## ***Software components:***

-***Cross-compilation toolchain:*** Compiler that runs on the development machine, but generates code for the target
-***Bootloader:*** Started by the hardware, responsible for basic initialization, loading and executing the kernel
-***Linux Kernel:*** Contains the process and memory management, network stack, device drivers and provides services to user space applications.
-***C library:*** The interface between the kernel and the user space applications
-***Libraries and applications:*** Third-party or in-house
* Several distinct tasks are needed when deploying embedded Linux in a product:
-Board Support Package development: A BSP contains a bootloader and kernel with the suitable device drivers for the targeted hardware.
-***System integration:*** Integrate all the components, bootloader, kernel, third-party libraries and applications and in-house applications into a working system.
-***Development of applications:*** Normal Linux applications, but using specifically chosen libraries.

## ***Host vs. target:***

-When doing embedded development, there is always a split between:

- ***The host:*** the development workstation, which is typically a powerful PC.
- ***The target:*** which is the embedded system under development.

-They are connected by various means: almost always a serial line for debugging purposes, frequently an Ethernet connection, sometimes a JTAG interface for low-level debugging.

## ***Serial line communication program:***

-An essential tool for embedded development is a serial line communication program, like HyperTerminal in Windows.
-There are multiple options available in Linux: Minicom, Picocom, Gtkterm, Putty, screen, etc.
-SERIAL_DEVICE is typically:

- ***ttyUSBx:*** for USB to serial converters
- ***ttySx:*** for real serial ports

-Most frequent command: *picocom -b 115200 /dev/ttyUSB0.*

## ***Cross-compiling toolchains***

*-The usual development tools available on a GNU/Linux workstation is a native toolchain. This toolchain runs on your workstation and generates code for your workstation, usually x86.*
*-For embedded system development, it is usually impossible or not interesting to use a native toolchain:*

- *The target is too restricted in terms of storage and/or memory.*
- *The target is very slow compared to your workstation.*
- *You may not want to install all development tools on your target.*

*-Therefore, cross-compiling toolchains are generally used. They run on your workstation but generate code for your target.*
*-Three machines must be distinguished when discussing toolchain creation*

- ***The build machine:*** *where the toolchain is built.*
- ***The host machine:*** *where the toolchain will be executed.*
- ***The target machine:*** *where the binaries created by the toolchain are executed.*

## ***Different toolchain build procedures:***

***-Native build:*** *used to build the normal gcc of a workstation.*
***-Cross native build:*** *used to build a toolchain that runs on your target and generates binaries for the target.*
***-Cross build:*** *used to build a toolchain that runs on your workstation and generate binaries for the target.*
⇒ *The most common case in Embedded development.*
***-Canadian cross build:*** *to build on architecture A a toolchain that runs on B and generates binary for architecture C.*

## ***Components:***

***Binutils:***

*- Binutils is a set of tools to generate and manipulate binaries for a given CPU architecture:*
  *-as: the assembler, that generates binary code from assembler source code*
  *-ld: the linker*
  *-ar, ranlib: to generate .a archives (static libraries)*
  *-objdump, readelf, size, nm, strings: to inspect binaries. Very useful analysis tools!*
  *-objcopy: to modify binaries*
  *-strip: to strip parts of binaries that are just needed for debugging (reducing their size).*

***Kernel headers:***

*- The C library and compiled programs need to interact with the kernel. Therefore, compiling the C library requires kernel headers, and many applications also require them.*
*- They are available in <linux/...> and <asm/...> and a few other directories corresponding to the ones visible in include/uapi/ and in arch/<arch>/include/uapi in the kernel sources.*
*- The kernel headers are extracted from the kernel sources using the headers_install kernel Makefile target.*
*- The kernel to user space ABI is backward compatible: **ABI = Application Binary Interface** - It's about binary compatibility.*

- *Binaries generated with a toolchain using kernel headers older than the running kernel will work without problem, but won't be able to use the new system calls, data structures, etc.*
- *Binaries generated with a toolchain using kernel headers newer than the running kernel might work only if they don't use the recent features, otherwise they will break.*

***C/C++ Compiler:***
*GCC: GNU Compiler Collection, the famous free software compiler.*
***C library:***
*-The C library is an essential component of a Linux system:*

- *Interface between the applications and the kernel.*
- *Provides the well-known standard C API to ease application development.*

*-Several C libraries are available: glibc, uClibc, musl, klibc, newlib..*
*-The choice of the C library must be made at cross-compiling toolchain generation time, as the GCC compiler is compiled against a specific C library.*
*⇒ Advice for choosing the C library*
*-Advice to start developing and debugging your applications with glibc, which is the most standard solution.*
*-Then, when everything works, if you have size constraints, try to compile your app and then the entire filesystem with uClibc or musl.*
*-If you run into trouble, it could be because of missing features in the C library.*
*-In case you wish to make static executables, musl will be an easier choice. Note that static executables built with a given C library can be used in a system with a different C library.*

## ***Toolchain Options:***

***ABI:***
*- When building a toolchain, the ABI used to generate binaries needs to be defined.*
*- ABI, for Application Binary Interface, defines the calling conventions (how function arguments are passed, how the return value is passed, how system calls are made) and the organization of structures (alignment, etc.)*
*- All binaries in a system are typically compiled with the same ABI, and the kernel must understand this ABI.*
*⇒ On ARM, two main ABIs: **OABI** and **EABI:** Nowadays everybody uses EABI*
*⇒ On MIPS, several ABIs: o32, o64, n32, n64*
***Floating point support:***
*- Some processors have a floating point unit, some others do not. For processors having a floating point unit, the toolchain should generate hard float code, in order to use the floating point instructions directly.*
*- For processors without a floating point unit, two solutions:*

- *Generate hard float code and rely on the kernel to emulate the floating point instructions. This is very slow.*
- *Generate soft float code, so that instead of generating floating point instructions, calls to a user space library are generated.*

***CPU optimization flags:***
*- A set of cross-compiling tools is specific to a CPU architecture (ARM, x86, MIPS, PowerPC).*
*- However, gcc offers further options:*

- *-march allows to select a specific target instruction set*
- *-mtune allows to optimize code for a specific CPU*
  *For example: -march=armv7 -mtune=cortex-a8*
- *-mcpu=cortex-a8 can be used instead to allow gcc to infer the target instruction set (-march=armv7) and cpu optimizations (-mtune=cortex-a8)*

## ***Obtaining a Toolchain***

***Get a pre-compiled toolchain:***
*- Make sure the toolchain you find meets your requirements: CPU, endianness, C library, component versions, ABI, soft float or hard float, etc.*

*⇒ Possible choices:*

- *Toolchains packaged by your distribution Ubuntu example: sudo apt install gcc-arm-linux-gnueabihf*
- *Bootlin's toolchains (for most architectures): https://toolchains.bootlin.com*
- *Toolchain provided by your hardware vendor.*

*- Another solution is to use utilities that automate the process of building the toolchain:*

***Crosstool-ng:***

- *Rewrite of the older Crosstool, with a menuconfig-like configuration system.*
- *Feature-full: supports uClibc, glibc and musl, hard and soft float, many architectures.*

*-Many root filesystem build systems also allow the construction of a cross-compiling toolchain:*

- ***Buildroot:*** *Makefile-based. Can build glibc, uClibc and musl based toolchains, for a wide range of architectures. Use make sdk to only generate a toolchain.*
- ***OpenEmbedded / Yocto Project:*** *A featureful, but more complicated build system.*

## *Crosstool-NG:*

*- Installation of Crosstool-NG can be done system-wide, or just locally in the source directory.*

** For local installation:*

```
./configure --enable-local
make
make install
```

** Some sample configurations for various architectures are available in samples, they can be listed using:*

```
./ct-ng list-samples
```

