ./app
==291== Memcheck, a memory error detector
==291== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==291== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==291== Command: ./app
==291==
==291==
==291== HEAP SUMMARY:
==291==     in use at exit: 63,760 bytes in 3,985 blocks
==291==   total heap usage: 4,000 allocs, 15 frees, 64,000 bytes allocated
==291==
==291== 63,760 (16 direct, 63,744 indirect) bytes in 1 blocks are definitely lost in loss record 3 of 3
==291==    at 0x4848400: malloc (in /usr/lib/valgrind/vgpreload_memcheck-arm64-linux.so)
==291==    by 0x4006B7: alloc2 (mem_leaks.c:65)
==291==    by 0x40077F: alloc (mem_leaks.c:82)
==291==    by 0x40081B: main (mem_leaks.c:98)
==291==
==291== LEAK SUMMARY:
==291==    definitely lost: 16 bytes in 1 blocks
==291==    indirectly lost: 63,744 bytes in 3,984 blocks
==291==      possibly lost: 0 bytes in 0 blocks
==291==    still reachable: 0 bytes in 0 blocks
==291==         suppressed: 0 bytes in 0 blocks
==291==
==291== For lists of detected and suppressed errors, rerun with: -s
==291== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
#
```

LTTng

- **LTTng (Linux Trace Toolkit – next generation) est un outil moderne pour la corrélation de traces prises dans le noyau et/ou l'espace utilisateur.**
- **LTTng utilise la même base de temps pour le traçage d'événements en espace utilisateur et en espace noyau.**
- **LTTng est intégré dans le noyau Linux depuis la version 2.6.38**
- **LTTng est intégré dans l'IDE Eclipse depuis la version Juno**
- **Plus de détails sous https://lttng.org**



PROC file system

- Le noyau Linux fournit énormément d'informations sur le système et son comportement par l'intermédiaire du «proc file system» `/proc/<...>`
- Par exemple `/proc/cpuinfo, /proc/meminfo, /proc/1/maps`
- Pour obtenir l'information, il suffit d'utiliser la commande `cat`

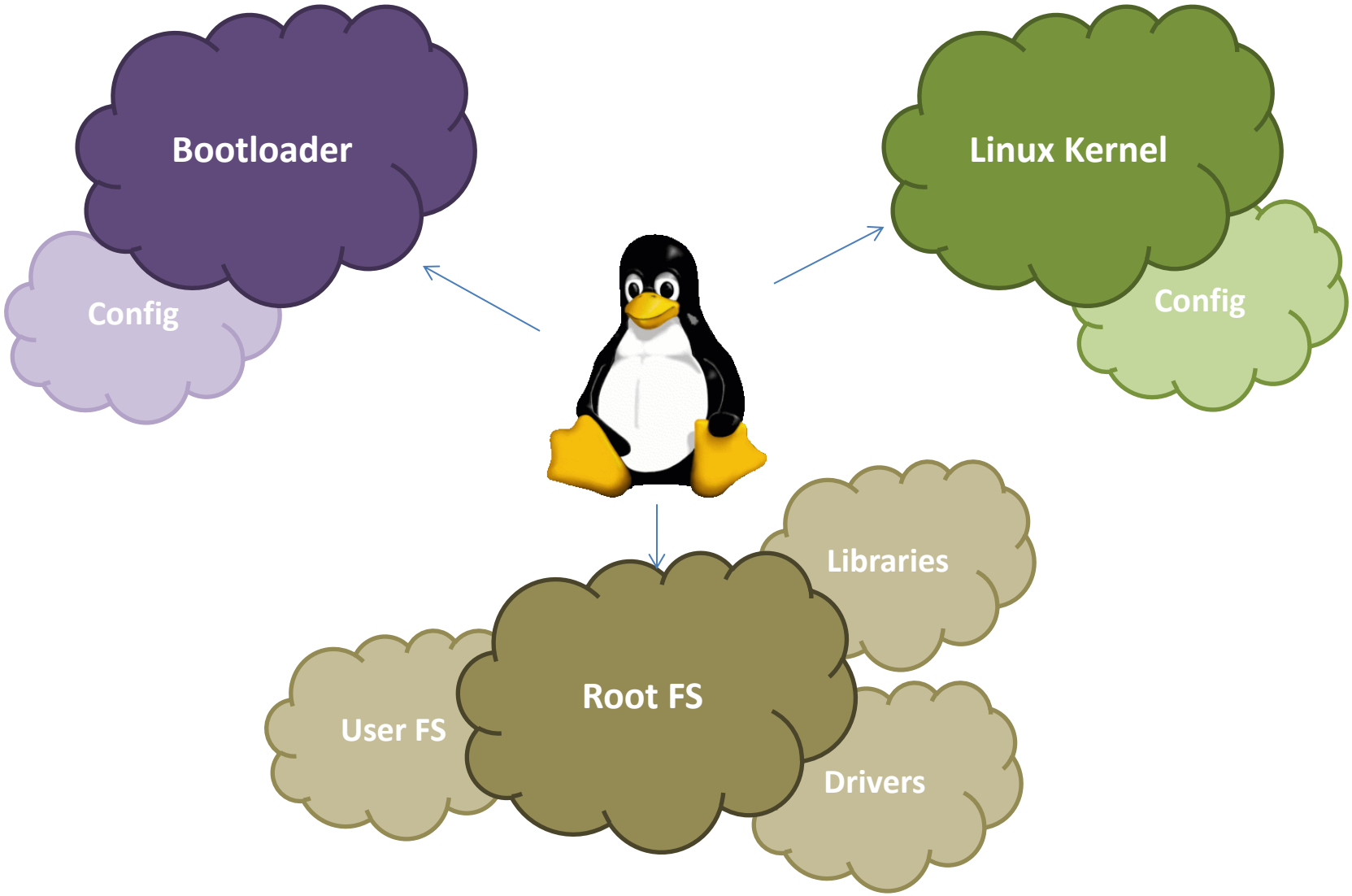
```
# cat /proc/1/maps
00400000-004b3000 r-xp 00000000 b3:02 16                 /bin/busybox
004c2000-004c3000 r--p 000b2000 b3:02 16                 /bin/busybox
004c3000-004c4000 rw-p 000b3000 b3:02 16                 /bin/busybox
004c4000-004c5000 rw-p 00000000 00:00 0
2e127000-2e148000 rw-p 00000000 00:00 0                  [heap]
ffff96afb000-ffff96c4c000 r-xp 00000000 b3:02 192        /lib/libc-2.30.so
ffff96c4c000-ffff96c5b000 ---p 00151000 b3:02 192        /lib/libc-2.30.so
ffff96c5b000-ffff96c5f000 r--p 00150000 b3:02 192        /lib/libc-2.30.so
ffff96c5f000-ffff96c61000 rw-p 00154000 b3:02 192        /lib/libc-2.30.so
ffff96c61000-ffff96c65000 rw-p 00000000 00:00 0
ffff96c65000-ffff96c77000 r-xp 00000000 b3:02 215        /lib/libresolv-2.30.so
ffff96c77000-ffff96c87000 ---p 00012000 b3:02 215        /lib/libresolv-2.30.so
ffff96c87000-ffff96c88000 r--p 00012000 b3:02 215        /lib/libresolv-2.30.so
ffff96c88000-ffff96c89000 rw-p 00013000 b3:02 215        /lib/libresolv-2.30.so
ffff96c89000-ffff96c8b000 rw-p 00000000 00:00 0
ffff96c8b000-ffff96caa000 r-xp 00000000 b3:02 186        /lib/ld-2.30.so
ffff96cb4000-ffff96cb8000 rw-p 00000000 00:00 0
ffff96cb8000-ffff96cb9000 r--p 00000000 00:00 0          [vvar]
ffff96cb9000-ffff96cba000 r-xp 00000000 00:00 0          [vdso]
ffff96cba000-ffff96cbb000 r--p 0001f000 b3:02 186        /lib/ld-2.30.so
ffff96cbb000-ffff96cbd000 rw-p 00020000 b3:02 186        /lib/ld-2.30.so
ffffdd2e9000-ffffdd30a000 rw-p 00000000 00:00 0          [stack]
```

Distribution Linux embarqué

MASTER OF SCIENCE IN ENGINEERING

MSE

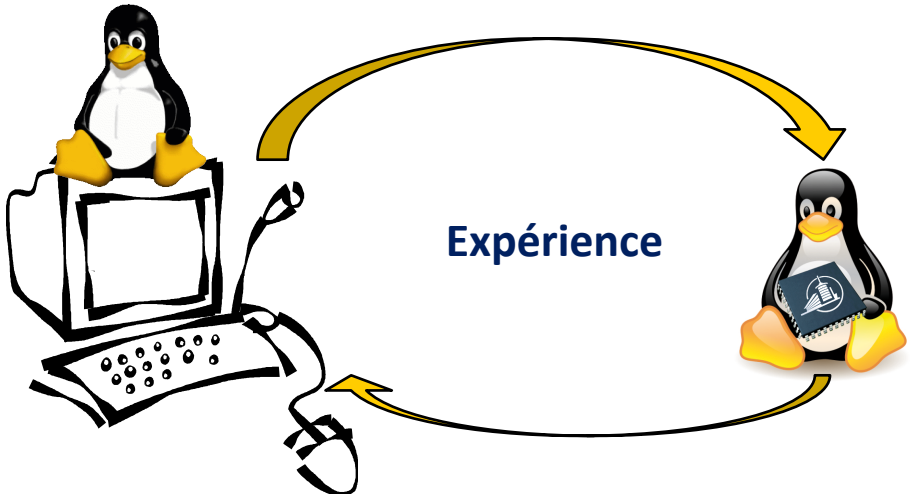
Composantes du système Linux



MASTER OF SCIENCE IN ENGINEERING
MSE

Composants et OS de la machine de développement

- **La machine de développement fournit toute la chaîne d'outils de développement permettant de générer le code exécutable pour la cible**
  - Compilateur, debugger,
  - Outils de gestions des codes sources,
  - …
- **Bien qu'il soit possible de développer des applications pour Linux embarqué sur différents types de machine (Windows, Mac OS X, …), il est préférable d'utiliser une machine avec Linux comme système opératif**
  - Meilleur support de la communauté
  - L'expérience acquise sur la machine hôte sert au développement sur la cible