** To load a sample configuration*

```
./ct-ng <sample-name>
```

** To adjust the configuration*

```
./ct-ng menuconfig
```

** To build the toolchain*

```
./ct-ng build
```

** Clean the toolchain*

```
./ct-ng clean
```

## *Toolchain contents*

*- The cross compilation tool binaries, in bin/: This directory should be added to your PATH to ease usage of the toolchain.*

*- One or several sysroot, each containing:*

- ‣ *The C library and related libraries, compiled for the target.*
- ‣ *The C library headers and kernel headers.*

*- There is one sysroot for each variant: toolchains can be multilib if they have several copies of the C library for different configurations (for example: ARMv4T, ARMv5T, etc.)*

*- Old CodeSourcery ARM toolchains were multilib, the sysroots in: arm-none-linux-gnueabi/libc/, arm-none-linux-gnueabi/libc/armv4t/, arm-none-linux-gnueabi/libc/thumb2*

*- Crosstool-NG toolchains can be multilib too (still experimental), otherwise the sysroot is in: arm-unknown-linux-uclibcgnueabi/sysroot.*

## *Bootloaders:*

*-The bootloader is a piece of code responsible for:*

- *Basic hardware initialization*
- *Loading of an application binary, usually an operating system kernel, from flash storage, from the network, or from another type of non-volatile storage.*
- *Possibly decompression of the application binary*
- *Execution of the application*

*- Besides these basic functions, most bootloaders provide a shell with various commands implementing different operations like: Loading of data from storage or network, memory inspection, hardware diagnostics and testing.*

### ***Bootloaders on BIOS-based x86***

*1- The x86 processors are typically bundled on a board with a non-volatile memory containing a program, the BIOS.*

*2- On old BIOS-based x86 platforms: the BIOS is responsible for basic hardware initialization and loading of a very small piece of code from non-volatile storage.*

*3- This piece of code is typically a 1st stage bootloader, which will load the full bootloader itself.*

*4- It typically understands file system formats so that the kernel file can be loaded directly from a normal filesystem.*

*Example:*

*GRUB: Grand Unified Bootloader, the most powerful one: Can read many file system formats to load the kernel image and the configuration, provides a powerful shell with various commands, can load kernel images over the network, etc.*

***Booting on embedded CPUs: case 1:***

*1- When powered, the CPU starts executing code at a fixed address.*
*2- The hardware design must ensure that a NOR flash chip is wired so that it is accessible at the address at which the CPU starts executing instructions.*
*3- The first stage bootloader must be programmed at this address in the NOR.*
*4- NOR is mandatory, because it allows random access, which NAND doesn't allow.*

***Booting on embedded CPUs: case 2***

*- The CPU has an integrated boot code in ROM: BootROM on AT91 CPUs, "ROM code" on OMAP, etc.*
*- This boot code is able to load a first stage bootloader from a storage device into an internal SRAM (DRAM not initialized yet):*

- *Storage devices can typically be: MMC, NAND, SPI flash, UART (transmitting data over the serial line), etc.*

*- The first stage bootloader is*

- *Limited in size due to hardware constraints (SRAM size)*
- *Provided either by the CPU vendor or through community projects*

*- This first stage bootloader must initialize DRAM and other hardware devices and load a second stage bootloader into RAM.*

*Example case:*

**** Booting on Microchip ARM SAMA5D3:***

***1- RomBoot:** tries to find a valid bootstrap image from various storage sources, and loads it into SRAM (DRAM not initialized yet). Size limited to 64 KB. No user interaction possible in standard boot mode.*
***2- U-Boot SPL:** runs from SRAM. Initializes the DRAM, the NAND or SPI controller, and loads the secondary bootloader into RAM and starts it. No user interaction possible.*
***3- U-Boot:** runs from RAM. Initializes some other hardware devices (network, USB, etc.). Loads the kernel image from storage or network to RAM and starts it. Shell with commands provided.*
***4- Linux Kernel:** runs from RAM. Takes over the system completely (the bootloader no longer exists).*

**** Booting on Marvell SoCs:***

***1- ROM Code:** tries to find a valid bootstrap image from various storage sources, and load it into RAM. The RAM configuration is described in a CPU-specific header, prepended to the bootloader image.*
***2- U-Boot:** runs from RAM. Initializes some other hardware devices (network, USB, etc.). Loads the kernel image from storage or network to RAM and starts it. Shell with commands provided. File called u-boot.kwb*
***3- Linux Kernel:** runs from RAM. Takes over the system completely.*

***⇒ Generic bootloaders for embedded CPUs:***

*- We will focus on the generic part, the main bootloader, offering the most important features.*
*- There are several open-source generic bootloaders. Here are the most popular ones:*

- ***U-Boot:** the universal bootloader by Denx. The most used on ARM, also used on PPC, MIPS, x86, m68k, RiscV, etc.*
- ***Barebox:** an architecture-neutral bootloader created by Pengutronix. It doesn't have as much hardware support as U-Boot yet.*

## *The U-boot bootloader*

***U-Boot configuration:***

*- Get the source code from the website or from git, and uncompress it.*
*- The configs/ directory contains one or several configuration file(s) for each supported board.*

- *It defines the CPU type, the peripherals and their configuration, the memory mapping, the U-Boot features that should be compiled in, etc.*
- *Examples:*
  - *configs/stm32mp15_basic_defconfig*
  - *configs/stm32mp15_trusted_defconfig*

*- Note: U-Boot is migrating from board configuration defined in C header files (include/configs/) to defconfig like in the Linux kernel (configs/):*

- *Many boards still have both hardcoded configuration settings in .h files, and configuration settings in defconfig files that can be overridden with configuration interfaces.*

***Configuring and compiling U-Boot***

*- U-Boot must be configured before being compiled*

- *Configuration stored in a .config file*
- *make BOARDNAME_defconfig*
- *Where BOARDNAME is the name of a configuration, as visible in the configs/ directory.*
- *You can then run: make menuconfig to further customize U-Boot configuration!*

*- Make sure that the cross-compiler is available in PATH*

*- Compile U-Boot, by specifying the cross-compiler prefix.*

*Example: if your cross-compiler executable is arm-linux-gcc: make CROSS_COMPILE=arm-linux-*

*⇒ The main result is a **u-boot.bin** file, which is the U-Boot image.*

*- Depending on your specific platform, or what storage device you're booting from (NAND or MMC), there may be other specialized images: u-boot.img, u-boot.kwb...*

*- This also generates the U-Boot SPL image to be flashed together with U-Boot.*

***Installing U-Boot:***

*- U-Boot must usually be installed in flash memory to be executed by the hardware. Depending on the hardware, the installation of U-Boot is done in a different way:*

- *The CPU provides some kind of specific boot monitor with which you can communicate through the serial port or USB using a specific protocol.*
- *The CPU boots first on removable media (MMC) before booting from fixed media (NAND). In this case, boot from MMC to reflash a new version.*
- *U-Boot is already installed, and can be used to flash a new version of U-Boot. However, be careful: if the new version of U-Boot doesn't work, the board is unusable.*
- *The board provides a JTAG interface, which allows it to write to the flash memory remotely, without any system running on the board. It also allows you to rescue a board if the bootloader doesn't work.*

***Information commands:***

*- Flash information (NOR and SPI flash): $ flinfo*

*- NAND flash information: $ nand info*

*- Version details: $ version*

*- loads a file from a FAT filesystem to RAM: $ fatload*

- *Example: **fatload usb 0:1 0x21000000 zImage***

*Also other commands: fatinfo, fatls, fatsize, fatwrite...*

*- Loads a file from an ext2 filesystem to RAM: $ ext2load*

*also ext2info, ext2ls, ext2size, ext2write...*

*- Similar commands for other filesystems: ext4load, ext4ls, sqfsload, sqfsls...*