Tâches de développement sous Linux embarqué

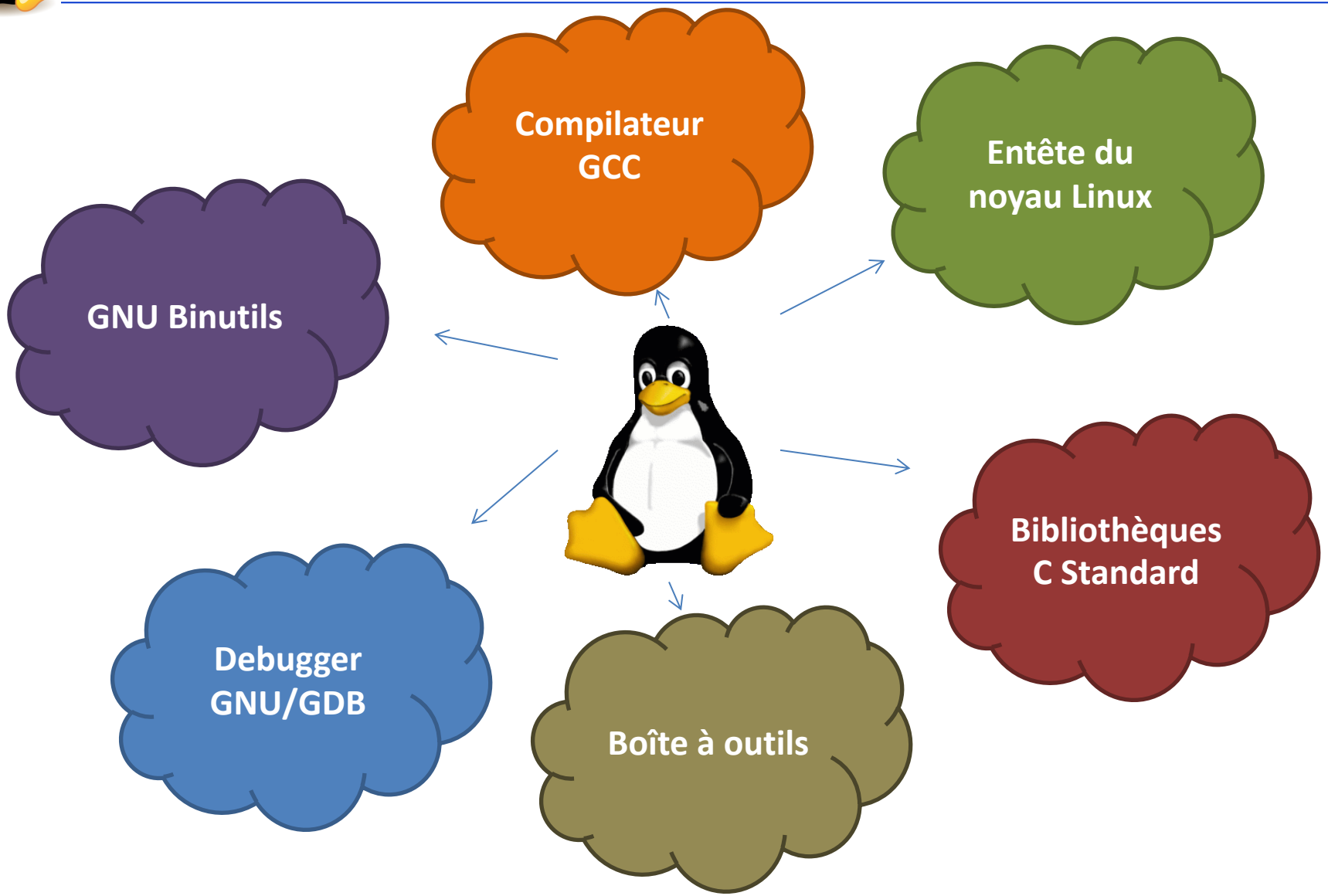
- **La mise en œuvre d'application sous Linux embarqué peut être répartie en 4 tâches distinctes**
  - Développement de «Board Support Package»
    - *Tâche consistant à mettre en œuvre une distribution Linux avec son bootloader pour un système embarqué cible (adaptation des routines d'initialisation du matériel, des pilotes de périphériques, …)*
  - Développement de logiciels noyau
    - *Tâche consistant à développer des logiciels fonctionnant dans le noyau Linux (développement de pile de protocoles, de pilotes de périphériques, …)*
  - Développement d'applications système
    - *Tâche consistant à développer des applications et/ou de bibliothèques logicielles dans l'espace utilisateur spécifique au système embarqué cible*
  - Intégration du système
    - *Tâche consistant à intégrer et valider les différentes composantes (bootloader, noyau, applications, …) pour construire le système final*

Solutions Linux embarqué

- **Deux chemins sont possibles pour aller vers du Linux embarqué**
  - Utiliser des **solutions propriétaires** développées et maintenues par des fournisseurs, tels que MontaVista, Wind River ou TimeSys
    - *Faciles d'accès, mais souvent coûteuses (royalities, runtime fees, …)*
    - *Ces solutions viennent avec leurs propres environnements de développement pas toujours suffisamment flexible pour les besoins.*
    - *Leurs outils de développement utilisent un mélange de composants open-source et les outils propriétaires.*
  - Utiliser des **solutions open-source** développées par une large communauté
    - *Exigences plus élevées, mais sans redevance*
    - *Solutions très flexible et adaptable aux besoins*
    - *Tous les logiciels sont ouverts et soutenus par la communauté*
- **Dans le cadre de ce cours, nous allons utiliser une solution open-source**
  - Avec les concepts présentés dans ce cours, un passage à des solutions de propriétaire sera facile

MASTER OF SCIENCE IN ENGINEERING
MSE

Composantes de la chaîne d'outils



MASTER OF SCIENCE IN ENGINEERING
MSE

Compilateur GCC

- **GCC, the GNU Compiler Collection**
  - Célèbre compilateur de la non moins célèbre «Free Software Fondation»
  - Disponible sous licence GPL et les bibliothèques sous LGPL
  - Référence: http://gcc.gnu.org/

- **GCC est capable de compiler**
  - C, C++, Objective-C, Fortran, Java, Ada et Go
  - Bibliothèques pour ces langages (libstdc++, libgcj, …)

- **GCC permet de générer des exécutables pour une large palette de CPU**
  - ARM, AVR, Blackfin, CRIS, FRV, M32, MIPS, MN20300, PowerPC, SH
  - v850, i386, x86_64, IA64, Xtensa, …

GNU Binutils

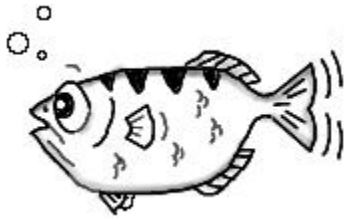
- **Binutils est une suite d'outils permettant de manipuler les fichiers objets pour une architecture de processeur donnée,**
  - Disponible sous licence GPL
  - Référence: http://www.gnu.org/software/binutils/

- **Les deux outils les plus connus**
  - **ld** – l'éditeur de liens GNU
  - **as** – l'assembleur GNU
- **Il contient également**
  - ar, ranlib – outils pour générer et manipuler des archives (*.a)
  - objdump, objcopy, nm, readelf, size, strings, strip – outils pour analyser et manipuler les fichiers objets
  - addr2line, c++filt, gprof – outils de debugging en complément à GDB

MASTER OF SCIENCE IN ENGINEERING
MSE

Debugger GNU

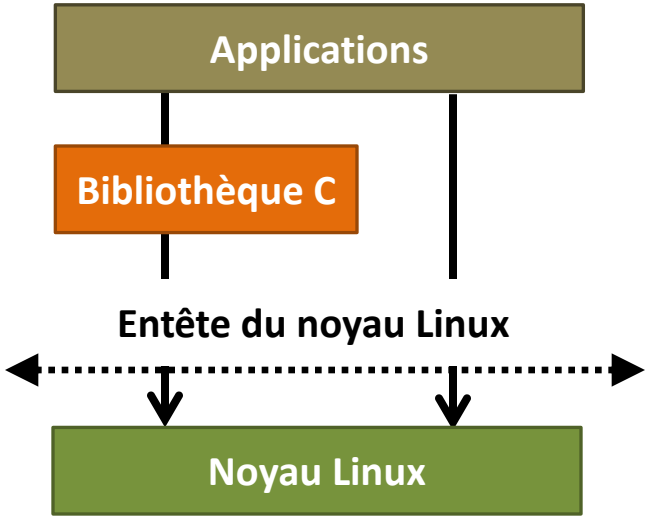
- **GDB: The GNU Projet Debugger**
  - Le debugger GDB permet de voir et d'analyser ce qui se passe à l'intérieur d'un programme lors de son exécution ou ce que le programme faisait au moment où il s'est écrasé.
  - Disponible sous licence GPL
  - Référence : http://www.gnu.org/software/gdb/



- **GDB est capable debugger**
  - Des programmes écrits dans les langages supportés par le compilateur GCC
  - Des programmes s'exécutant sur la même machine que GDB (natif)
  - Des programmes s'exécutant sur des machines distantes (remote)

Entêtes du noyau Linux

- **Les applications ainsi que les bibliothèques C standard l'interface doivent pouvoir interagir avec les services du noyau Linux.**
- **Les entêtes du noyau Linux fournissent**