*- Loads a file from the network to RAM: $ tftp*

*- Test the network: $ ping*

*- Runs the default boot command, stored in bootcmd: $ bootd (can be abbreviated as boot).*

*- Starts a kernel image loaded at the given address in RAM: $ bootz <address>*

*- Load a file from the serial line to RAM: $ loadb, loads, loady*

*- Initialize and control the USB subsystem, mainly used for USB storage devices such as USB keys: $ usb*

*- Initialize and control the MMC subsystem, used for SD and microSD cards: $ mmc*

*- Erase, read and write contents to NAND flash: $ nand*

*- Erase, modify protection and write to NOR flash: $ erase, protect, cp*

*- displays memory contents: $ md*

*⇒ Can be useful to check the contents loaded in memory, or to look at hardware registers.*

*- Modifies memory contents: $ mm*

*⇒ Can be useful to modify directly hardware registers, for testing purposes.*

***U-Boot bdinfo command:***

*- Allow to find valid RAM addresses without needing the SoC datasheet or board manual: $ bdinfo*

***Environment variables: principle***

*- Environment variables are loaded from persistent storage to RAM at U-Boot startup. They can be defined or modified and saved back to storage for persistence.*

*- Depending on the configuration, the U-Boot environment is typically stored in:*

- *At a fixed offset in NAND flash.*
- *At a fixed offset on MMC or USB storage, before the beginning of the first partition.*
- *In a file (uboot.env) on a FAT or ext4 partition.*

***Environment variables commands:***

*- Commands to manipulate environment variables:*

- *printenv: Shows all variables*
- *printenv <variable-name>: Shows the value of a variable*
- *setenv <variable-name> <variable-value>: Changes the value of a variable or defines a new one, only in RAM*
- *editenv <variable-name>: Edits the value of a variable, only in RAM*
- *saveenv: Saves the current state of the environment to storage*

***Important U-Boot env variables***

*- bootcmd: specifies the commands that U-Boot will automatically execute at boot time after a configurable delay (bootdelay), if the process is not interrupted.*

*- bootargs: contains the arguments passed to the Linux kernel, covered later.*

*- serverip: the IP address of the server that U-Boot will contact for network related commands.*

*- ipaddr: the IP address that U-Boot will use*

*- netmask: the network mask to contact the server*

*- ethaddr: the MAC address, can only be set once*

*- autostart: if set to yes, U-Boot automatically tries to execute a binary after loading it in memory (tftp, fatload...)*

*- filesize: the size of the latest copy to memory (from tftp, fatload, nand read...)*

***Scripts in environment variables:***

*- Environment variables can contain small scripts, to execute several commands and test the results of commands.*

*- Examples:*

- *setenv bootcmd 'tftp 0x21000000 zImage; tftp 0x22000000 dtb; bootz 0x21000000 - 0x22000000'*
- *setenv mmc-boot 'if fatload mmc 0 80000000 boot.ini; then source; else if fatload mmc 0 80000000 zImage; then run mmc-do-boot; fi; fi'*

***Transferring files to the target***

*- U-Boot is mostly used to load and boot a kernel image, but it also allows to change the kernel image and the root filesystem stored in flash.*

*- Files must be exchanged between the target and the development workstation. This is possible:*

- *Through the network (Ethernet if a network port is available, Ethernet over USB device...), if U-Boot has drivers for such networking. This is the fastest and most efficient solution.*
- *Through a USB key, if U-Boot supports the USB controller of your platform*
- *Through a SD or microSD card, if U-Boot supports the MMC controller of your platform.*
- *Through the serial port (loadb, loadx or loady command)*

***TFTP:***

*- Network transfer from the development workstation to U-Boot on the target takes place through TFTP.*

*- A **TFTP server** is needed on the development workstation*

- *sudo apt install tftpd-hpa*
- *All files in /var/lib/tftpboot or in /srv/tftp (if /srv exists) are then visible through TFTP*
- *A TFTP client is available in the tftp-hpa package, for testing*

*- **TFTP client** is integrated into U-Boot*

- *Configure the ipaddr, serverip, and ethaddr environment variables*
- *Use tftp <address> <filename> to load file contents to the specified RAM address*
- *Example: tftp 0x21000000 zImage.*

## ***Linux Kernel:***

*Linux kernel main roles:*

*- Manage all the hardware resources: CPU, memory, I/O.*

*- Provide a set of portable, architecture and hardware independent APIs to allow user space applications and libraries to use the hardware resources. Handle concurrent accesses and usage of hardware resources from different applications.*

***Example:*** *a single network interface is used by multiple user space applications through various network connections. The kernel is responsible to "multiplex" the hardware resource.*

- *Systems Call: The main interface between the kernel and user space is the set of system calls*
- *Pseudo filesystems: Linux makes system and kernel information available in user space through pseudo filesystems, sometimes also called virtual filesystems.*

*- Pseudo filesystems allow applications to see directories and files that do not exist on any real storage: they are created and updated on the fly by the kernel.*
*The two most important pseudo filesystems are:*

- *proc, usually mounted on /proc: Operating system related information (processes, memory management parameters...)*
- *sysfs, usually mounted on /sys: Representation of the system as a tree of devices connected by buses. Information gathered by the kernel frameworks managing these devices.*

***Building the kernel***
*Kernel configuration:*
*- The kernel contains thousands of device drivers, filesystem drivers, network protocols and other configurable items.*
*- The kernel configuration is the process of defining the set of options with which you want your kernel to be compiled.*
*The set of options depends*

- *On the target architecture and on your hardware (for device drivers, etc.)*
- *On the capabilities you would like to give to your kernel (network capabilities, filesystems, real-time, etc.). Such generic options are available in all architectures.*

***Kernel configuration and build system:***
*- The kernel configuration and build system is based on multiple Makefiles.*
*- One only interacts with the main Makefile, present at the top directory of the kernel source tree.*
*- Interaction takes place:*

- *using the make tool, which parses the Makefile*
- *through various targets, defining which action should be done (configuration, compilation, installation, etc.).*

*- The configuration is stored in the .config file at the root of kernel sources:*

- *Simple text file, CONFIG_PARAM=value (included by the kernel Makefile)*

*- As options have dependencies, typically never edited by hand, but through graphical or text interfaces:*

- *make xconfig, make gconfig (graphical)*
- *make menuconfig, make nconfig (text)*
- *You can switch from one to another, they all load/save the same .config file, and show the same set of options.*

***Initial configuration:***
*- Difficult to find which kernel configuration will work with your hardware and root filesystem.*
*Desktop or server case:*

- *Advisable to start with the configuration of your running kernel,usually available in /boot:*
  *cp /boot/config-`uname -r` .config.*

*Embedded platform case:*

- *Default configuration files are available, usually for each CPU family.*
- *They are stored in arch/<arch>/configs/, and are just minimal .config files (only settings different from default ones).*
- *Run make help to find if one is available for your platform*
- *To load a default configuration file, just run: make cpu_defconfig*
- *This will overwrite your existing .config file!*

*⇒ Now, you can make configuration changes (make menuconfig...).*
***Create your own default configuration:***
*- To create your own default configuration file:*

- *make savedefconfig: This creates a minimal configuration (non-default settings)*
- *mv defconfig arch/<arch>/configs/myown_defconfig: This way, you can share a reference configuration inside the kernel sources.*

***Kernel or module:***
*- The kernel image is a single file, resulting from the linking of all object files that correspond to features enabled in the configuration:*

- *This is the file that gets loaded in memory by the bootloader*
- *All included features are therefore available as soon as the kernel starts, at a time where no filesystem exists*

*- Some features (device drivers, filesystems, etc.) can however be compiled as modules:*

- *These are plugins that can be loaded/unloaded dynamically to add/remove features to the kernel*
- *Each module is stored as a separate file in the filesystem, and therefore access to a filesystem is mandatory to use modules*