  - Les appels système et leur numéro
  - La définition des constantes
  - Les structures de données
  - …



- **Pour compiler les applications et les bibliothèques, il est indispensable de pouvoir disposer de ces entêtes.**
- **Elles sont généralement disponibles sous**
  - `<linux/...>` et `<asm/...>` et quelques autres répertoires, visible dans les sources du noyau sous le répertoire `<include/>`

Bibliothèques C standard

- **Les bibliothèques C standard sont des composants indispensables au fonctionnement d'un système Linux**
  - Interface entre le noyau et les applications dans l'espace utilisateur
  - Suite de fonctions/services standardisés et bien connus facilitant le développement d'applications
- **Il existe tout un catalogue de fournisseur de ces bibliothèques**
  - Les plus connues: **glibc** de GNU et **uClibc** de CodePoet Consulting
  - Autres implémentations:
    - *eglibc* → *http://www.eglibc.org/*
    - *dietlibc* → *http://www.fefe.de/dietlibc/*
    - *newlib* → *http://sourceware.org/newlib/*
    - *…*
- **Le choix d'un bibliothèque dépendra de**
  - La fonctionnalité désirée et nécessaire
  - La place mémoire disponible sur la cible
- **Le choix doit se faire lors de la compilation de la chaîne d'outils**

GNU glibc

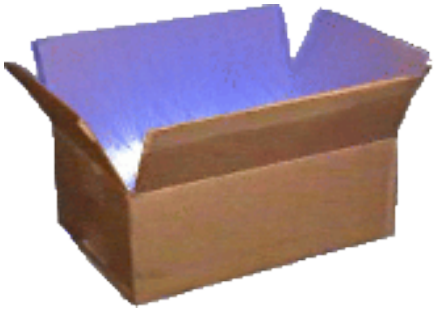
- **Caractéristiques principales**
  - Licence LGPL
  - Référence http://www.gnu.org/software/libc/
  - Bibliothèque C standard dérivée du «GNU Projet»
  - Maintenance très active
  - Disponible sur toutes les machines hôtes GNU / Linux
  - Design fait pour les performances et la portabilité
  - Conformité avec les standards garantie
  - Taille souvent un peu grande pour les systèmes embarqués
    - *Environ 2.5 MiB en version 2.9 pour les processeurs ARM*

µClibc

- **Caractéristiques principales**
  - Licence LGPL
  - Référence http://www.uclibc.org/
  - Bibliothèque C allégée pour les systèmes embarqués
    - *Hautement configurable (fonctions peuvent être activées ou désactivées)*
    - *Disponible sous Linux et uClinux*
    - *Focus sur la taille avant les performances*
      - » *env. 4x plus petit que glibc pour ARM*
    - *Certains services ne sont pas disponibles ou que partiellement*
      http://www.kernel.org/pub/linux/libs/uclibc/Glibc_vs_uClibc_Differences.txt
  - Largement utilisée sur les systèmes embarqués
  - Maintenance très active assurée par une grande communauté
  - Supportée par MontaVista, TimeSys et Wind River
  - Design fait pour les performances et la portabilité

Boîte à outils

- **Pour fonctionner, un système Linux a besoin d'un certain nombre de programmes et utilitaires**
  - Un programme d'initialisation
  - Une ligne de commande (une shell)
  - Divers outils basiques pour la manipulation de fichiers et la configuration du système



- **Sur les systèmes Linux standard**
  - Outils fournis par différents projets coreutils, bash, grep, sed, tar, wget, modutils par exemple
  - Nécessite l'intégration de divers composants
  - Composants pas toujours conçu pour les systèmes embarqués et disposant souvent de trop de fonctionnalité et pas toujours configurable à souhait
- **Pour les systèmes Linux embarqués, Busybox est une solution très attractive**
  - Très commune dans les systèmes embarqués
  - Un seul exécutable hautement configurable
  - Taille raisonnable (env. < 500 KiB avec uClib et < 1 MiB avec glibc)
  - Référence http://busybox.net/

MASTER OF SCIENCE IN ENGINEERING
MSE

Génération de la chaîne d'outils

- **La génération manuelle d'une chaîne d'outils est une tâche difficile et ardue, qui peut prendre plusieurs jours ou semaines!**
- **Celle-ci nécessite de**
  - Apprendre énormément de détails sur énormément de composants pas toujours simples à configurer
  - Prendre énormément de décisions (choix de bibliothèques et des versions, …)
  - Disposer des sources et entêtes du noyau
  - Disposer des sources des bibliothèques C standard
  - …
- **Alternatives**
  - Utilisation d'une chaine d'outils précompilée
    - *Solution la plus simple et la plus conviviale*
    - *Impossible d'adapter l'outil à ses besoins spécifiques*
  - Utilisation d'un outil permettant de générer une distribution Linux à partir de zéro
    - *Solution ne nécessitant aucun détail du processus de génération*
    - *Solution offrant des services de configuration (version, composants, …)*
    - *Solution permettant de patches les composants*

Chaînes d'outils précompilées

- **Linaro**
  - Référence https://wiki.linaro.org/
  - Linaro contribue à améliorer le compilateur GCC pour ARM

- **CodeSourcery**
  - Référence http://www.codesourcery.com/
  - Compagnie avec une grande expertise sur les chaînes d'outils libres
  - Ces chaînes d'outils sont disponibles pour ARM, MIPS, PowerPC, SuperH, …
  - Egalement possible d'acheter ces outils avec du support

- **Autres distributions**
  - Références http://elinux.org/Toolchains

Outils de génération pour des distributions Linux

- **Buildroot**
  - Référence http://buildroot.uclibc.org/
  - Système basé sur les Makefile
  - Développé et maintenu par la communauté

- **Yocto Projet**
  - Référence https://www.yoctoproject.org
  - Système très performant et complet, mais complexe
  - Développé et maintenu par la communauté

- **PTXdist**
  - Référence http://www.ptxdist.org/
  - Système basé sur les Makefile
  - Développé et maintenu par Pengutronix

- **ELDK (Embedded Linux Development Kit)**
  - Référence http://www.denx.de/wiki/ELDK-5/WebHome
  - Système basé sur le projet Yocto (http://www.yoctoproject.org/)
  - Développé et maintenu par la maison Denx

MASTER OF SCIENCE IN ENGINEERING

MSE

Buildroot

- **Buildroot est outil permettant de générer une distribution à partir de zéro, comprenant**
  - Chaîne d'outils croisée
  - Image du noyau Linux sous différent format
  - RootFS complet sous différents types de systèmes de fichiers
  - Bootloader (U-Boot, Barebox, …)
- **Buildroot ne permet de générer qu'une chaîne d'outils basée sur uClib, mais on pourra utiliser une chaîne d'outils externe si la glibc est préférée**
- **Plus 500 applications et bibliothèques ont été intégrées dans la distribution**
  - X.org, Gstreamer, Qt, Gtk, Webkit, ….
- **Bien adapté pour les systèmes embarqués de petite à moyenne taille**
  - Pas d'aide pour la génération de packages (.deb ou .ipk)
  - Nécessite une compilation complète pour les grandes modifications de configuration
- **Communauté très active avec une release tous les 3 mois**

Distributions

- **Debian GNU/Linux**
  - Référence http://www.debian.org/
  - Disponible sur les architectures ARM, MIPS et PowerPC
  - Propose une solution clef en main avec tous les logiciels nécessaires
  - Offre une très grande flexibilité grâce au gestionnaire de «package», mais nécessite une grande capacité de stockage (>300MiB de Flash) et de mémoire (> 64MiB RAM)
  - Le projet Emdebian (http://www.emdebian.org) travaillant sur la migration des distributions Debian pour les systèmes embarqués
- **Ubuntu GNU/Linux**
  - Référence http://www.ubuntu.com/
  - Basé sur la distribution avec les mêmes avantages
  - Disponible sur ARM, mais seulement dès les processeurs Cortex A8 et plus
  - Release tous les 6 mois
- **Fedora**
  - Référence http://fedoraproject.org/wiki/Architectures/ARM
  - Egalement disponible sur ARM, mais pas très activement

Mise en mode production

Linux initialisation

- **Une propriété des systèmes embarqués est autonomie, c.-à-d. leur capacité à fonctionner sans intervention extérieure.**
- **Pour atteindre cette autonomie, il est essentiel de pouvoir lancer les logiciels d'application au démarrage de la cible sans aide extérieure. Linux implémente un processus d'initialisation. Ce processus est choisi dans l'ordre suivant :**

1. `/sbin/init` → l'emplacement préféré et le plus probable pour le processus init
2. `/etc/init` → un autre emplacement probable pour le processus init
3. `/bin/init` → également un emplacement possible pour le processus init
4. `/bin/sh` → l'emplacement de la shell Bourne

- **Sources**
  - Fichier: init/main.c
  - Routine: kernel_init

MASTER OF SCIENCE IN ENGINEERING

MSE

BusyBox processus init

- **Le logiciel BusyBox est un environnement de commandes très populaire dans les systèmes embarqués, à l'instar des GNU Core Utilities.**
- **Le processus d'initialisation de BusyBox est appelé par le noyau à l'aide d'un lien symbolique**