***Kernel option types:***

*-There are different types of options, defined in Kconfig files:*

*bool options: they are either*

- *true (to include the feature in the kernel) or*
- *false (to exclude the feature from the kernel)*

*Tristate options: they are either*

- *true (to include the feature in the kernel image) or*
- *module (to include the feature as a kernel module) or*
- *false (to exclude the feature)*

*int options: to specify integer values*

*hex options: to specify hexadecimal values*

***Example:*** *CONFIG_PAGE_OFFSET=0xC0000000*

*string options: to specify string values*

***Example:*** *CONFIG_LOCALVERSION=-no-network*

***Kernel option dependencies***

*There are dependencies between kernel options*

*- For example, enabling a network driver requires the network stack to be enabled*

*- Two types of dependencies:*

- *depends on dependencies. In this case, option B that depends on option A is not visible until option A is enabled.*
- *select dependencies. In this case, with option B depending on option A, when option A is enabled, option B is automatically enabled. In particular, such dependencies are used to declare what features a hardware architecture supports.*

*⇒ With the Show All Options option, make xconfig allows to see all options, even the ones that cannot be selected because of missing dependencies. Values for dependencies are shown.*

***Compiling and installing kernel:***

*Choose compiler:*

*The compiler invoked by the kernel Makefile is $(CROSS_COMPILE)gcc*

- *When compiling natively:*
  *Leave CROSS_COMPILE undefined and the kernel will be natively compiled for the host architecture using gcc.*
- *When using a cross-compiler:*
  *To make the difference with a native compiler, cross-compiler executables are prefixed by the name of the target system,architecture and sometimes library.*

*Examples:*

- *mips-linux-gcc: the prefix is mips-linux-*
- *arm-linux-gnueabi-gcc: the prefix is arm-linux-gnueabi-*

*⇒ So, you can specify your cross-compiler as follows: export CROSS_COMPILE=arm-linux-gnueabi-*

*- CROSS_COMPILE is actually the prefix of the cross compiling tools (gcc, as, ld, objcopy, strip...).*

***Specifying ARCH and CROSS_COMPILE:***

*- There are actually two ways of defining ARCH and CROSS_COMPILE:*

- *Pass ARCH and CROSS_COMPILE on the make command line:*
  *make ARCH=arm CROSS_COMPILE=arm-linux- ...*
- *Define ARCH and CROSS_COMPILE as environment variables:*
  *export ARCH=arm*
  *export CROSS_COMPILE=arm-linux-*

***Kernel compilation:***

*make:*

- *Run it In the main kernel source directory!*
- *Remember to run multiple jobs in parallel if you have multiple CPU cores. Our advice: ncpus * 2 or ncpus + 2, to fully load the CPU and I/Os at all times.*
  *Example: make -j 8*
- *To recompile faster (7x according to some benchmarks), use the ccache compiler cache:*
  *export CROSS_COMPILE="ccache riscv64-linux-"*

***Kernel compilation results:***

- ***vmlinux:*** *the raw uncompressed kernel image, in the ELF format, useful for debugging purposes, but cannot be booted*

- ***arch/<arch>/boot/*Image:*** *the final, usually compressed, kernel image that can be booted: Example: bzImage for x86, zImage for ARM, vmlinux.bin.gz for ARC, etc.*
- ***arch/<arch>/boot/dts/*.dtb:*** *compiled Device Tree files (on some architectures)*

*-All kernel modules, spread over the kernel source tree, as .ko (Kernel Object) files.*

***Kernel installation: native case***

*make install: Does the installation for the host system by default, so needs to be run as root.*

*Result of Installs:*

- ***/boot/vmlinuz-<version>:*** *Compressed kernel image. Same as the one in arch/<arch>/boot*
- ***/boot/System.map-<version>:*** *Stores kernel symbol addresses for debugging purposes (obsolete: such information is usually stored in the kernel itself)*
- ***/boot/config-<version>:*** *Kernel configuration for this version*

***Kernel installation: embedded case***

*- make install is rarely used in embedded development, as the kernel image is a single file, easy to handle.*

*- Another reason is that there is no standard way to deploy and use the kernel image.*

*- Therefore making the kernel image available to the target is usually manual or done through scripts in build systems.*

*- It is however possible to customize the make install behavior in arch/<arch>/boot/install.sh*

***Module installation: native case:***

*make modules_install: Does the installation for the host system by default, so needs to be run as root*

*- Installs all modules in /lib/modules/<version>/*

- ▸ *kernel/ :*
    - *Module .ko: (Kernel Object) files, in the same directory structure as in the sources.*
- ▸ *modules.alias, modules.alias.bin:*
    - *Aliases for module loading utilities. Used to find drivers for devices. Example line: alias usb:v066Bp20F9d*dc*dsc*dp*ic*isc*ip*in* asix*
- ▸ *modules.dep, modules.dep.bin:*
    - *Module dependencies*
- ▸ *modules.symbols, modules.symbols.bin:*
    - *Tells which module a given symbol belongs to.*

***Module installation: embedded case***

*- In embedded development, you can't directly use: make modules_install as it would install target modules in /lib/modules on the host!*

*- The INSTALL_MOD_PATH variable is needed to generate the module related files and install the modules in the target root filesystem instead of your host root filesystem: make INSTALL_MOD_PATH=<dir>/ modules_install*

***Kernel cleanup targets:***

*make clean*

*- Clean-up generated files (to force re-compilation).*

*- Remove all generated files. Needed when switching from one architecture to another.*

*Caution: it also removes your .config file:*

*make mrproper*

*- Also remove editor backup and patch reject files (mainly to generate patches): make distclean*

*- If you are in a git tree, remove all files not tracked (and ignored) by git: git clean -fdx*

***Booting the kernel:***

*- Device Tree:*

*A Device Tree Source, written by kernel developers, is compiled into a binary Device Tree Blob, and needs to be passed to the kernel at boot time.*

- *There is one different Device Tree for each board/platform supported by the kernel, available in arch/arm/boot/dts/<board>.dtb.*
- *See arch/arm/boot/dts/at91-sama5d3_xplained.dts for example.*

*⇒ The bootloader must load both the kernel image and the Device Tree Blob in memory before starting the kernel.*

***Booting with U-Boot:***

*-Recent versions of U-Boot can boot the zImage binary.*

*-Older versions require a special kernel image format: uImage*

- *uImage is generated from zImage using the mkimage tool. It is done automatically by the kernel make uImage target.*
- *On some ARM platforms, make uImage requires passing a LOADADDR environment variable, which indicates at which physical memory address the kernel will be executed.*

*-In addition to the kernel image, U-Boot can also pass a Device Tree Blob to the kernel.*

*-The typical boot process is therefore:*

*1. Load zImage or uImage at address X in memory*
*2. Load <board>.dtb at address Y in memory*
*3. Start the kernel with bootz X - Y (zImage case), or bootm X - Y (uImage case)*
*The - in the middle indicates no initramfs*

***Using kernel modules***

*Advantages of modules*

*- Modules make it easy to develop drivers without rebooting: load, test, unload, rebuild, load...*
*- Useful to keep the kernel image size to the minimum (essential in GNU/Linux distributions for PCs).*
*- Also useful to reduce boot time: you don't spend time initializing devices and kernel features that you only need later.*
*- Caution: once loaded, have full control and privileges in the system. No particular protection. That's why only the root user can load and unload modules.*
*- To increase security, possibility to allow only signed modules, or to disable module support entirely.*

***Module dependencies:***

*- Some kernel modules can depend on other modules, which need to be loaded first.*

- *Example: the ubifs module depends on the ubi and mtd modules.*

*- Dependencies are described both in*
*/lib/modules/<kernel-version>/modules.dep and in*
*/lib/modules/<kernel-version>/modules.dep.bin (binary hashed format)*
*- These files are generated when you run make modules_install.*

***kernel log:***

*- When a new module is loaded, related information is available in the kernel log.The kernel keeps its messages in a circular buffer (so that it doesn't consume more memory with many messages)*
*- Kernel log messages are available through the dmesg command (diagnostic message). Kernel log messages are also displayed in the system console (console messages can be filtered by level using the loglevel kernel command line parameter, or completely disabled with the quiet parameter).*

- *Example: console=ttyS0 root=/dev/mmcblk0p2 loglevel=5*

*- Note that you can write to the kernel log from user space too: echo "<n>Debug info" > /dev/kmsg*

***Module utilitie:***

*<module_name>: name of the module file without the trailing .ko*

- *modinfo <module_name> (for modules in /lib/modules):*
  *modinfo <module_path>.ko: Gets information about a module without loading it: parameters, license, description and dependencies.*
- *sudo insmod <module_path>.ko: Tries to load the given module. The full path to the module object file must be given*
- *sudo modprobe <module_name>: Most common usage of modprobe: tries to load all the modules the given module depends on, and then this module. Lots of other options are available. modprobe automatically looks in /lib/modules/<version>/ for the object file corresponding to the given module name.*
- *lsmod:Displays the list of loaded modules. Compare its output with the contents of /proc/modules!*
- *sudo rmmod <module_name>: Tries to remove the given module. Will only be allowed if the module is no longer in use (for example, no more processes opening a device file)*
- *sudo modprobe -r <module_name>: Tries to remove the given module and all dependent modules (which are no longer needed after removing the module)*

***Passing parameters to modules***

*- Find available parameters:*

- *modinfo usb-storage*

*- Through insmod:*

- *sudo insmod ./usb-storage.ko delay_use=0*

*- Through modprobe:*

- *Set parameters in /etc/modprobe.conf or in any file in /etc/modprobe.d/: options usb-storage delay_use=0*

*- Through the kernel command line, when the driver is built statically into the kernel: usb-storage.delay_use=0:*

- *usb-storage is the driver name*
- *delay_use is the driver parameter name. It specifies a delay before accessing a USB storage device (useful for rotating devices).*
- *0 is the driver parameter value*

***Check module parameter values***

*How to find/edit the current values for the parameters of a loaded module?*

- *Check /sys/module/<name>/parameters.*
- *There is one file per parameter, containing the parameter value.*
- *Also possible to change parameter values if these files have write permissions (depends on the module code).*
- *Example: echo 0 > /sys/module/usb_storage/parameters/delay_use*

## ***Linux Root Filesystem***

***FileSystems:***

*-Create a mount point, which is just a directory: $ sudo mkdir /mnt/usbkey*

*-It is empty: $ ls /mnt/usbkey*

*-Mount a storage device in this mount point: $ sudo mount -t vfat /dev/sda1 /mnt/usbkey*

*-You can access the contents of the USB key: $ ls /mnt/usbkey*

***mount / umount:***

*-mount allows to mount filesystems*

- *mount -t type device mountpoint*
- *type is the type of filesystem (optional for non-virtual filesystems)*
- *device is the storage device, or network location to mount*
- *mountpoint is the directory where files of the storage device or network location will be accessible*
- *mount with no arguments shows the currently mounted filesystems*

*-umount allows to unmount filesystems*

- *This is needed before rebooting, or before unplugging a USB key, because the Linux kernel caches writes in memory to increase performance. umount makes sure that these writes are committed to the storage.*

***Root Filesystems:***

*-A particular filesystem is mounted at the root of the hierarchy, identified by /. This filesystem is called the root filesystem.*

*-As mount and umount are programs, they are files inside a filesystem. They are not accessible before mounting at least one filesystem.*

*-As the root filesystem is the first mounted filesystem, it cannot be mounted with the normal mount command.*

*⇒ It is mounted directly by the kernel, according to the root= kernel option.*

*-When no root file system is available, the kernel panics.*

***Mounting rootfs from storage devices***

*-Partitions of a hard disk or USB key*

- *root=/dev/sdXY: where X is a letter indicating the device, and Y a number indicating the partition*
- */dev/sdb2 is the second partition of the second disk drive (either USB key or ATA hard drive)*

*-Partitions of an SD card*

- *root=/dev/mmcblkXpY: where X is a number indicating the device and Y a number indicating the partition*
- */dev/mmcblk0p2: is the second partition of the first device*

*-Partitions of flash storage*

- *root=/dev/mtdblockX: where X is the partition number*
- */dev/mtdblock3: is the fourth partition of a NAND flash chip (if only one NAND flash chip is present)*

***Mounting rootfs over the network***

*-Once networking works, your root filesystem could be a directory on your GNU/Linux development host, exported by NFS (Network File System)*

*-On the development workstation side, a NFS server is needed*

- *Install an NFS server (example: Debian, Ubuntu): sudo apt install nfs-kernel-server*
- *Add the exported directory to your /etc/exports file: /home/tux/rootfs*

*-192.168.1.111(rw,no_root_squash,no_subtree_check)*

- *192.168.1.111 is the client IP address*
- *rw,no_root_squash,no_subtree_check are the NFS server options for this directory export.*

*-Start or restart your NFS server (example: Debian, Ubuntu): sudo /etc/init.d/nfs-kernel-server restart*

*On the target system:*

*-The kernel must be compiled with*

- *CONFIG_NFS_FS=y (NFS client support)*
- *CONFIG_IP_PNP=y (configure IP at boot time)*
- *CONFIG_ROOT_NFS=y (support for NFS as rootfs)*

*-The kernel must be booted with the following parameters:*

- *root=/dev/nfs (we want rootfs over NFS)*
- *ip=192.168.1.111 (target IP address)*
- *nfsroot=192.168.1.110:/home/tux/rootfs/ (NFS server details)*
- *You may need to add ",nfsvers=3,tcp" to the nfsroot setting, as an NFS version 2 client and UDP may be rejected by the NFS server in recent GNU/Linux distributions.*

***rootfs in memory: initramfs***

*-It is also possible to have the root filesystem integrated into the kernel image. It is therefore loaded into memory together with the kernel. This mechanism is called initramfs*
*-It is useful for two cases:*

- *Fast booting of very small root filesystems. As the filesystem is completely loaded at boot time, application startup is very fast.*
- *As an intermediate step before switching to a real root filesystem, located on devices for which drivers not part of the kernel image are needed (storage drivers, filesystem drivers, network drivers). This is always used on the kernel of desktop/server distributions to keep the kernel image size reasonable.*

*-The contents of an initramfs are defined at the kernel configuration level, with the CONFIG_INITRAMFS_SOURCE option.*
*-The kernel build process will automatically take the contents of the CONFIG_INITRAMFS_SOURCE option and integrate the root filesystem into the kernel image.*

***Separation of programs and libraries***

*-Basic programs are installed in /bin and /sbin and basic libraries in /lib. All other programs are installed in /usr/bin and /usr/sbin and all other libraries in /usr/lib.*
*-In the past, on UNIX systems, /usr was very often mounted over the network, through NFS*
*-In order to allow the system to boot when the network is down, some binaries and libraries are stored in /bin, /sbin and /lib.*

- */bin and /sbin contain programs like ls, ip, cp, bash, etc.*
- */lib contains the C library and sometimes a few other basic libraries*
- *All other programs and libraries are in /usr*

## ***Device File:***

***Device:***

*-In the Linux kernel, most devices are presented to user space applications through two different abstractions:*

- *Character device*
- *Block device*

*-Internally, the kernel identifies each device by a triplet of information*

- *Type (character or block)*
- *Major (typically the category of device)*
- *Minor (typically the identifier of the device)*

***Types of devices:***

*- Block devices: A device composed of fixed-sized blocks, that can be read and written to store data. Used for hard disks, USB keys, SD cards*
*- Character devices*