```
# ls -l /sbin/init
lrwxrwxrwx  1 default  default 14 Feb  5  2012 /sbin/init -> ../bin/busybox
```

- **Dans ce processus, BusyBox analyse le fichier `/etc/inittab` et exécute les instructions qu'il contient. Ce fichier était à l'origine le fichier de configuration des daemons du système Unix V.**
- **BusyBox offre deux possibilités:**
  - Mettre une entrée dans le fichier `/etc/inittab`
  - Placer un script de lancement dans `/etc/init.d`
    - *Le script `/etc/init.d/rcS` exécutera dans l'ordre alphabétique tous les fichiers `S??*`*

MASTER OF SCIENCE IN ENGINEERING

MSE

Fichier /etc/inittab

- **Le fichier /etc/inittab contient une liste d'instructions qui sera interprétée par BusyBox au démarrage du noyau Linux.**
- **Chaque ligne du fichier a la structure suivante:**

  `id:runlevel:action:process`

  - `id`: nom du périphique tty, peut être laissé vide
  - `runlevel`: complètement ignoré par BusyBox (laissé toujours vide)
  - `action`: action à appliquer sur le programme, 8 possibilité
    1. `sysinit`: script d'initialisation
    2. `respawn`: redémarre le processus/programme chaque fois qu'il se termine
    3. `askfirst`: similaire à 2. mais demande à la console s'il doit être réactivé
    4. `wait`: attend que le processus soit terminé pour continuer
    5. `once`: lance le programme une seule fois sans attendre
    6. `ctrlatldel`: lance le programme lorsque les touches Ctrl-Alt-Delete sont pressées
    7. `shutdown`: lance le programme lorsque le système est en « shutdown »
    8. `restart`: processus à lancer lorsque le processus d'init est redémarré
  - `process`: spécifie le programme (avec son chemin) que l'action doit contrôler

Fichier /etc/inittab (II)

Exemple de fichier tiré du NanoPi NEO Plus2