- *Originally, an infinite stream of bytes, with no beginning, no end, no size. The pure example: a serial port.*
- *Used for serial ports, terminals, but also sound cards, video acquisition devices, frame buffers*
- *Most of the devices that are not block devices are represented as character devices by the Linux kernel*

*⇒ Devices: everything is a file*
*- A very important UNIX design decision was to represent most system objects as files.It allows applications to manipulate all system objects with the normal file API (open, read, write, close, etc.). So, devices had to be represented as files to the applications.*
*- This is done through a special artifact called a device file: It is a special type of file, that associates a file name visible to user space applications to the triplet (type, major, minor) that the kernel understands.*
*- All device files are by convention stored in the /dev directory*

***Creating device files***

*- The device files had to be created manually using the mknod command:*

- *mknod /dev/<device> [c|b] major minor*

*-The devtmpfs virtual filesystem can be mounted on /dev and contains all the devices known to the kernel. The CONFIG_DEVTMPFS_MOUNT kernel configuration option makes the kernel mount it automatically at boot time, except when booting on an initramfs.*

***Pseudo Filesystems:***

***proc virtual filesystem:***

*- It allows*

- *The kernel to expose statistics about running processes in the system*
- *The user can adjust at runtime various system parameters about process management, memory management, etc.*

*- The proc filesystem is used by many standard user space applications, and they expect it to be mounted in /proc. Applications such as ps or top would not work without the proc filesystem*

- *Command to mount proc: mount -t proc nodev /proc*

***proc contents***

*- One directory for each running process in the system: /proc/<pid>. It contains details about the files opened by the process, the CPU and memory usage, etc.*

*- /proc/interrupts, /proc/devices, /proc/iomem, /proc/ioports contain general device-related information*

*- /proc/cmdline contains the kernel command line*

*- /proc/sys contains many files that can be written to adjust kernel parameters*

***sysfs filesystem:***

*- It allows us to represent in user space the vision that the kernel has of the buses, devices and drivers in the system. It is useful for various user space applications that need to list and query the available hardware, for example udev or mdev*

*- All applications using sysfs expect it to be mounted in the /sys directory Command to mount /sys: mount -t sysfs nodev /sys*

***Minimal filesystem***

*- Basic applications:*

*An init application, which is the first user space application started by the kernel after mounting the root filesystem:*

- ▸ *The kernel tries to run the command specified by the init= command line parameter if available.*
- ▸ *Otherwise, it tries to run /sbin/init, /bin/init, /etc/init and /bin/sh.*
- ▸ *In the case of an initramfs, it will only look for /init. Another path can be supplied by the rdinit= kernel argument.*
- ▸ *If none of this works, the kernel panics and the boot process is stopped.*
- ▸ *The init application is responsible for starting all other user space applications and services.*

*Basic UNIX executables, for use in system scripts or in interactive shells: mv, cp, mkdir, cat, modprobe, mount, ip, etc.*

*⇒ These basic components have to be integrated into the root filesystem to make it usable*

*- Overall booting process:*

*1. Bootloader:*

- *loads the DTB and kernel to RAM, starts the kernel.*

*2. Kernel:*

- *Initializes hardware device and kernel subsystems.*
- *Mount the root filesystem indicated by root=*
- *Start the init application, /sbin/init by default.*

*3. Root filesystems: /sbin/init:*

- *Stat other user space service and application: shell + other application*

*- Overall booting process with initramfs:*

*1. Bootloader:*

- *loads the initramfs archive to the RAM.*
- *loads the DTB and kernel to RAM, starts the kernel.*

*2. Kernel:*

- *Initializes hardware device and kernel subsystems.*
- *Extract the initramfs archive to the file cache.*
- *Start the /init executable if found.*

*3. Initramfs: /Init:*

- *Start early user space commands.*
- *Load drivers needed to access the final root filesystems.*
- *Mount the root filesystems and switch to it.*

*4. Root filesystems: /sbin/init:*

- *Regular systems startup*

## ***Busybox:***

*- A Linux system needs a basic set of programs to work*

- *An init program*
- *A shell*
- *Various basic utilities for file manipulation and system configuration*

*⇒ Busybox is an alternative solution, extremely common on embedded systems.*

*- BusyBox in the root filesystem: All the utilities are compiled into a single executable, /bin/busybox.*

***Configuring BusyBox:***

*- Get the latest stable sources from https://busybox.net*

*- Configure BusyBox (creates a .config file):*

- *make defconfig: Configures BusyBox with all options for regular users.*
- *make allnoconfig: Unselects all options. Good to configure only what you need.*
- *make menuconfig (text): Same configuration interfaces as the ones used by the Linux kernel.*

***Compiling BusyBox***

*- Set the cross-compiler prefix in the configuration interface:*

- *Settings -> Build Options -> Cross Compiler prefix*
- *Example prefix: arm-linux-*

*- Set the installation directory in the configuration interface:*

- *Settings -> Installation Options -> BusyBox installation prefix.*

*- Add the cross-compiler path to the PATH environment variable:*

- *export PATH=$HOME/x-tools/arm-unknown-linux-uclibcgnueabi/bin:$PATH*

*- Compile BusyBox:*

- *make*

*- Install it (this creates a UNIX directory structure symbolic links to the busybox executable):*

- *make install*

***Busybox init:***

*- Busybox provides an implementation of an init program. Simpler than the init implementation found on desktop/server systems (SysV init or systemd).*

*- A single configuration file: /etc/inittab*

- *Each line has the form <id>::<action>:<process>*

*⇒ Allows to run services at startup, and to make sure that certain services are always running on the system.*

## ***Block filesystems:***

*- Block vs. flash: Storage devices are classified in two main types: block devices and flash devices.*

*- Block devices can be read and written to on a per-block basis, in random order, without erasing: Hard Disks, Ram Disks.*

*- Raw flash devices are driven by a controller on the SoC. They can be read, but writing requires prior erasing, and often occurs on a larger size than the "block" size: NOR flash, NAND flash.*

*- Block device list: The list of all block devices available in the system can be found in /proc/partitions:*

- *$ cat /proc/partitions*

*- /sys/block/ also stores information about each block device, for example whether it is removable storage or not.*

***Partitioning:***

*- Block devices can be partitioned to store different parts of a system. The partition table is stored inside the device itself, and is read and analyzed automatically by the Linux kernel:*

- *mmcblk0 is the entire device*
- *mmcblk0p2 is the second partition of mmcblk0*

*- Numerous tools to create and modify the partitions on a block device: fdisk, cfdisk, sfdisk, parted, etc.*

***Transfering data to a block device***

*- dd (disk duplicate) is the tool of choice for such transfers:*

- *dd if=/dev/mmcblk0p1 of=testfile bs=1M count=16*
  *Transfers 16 blocks of 1 MB from /dev/mmcblk0p1 to testfile*
- *dd if=testfile of=/dev/sda2 bs=1M seek=4:*
  *Transfers the complete contents of testfile to /dev/sda2, by blocks of 1 MB, but starting at offset 4 MB in /dev/sda2*
- *Typical mistake: copying a file to a filesystem without mounting it first:*
  *dd if=zImage of=/dev/sda1*
  *⇒ Wrong command !!!!*

- *Instead, you should use:*
  ```
  sudo mount /dev/sda1 /boot
  cp zImage /boot/
  ```

***Available filesystems***

*- Standard Linux filesystem format: ext2, ext3, ext4:*

*- The standard filesystem used on Linux systems is the series of ext{2,3,4} filesystems.It supports all features Linux needs in a root filesystem: permissions, ownership, device files, symbolic links, etc.*

*- Filesystem recovery after crashes:*

*Thanks to the journal, the recovery at boot time is quick, since the operations in progress at the moment of the unclean shutdown are clearly identified.*

*Does not mean that the latest writes made it to the storage: this depends on syncing the changes to the filesystem.*

*SquashFS: read-only filesystem:*

- *Read-only, compressed filesystem for block devices. Fine for parts of a filesystem which can be read-only (kernel, binaries...)*

*Tmpfs: filesystem in RAM*

- *Perfect to store temporary data in RAM: system log files, connection data, temporary files...More space-efficient than ramdisks*
- *How to use: choose a name to distinguish the various tmpfs instances you could have.*
  *Examples:*
  ```
  mount -t tmpfs run /var/run
  mount -t tmpfs shm /dev/shm
  ```

***Using block filesystems***

*Creating ext2/ext4 filesystems:*

*- To create an empty ext2/ext4 filesystem on a block device or inside an already-existing image file:*

- *mkfs.ext2 /dev/hda3*
- *mkfs.ext4 /dev/sda3*
- *mkfs.ext2 disk.img*

*- To create a filesystem image from a directory containing all your files and directories*

- *Use the genext2fs tool, from the package of the same name*
- *genext2fs -d rootfs/ rootfs.img*
- *Your image is then ready to be transferred to your block device*

***Mounting filesystem images:***

*-Once a filesystem image has been created, one can access and modifies its contents from the development workstation, using the loop mechanism:*

- *Example:*
  ```
  genext2fs -d rootfs/ rootfs.img
  mkdir /tmp/tst
  mount -t ext2 -o loop rootfs.img /tmp/tst
  ```

*- In the /tmp/tst directory, one can access and modify the contents of the rootfs.img file.*

*⇒ This is possible thanks to loop, which is a kernel driver that emulates a block device with the contents of a file.*

***Creating squashfs filesystems***

*- Need to install the squashfs-tools package*

*- Can only create an image: creating an empty squashfs filesystem would be useless, since it's read-only.*

*- To create a squashfs image:*

- *mksquashfs rootfs/ rootfs.sqfs -noappend*
- *-noappend: re-create the image from scratch rather than appending to it*

*-Mounting a squashfs filesystem:*

- *mount -t squashfs /dev/<device> /mnt*

***Mixing read-only and read-write filesystems***

*Good idea to split your block storage into:*

- *A compressed read-only partition (SquashFS) Typically used for the root filesystem (binaries, kernel...). Compression saves space. Read-only access protects your system from mistakes and data corruption.*
- *A read-write partition with a journaled filesystem (like ext4) Used to store user or configuration data. Guarantees filesystem integrity after power off or crashes.*
- *Ram storage for temporary files (tmpfs)*

## Flash Storage ad file systems:

***The MTD subsystem***

*- MTD stands for Memory Technology Devices*

*- Generic subsystem in Linux dealing with all types of storage media that are not fitting in the block subsystem. Supported media types: RAM, ROM, NOR flash, NAND flash, Dataflash. Independent of the communication interface (drivers available for parallel, SPI, direct memory mapping, ...). Abstract storage media characteristics and provide a simple API to access MTD devices.*

*- MTD device characteristics exposed to users:*

- *erasesize: minimum erase size unit*
- *writesize: minimum write size unit*
- *oobsize: extra size to store metadata or ECC data*
- *size: device size*
- *flags: information about device type and capabilities*

***MTD partitioning***

*- MTD devices are usually partitioned: It allows to use different areas of the flash for different purposes: read-only filesystem,  read-write filesystem, backup areas, bootloader area, kernel area, etc.*

*- Each partition becomes a separate MTD device*

- *Different from block device labeling (hda3, sda2)*
- */dev/mtd1 is either the second partition of the first flash device, or the first partition of the second flash device*
- *Note that the master MTD device (the device those partitions belongs to) is not exposed in /dev*

*- Linux: definition of MTD partitions:*

*The Device Tree is the standard place to define default MTD partitions and MTD device properties (such as ECC) for platforms with Device Tree support.*

*- U-Boot: definition of MTD partition:*

*U-Boot allows to define MTD partitions on flash devices, using the same syntax as Linux for declaring them. Sharing definitions allows to eliminate the risk of mismatches between Linux and U-Boot. Named partitions are also easier to use, and much less error prone than using offsets. Use flash specific commands (detailed soon), and pass partition names instead of numerical offsets.*

*▸ Example:*

```
# Association between U-Boot name and Linux name
setenv mtdids nand0=omap2-nand.0
# Partition definitions
setenv mtdparts mtdparts=omap2-nand.0:512k(XLoader)ro,1536k(UBoot)ro,512k(Env),4m(Kernel),-(Root)
```

*- This defines 5 partitions in the omap2-nand.0 device:*

- *1st stage bootloader (512 KiB, read-only)*
- *U-Boot (1536 KiB, read-only)*
- *U-Boot environment (512 KiB)*
- *Kernel (4 MiB)*
- *Root filesystem (Remaining space)*

*⇒  Partition sizes must be multiple of the erase block size. You can use sizes in hexadecimal too.*

- *ro lists the partition as read only*
- *- is used to use all the remaining*

*mtdids associates a U-Boot flash device name to a Linux flash device name:*

```
setenv mtdids <devid>=<mtdid>[,<devid>=<mtdid>]
```

*That's required because the Linux name is used in partition definitions.*

- *devid: U-Boot device identifier (from nand info or flinfo)*
- *mtdid: Linux mtd identifier. Displayed when booting the Linux kernel*

*mtdparts defines partitions for the different devices*

- *setenv mtdparts mtdparts=<mtdid>:<partition>[,partition]*
- *partition format: <size>[@offset](<name>)[ro]*

*- You can use the mtdparts command to check that your partitions definitions are understood correctly by U-Boot.*

*# U-Boot: sharing partition definitions with Linux*

*- Here is a recommended way to pass partition definitions from U-Boot to Linux:*

- *Define a bootargs_base environment variable: setenv bootargs_base console=ttyS0 root=....*
- *Define the final kernel command line (bootargs) through the bootcmd environment variable:*

```
setenv bootcmd 'setenv bootargs ${bootargs_base} ${mtdparts}; <rest of bootcmd>'
```

*#U-Boot: manipulating NAND devices*

- *nand info: Show available NAND devices and characteristics*
- *nand device [dev]: Select or display the active NAND device*
- *nand read[.option] <addr> <offset|partname> <size>: Read data from NAND*
- *nand erase <offset> <size>: Erase a NAND region*
- *nand erase.part <partname>: Erase a NAND partition*
- *nand write[.option] <addr> <offset|partname> <size>: Write data on NAND.*
  *Use nand write.trimffs to avoid writing empty pages (those filled with 0xff)*
- *nand bad: Shows bad blocks*

*#Linux: MTD devices interface with user space:*

*-MTD devices are visible in /proc/mtd*
*-User space only sees MTD partitions, not the flash device under those partitions*
*(unless the kernel is compiled with **CONFIG_MTD_PARTITIONED_MASTER**)*
*-The mtdchar driver creates a character device for each MTD device/partition of the system*

- *Named /dev/mtdX and /dev/mtdXro*
- *Provide ioctl() to erase and manage the flash*
- *Used by the mtd-utils utilities*

*#Linux: user space flash management tools*

*mtd-utils is a set of utilities to manipulate MTD devices*

- *mtdinfo: get detailed information about an MTD device*
- *flash_erase: partially or completely erase a given MTD device*
- *flashcp: write to NOR flash*
- *nandwrite: write to NAND flash*
- *Flash filesysem image creation tools: mkfs.jffs2, mkfs.ubifs, ubinize, etc.*

*-On your workstation: usually available as the mtd-utils package in your distribution.*
*-On your embedded target: most commands now also available in BusyBox.*

*#UBI: Unsorted Block Images*

*-Volume management system on top of MTD devices (similar to what LVM provides for block devices)*
*-Allows to create multiple logical volumes and spread writes across all physical blocks*
*-Takes care of managing the erase blocks and wear leveling. Makes filesystems easier to implement*
*-Volumes can be dynamically resized or, on the opposite, can be read-only (static)*