```
# /etc/inittab
#
# Copyright (C) 2001 Erik Andersen <andersen@codepoet.org>
#
# Note: BusyBox init doesn't support runlevels.  The runlevels field is
# completely ignored by BusyBox init. If you want runlevels, use
# sysvinit.
#
# Format for each entry: <id>:<runlevels>:<action>:<process>
#
# id        == tty to run on, or empty for /dev/console
# runlevels == ignored
# action    == one of sysinit, respawn, askfirst, wait, and once
# process   == program to run

# Startup the system
::sysinit:/bin/mount -t proc proc /proc
::sysinit:/bin/mount -o remount,rw /
::sysinit:/bin/mkdir -p /dev/pts /dev/shm
::sysinit:/bin/mount -a
::sysinit:/sbin/swapon -a
null::sysinit:/bin/ln -sf /proc/self/fd /dev/fd
null::sysinit:/bin/ln -sf /proc/self/fd/0 /dev/stdin
null::sysinit:/bin/ln -sf /proc/self/fd/1 /dev/stdout
null::sysinit:/bin/ln -sf /proc/self/fd/2 /dev/stderr
::sysinit:/bin/hostname -F /etc/hostname
# now run any rc scripts
::sysinit:/etc/init.d/rcS

# Put a getty on the serial port
console::respawn:/sbin/getty -L  console 0 vt100 # GENERIC_SERIAL

# Stuff to do for the 3-finger salute
#::ctrlaltdel:/sbin/reboot

# Stuff to do before rebooting
::shutdown:/etc/init.d/rcK
::shutdown:/sbin/swapoff -a
::shutdown:/bin/umount -a -r
```

Fichier S??*

- **Les scripts de lancement des applications sont appelés dans l'ordre alphabétique. Ils doivent impérativement débuter avec un S majuscule lequel sera suivi de deux chiffres.**
- **Exemple de fichier:** *(…/daemon)*

```
#!/bin/sh
#
# Application daemon
#
case "$1" in
  start)
  /usr/local/app_a
  ;;
  stop)
  killall app_a
  ;;
  restart|reload)
  killall app_a
  /usr/local/app_a
  ;;
  *)
  echo $"Usage: $0 {start|stop|restart}"
  exit 1
esac

echo "Application daemon launched"

exit $?
```

Chaîne d'outils GNU

Chaine d'outils GNU

- make — utilitaire pour l'automation de la génération de logiciel
- gcc — compilateur C/C++
- ar — utilitaire pour la création de bibliothèques
- as — assembleur
- ld — éditeur de liens

- gdb — debugger / debugger
- ddd — data display debugger
- addr2line — utilitaire pour convertir une adresse vers un fichier et une ligne

- nm — utilitaire pour lister les symboles d'un fichier de code objet
- objdump — utilitaire pour lister des informations d'un fichier de code objet
- size — utilitaire pour lister la taille du code objet
- strings — utilitaire pour lister les chaines de caractères imprimables
- strip — utilitaire pour éliminer les symboles d'un fichier de code objet

- gcov — code/test coverage
- gprof — profiling tool

MASTER OF SCIENCE IN ENGINEERING

MSE

Compilation

Avec la chaîne d'outils GNU sous Linux

 # compilation de l'application pour le debugging

- `gcc -g -c -Wall -Wextra -O2 -std=gnu11 -o fibonacci.o fibonnaci.c`
  - `-g:` enclenche les options pour le debugging
  - `-c:` indique au compilateur de ne pas linker l'application
  - `-Wall:` enclenche tous les warning
  - `-Wextra:` enclenche des warning supplémentaires
  - `-O2:` force l'optimisation du code généré
  - `-std=gnu11:` autorise l'utilisation des extensions de 2011
  - `-o:` spécifie le nom du fichier de sortie

 # Si tous les fichiers d'entête (header files) ne sont pas dans le même répertoire que le fichier à compiler:

- `-I` *`dir`*: spécifie le répertoire où le compilateur trouvera les *header files, p. ex:*
  `-I. -I.. -I /workspace/include`

MASTER OF SCIENCE IN ENGINEERING | MSE

Linkage

 # après la compilation de l'application vous devez linker tous les fichiers afin d'obtenir le fichier exécutable:

- `gcc fibonacci.o -o fibonacci`

 # il est également possible de compiler et de linker l'application en une seule commande:

- `gcc -g -Wall -Wextra -O2 -std=gnu11 -o fibonacci fibonnaci.c`
  - `-o:` spécifie le nom de l'exécutable
  - `-c:` attention ce flag ne doit être utilisé!!!

MASTER OF SCIENCE IN ENGINEERING

MSE

Lancement & désassemblage

```
# démarrage l'application
  ▸ ./fibonacci 15

# taille du code objet
  ▸ size -t fibonacci.o

# désassemblage du code objet
  ▸ objdump -d -S -C fibonacci.o

# désassemblage de l'application
  ▸ objdump -d -S -C fibonacci

# éliminer les symboles de debugging
  ▸ strip -g -o fibonacci_s fibonacci
```

MASTER OF SCIENCE IN ENGINEERING

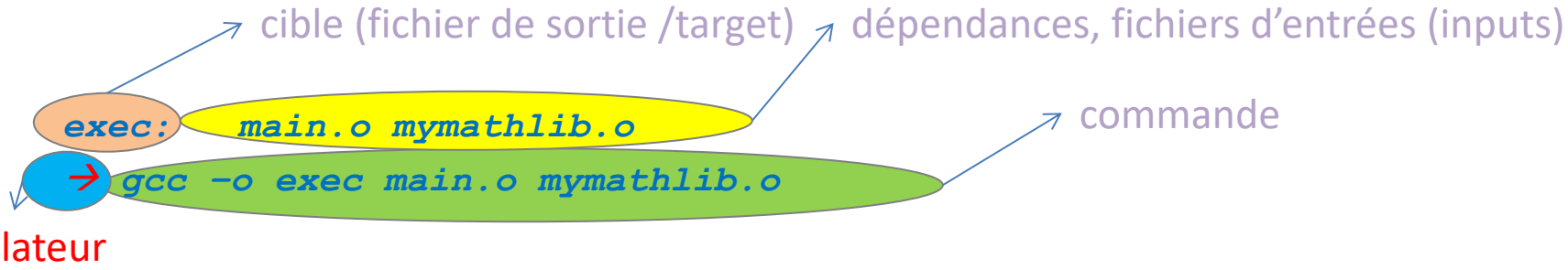
MSE

Utilitaire – make

- « `make` » est un utilitaire permettant de décrire les dépendances d'une application et de compiler les fichiers objets nécessaires pour sa génération.

`make` ou `make -f Makefile` ou `make –f mymakefile`

- La description de ces dépendances se font par l'intermédiaire d'un fichier texte. Le nom de défaut est «`Makefile`», mais il possible de le nommer différemment.
- Par exemple:



tabulateur

```
main.o: main.c mymathlib.h
  → gcc –c –Wall –Wextra –O2 –std=gnu11 –g –o main.o main.c

mymathlib.o: mymathlib.c mymathlib.h
  → gcc –c –Wall –Wextra –O2 –std=gnu11 –g –o mymathlib.o mymathlib.c
```

make - variables

- **L'utilisation de variables simplifie grandement l'écriture des «`Makefile`», p.ex.:**

```
EXEC=exec
CC=gcc
CFLAGS=-Wall -Wextra -O2 -std=gnu11 -g -c

$(EXEC): main.o mymathlib.o
   $(CC) $(LDFLAGS) -o $(EXEC) main.o mymathlib.o

main.o: main.c mymathlib.h
   $(CC) $(CFLAGS) -o main.o main.c

mymathlib.o: mymathlib.c mymathlib.h
   $(CC) $(CFLAGS) -o mymathlib.o mymathlib.c
```

Make – plusieurs cibles (targets)

- **Il est possible de spécifier dans un même «`Makefile`» plusieurs cibles différentes, p. ex. :**

```
EXEC=exec
CC=gcc
CFLAGS=-Wall -Wextra -O2 -std=gnu11 -g -c

all: $(EXEC)

$(EXEC): main.o mymathlib.o
  $(CC) $(LDFLAGS) -o $(EXEC) main.o \
    mymathlib.o

main.o: main.c mymathlib.h
  $(CC) $(CFLAGS) -o main.o main.c

mymathlib.o: mymathlib.c mymathlib.h
  $(CC) $(CFLAGS) -o mymathlib.o \
    mymathlib.c

clean:
  rm -f $(EXEC)
  rm -f *.o

.PHONY: all clean
```

Pour générer l'application (fichier de sortie), il suffit de taper la commande:

`make ou make all`

si aucune cible n'est spécifiée, c'est la première cible qui se trouve dans le «`Makefile`» qui sera générée.

Pour effacer le fichier de sortie et tous les fichiers de dépendances, il suffit de taper la commande:

`make clean`

Indique ces cibles sont factices et force ainsi leur reconstruction

MASTER OF SCIENCE IN ENGINEERING | MSE

make – règles suffixes

- **Les règles suffixes permettent d'éviter de spécifier pour chaque fichier (cible ou dépendance) les commandes à effectuer. Ce travail fastidieux peut être simplifié de la manière suivante:**

```
SRCS=main.c mymathlib.c
EXEC=exec
CC=gcc
CFLAGS=-Wall -Wextra -O2 -std=gnu11 -g -c

OBJS=$(SRCS:.c=.o)

all: $(EXEC)

.c.o:
   $(CC) $(CFLAGS) -o $*.o $<

$(EXEC): $(OBJS)
   $(CC) $(LDFLAGS) -o $@ $^

clean:
   rm -f $(EXEC)
   rm -f *.o

.PHONY: all clean
```

Expression transformant le nom des fichiers .c contenus dans la variable SRCS en .o

.c.o:
définit une règle suffixe pour les fichiers d'entrée .c et les fichiers de sorties .o

Explication des variables:

$@	nom de la cible (fichier de sortie)
$<	nom du fichier d'entrée
$*	nom du fichier sans le suffixe
$^	nom de toutes les dépendances
$?	noms de toutes les dépendances plus récentes que la cible

MASTER OF SCIENCE IN ENGINEERING | MSE

make - dépendances

- **Lors de la génération d'applications développées en C il est essentiel de garantir que celles-ci soient correctement générées et que toutes les modifications soient bien prises en compte. Pour ce faire, le compilateur GNU permet de générer un fichier des dépendances, qui pourra ensuite être utilisé dans le «`Makefile`».**

```
SRCS=main.c mymathlib.c
EXEC=exec
CC=gcc
CFLAGS=-Wall -Wextra -O2 -std=gnu11 -g -c
OBJS=$(SRCS:.c=.o)

all: $(EXEC)

-include $(OBJS:.o=.d)

.c.o:
    $(CC) $(CFLAGS) -MD -o $*.o $<
    $(CC) -MM $< -MF $*.d

$(EXEC): $(OBJS)
    $(CC) $(LDFLAGS) -o $@ $^

clean:
    rm -f $(EXEC)
    rm -f *.o *.d

.PHONY: all clean
```

ou

Cette instruction permet d'inclure tous les fichiers de dépendances pour les fichiers de code objet que nous désirons générer. Le signe « - » devant l'instruction « `include` » indique à l'utilitaire `make` de continuer la génération même si le fichier de dépendance n'existe pas.

La commande «-MD» du compilateur permet de générer un fichier de dépendance.

Pour certaines versions du compilateur il faut utiliser :

- **-MM: créer la dépendance**
- **-MF: spécifier le nom du fichier de sortie**

Ne pas oublier d'effacer les fichiers de dépendances…

MASTER OF SCIENCE IN ENGINEERING

MSE

make – conditions

- «`make`» offre des directives permettant d'exécuter conditionnellement une partie du «`Makefile`», ceci en fonction d'une variable et de sa valeur, p.ex.:

```
ifeq ($(VARIABLE), value)
   ## if true do that
else
   ## if false do this
endif
```

- La variable peut être contenue dans le «`Makefile`», mais il est également possible de la spécifier lors de l'appel du «`Makefile`», p.ex:

```
make VARIABLE=value <cible>
```

- Cette technique permet de générer un logiciel pour différentes plateformes (machine hôte, cible, …) ou en différentes versions (release, debug, …)

make – sous-Makefile

- «`make`» permet de créer plusieurs «`Makefile`» où chacun correspond à une partie distincte d'un grand projet. Pour simplifier sa génération, on préfère généralement appeler un «`Makefile`» unique gérant l'ensemble de la génération de l'application.

- La variable `$(MAKE)` fournit l'outil nécessaire pour l'appel de sous-«`Makefile`».

```
$(MAKE) -C <directory> <target>
```

- Le mot clef «export» permet de passer les variables du «`Makefile`» maître aux sous- «`Makefile`»

MASTER OF SCIENCE IN ENGINEERING

MSE

make – commandes silencieuses

- Pour éviter d'afficher une longue liste de commandes et d'arguments lors de la génération d'une application, «`make`» propose une instruction permettant de supprimer l'écho des lignes de commandes. Pour ceci, il suffit de rajouter le caractère «@» devant la ligne de commande, p. ex.:

```
.c.o:
   @echo "(CC) $<"
   @$(CC) $(CFLAGS) –MD -o $*.o $<

$(EXEC): $(OBJS)
   @echo "(LD) $@"
   @$(CC) $(LDFLAGS) -o $@ $^
```

MASTER OF SCIENCE IN ENGINEERING

MSE