*#Linux: UBI host tools:*

*-ubinize is the only host tool for the UBI layer*
*-It creates a UBI image to be flashed on an MTD partition, from a specification of the contents of its volumes*

***ubinize configuration file:***

***-** Each section describes a UBI volume*
*- static volumes are meant to store read-only blobs of data, and get the minimum corresponding size. CRC checks are done on them.*
*- A read-only UBIFS filesystem can go in a static volume, but in this case dynamic volumes are best for performance (CRC checking also done at UBIFS level).*
*- autoresize: allows to fill all remaining UBI space.*
*- ubinize command line:*
*ubinize the following arguments:*

- ▸ *-o <output-file-path>: Path to the output image file*
- ▸ *-p <peb-size>: The PEB size (MTD erase block size)*
- ▸ *-m <min-io-size>: The minimum write unit size (MTD write size)*
- ▸ *-s <subpage-size>: Subpage size, only needed if both your flash and your flash controller are supporting subpage writes*
- ▸ *The last argument is the path to the ubinize configuration file*
  - *Example: ubinize -o ubi.img -p 256KiB -m 4096 -s 2048 ubinize.cfg*

*#U-Boot: UBI tools*

*Grouped under the ubi command*

- ▸ *ubi part <part-name>: Attach an MTD partition to the UBI layer. Example: ubi part UBI*
- ▸ *ubi info [layout]: Display UBI device information*
- ▸ *ubi check <vol-name>: Check if a volume exists*
- ▸ *ubi readvol <dest-addr> <vol-name> [<size>]: Read volume contents.*
  - ***Example**: ubi readvol 0x21000000 kernel*

*U-Boot also has commands to modify the UBI volumes:*

- *ubi createvol <vol-name> [<size>] [<type>]: Create a new volume.*
    - ***Example:** ubi createvol initramfs 0x100000 static*
- *ubi removevol <vol-name>: Remove an existing volume*
- *ubi writevol <src-addr> <vol-name> <size>: Write to volume.*
    - ***Example:** tftp 0x21000000 data.ubifs ubi writevol 0x21000000 data ${filesize}*

*U-Boot also has commands to access UBIFS partitions:*

- *ubifsmount <volume-name>: Mount the specified volume. Example: ubifsmount root*
- *ubifsload <addr> <filename> [bytes]: Load file contents at the specified address.*
    - ***Example:** ubifsload 0x21000000 boot/zImage*
- *Other commands: ubifsls and ubifsumount*

*⇒ This shows that to optimize space, the kernel and DTB can also be stored in a UBIFS partition. Otherwise, the DTB volume consumes an entire LEB.*

*#Linux: UBI target tools:*

*-Tools used on the target to dynamically create and modify UBI elements*

***UBI device management:***

- *ubiformat /dev/mtdx: Format an MTD partition and preserve Erase Counter information if any.*
    - ***Example:** ubiformat /dev/mtd1*
- *ubiattach -m <MTD-device-id> /dev/ubi_ctrl: Attach an MTD partition/device to the UBI layer, and create a UBI device.*
    - ***Example:** ubiattach -m 1 /dev/ubi_ctrl*
- *ubidetach -m <MTD-device-id> /dev/ubi_ctrl: Detach an MTD partition/device from the UBI layer, and remove the associated UBI device.*
    - *Example: ubidetach -m 1 /dev/ubi_ctrl*
- *ubinfo -a: Print information about all UBI devices and volumes*

***UBI volume management:***

- *ubimkvol /dev/ubi<UBI-device-id> -N <name> -s <size>*
  *Create a new volume. Use -m in place of -s <size> if you want to assign all the remaining space to this volume.*
    - ***Example**: ubimkvol /dev/ubi0 -N varlog -s 16MiB*
- *ubirmvol /dev/ubi<UBI-device-id> -N <name>*
  *Delete a UBI volume.*
    - ***Example**: ubirmvol /dev/ubi0 -N varlog*
- *ubiupdatevol /dev/ubi<UBI-device-id>_<UBI-vol-id> [-s <size>] <img-file>*
  *Update volume contents.*
    - ***Example:** ubiupdatevol /dev/ubi0_1 rootfs.img*
- *ubirsvol /dev/ubi<UBI-device-id> -N <name> -s <size>*
  *Resize a UBI volume.*
    - ***Example:** ubirsvol /dev/ubi0 -N varlog -s 32MiB*
- *ubirename /dev/ubi<UBI-device-id> <old-name> <new-name>*
  *Rename a UBI volume.*
    - ***Example:** ubirename /dev/ubi0 varlog var*

*#Linux: UBIFS host tools:*

*-UBIFS filesystems images can be created using **mkfs.ubifs***

- *mkfs.ubifs -m 4096 -e 258048 -c 1000 -r rootfs/ ubifs.img*
    - ***-m 4096:** minimal I/O size (see /sys/class/mtd/mtdx/writesize).*
    - ***-e 258048:** logical erase block size (smaller than PEB size, can be found in the kernel log after running ubiattach)*
    - ***-c 1000:** maximum size of the UBI volume the image will be flashed into, in number of logical erase blocks.*

*⇒ Do not make this number unnecessary big, otherwise the UBIFS data structures will be bigger than needed and performance will be degraded.*

*-Once created*

- *Can be written to a UBI volume from the target either using ubiupdatevol in Linux, or ubi writevol in U-Boot.*
- *Or, can be included in a UBI image (using ubinize on the host)*

*#Linux: UBIFS target tools*

*- No specific tools are required to create a UBIFS filesystem. An empty filesystem is created the first time it is mounted. The same applies to JFFS2.*

*- Mounting a UBIFS filesystem is done with mount:* ***mount -t ubifs <ubi-device-id>:<volume-name> <mount-point>***

- ***Example:*** *mount -t ubifs ubi0:data /data*

*#Linux: Using a UBIFS filesystem as root filesystem*

*-You just have to pass the following information on the kernel command line:*

- *ubi.mtd=UBI*

  *Attach the MTD partition named UBI to the UBI layer and create ubi0.*

  *Note: you can also use the MTD partition number (more error prone): ubi.mtd=1*
- *rootfstype=ubifs root=ubi0:rootfs*

  *Mount the rootfs volume on ubi0 as a UBIFS filesystem*

  - ***rootfstype=*** *lets the kernel know what filesystem to mount as root filesystem. It's mandatory for UBIFS, but it can also be used for block filesystems. This way the kernel doesn't have to try all the filesystems it supports. This reduces boot time.*
  - ***Example:*** *rootfstype=ubifs ubi.mtd=UBI root=ubi0:rootfs*

*# Summary: how to boot on a UBIFS filesystem*

*In U-Boot:*

- *Define partitions:*

  *setenv mtdids ...*

  *setenv mtdparts ...*
- *Define the base Linux kernel bootargs, specifying booting on UBIFS, the UBI volume used as root filesystem, and the MTD partition attached to UBI.*
- *Example: setenv bootargs_base console=ttyS0 rootfstype=ubifs root=ubi0:rootfs ubi.mtd=UBI*
- *Define the boot command sequence, loading the U-Boot partition definitions, loading kernel and DTB images from UBI partitions, and adding mtdparts to the kernel command line.*
- *Example:*

  *setenv bootcmd 'ubi part UBI; ubi readvol 0x81000000 kernel; ubi readvol 0x82000000 dtb; setenv bootargs ${bootargs_base} ${mtdparts}; bootz 0x81000000 - 0x82000000'*

*#Linux: Block emulation layers:*

*- Linux provides two block emulation layers:*

- *mtdblock: block devices emulated on top of MTD devices*
- *ubiblock: block devices emulated on top of UBI volumes*

***Linux: mtdblock***

*- The mtdblock layer creates a block device for each MTD device of the system. Usually named /dev/mtdblockX.*

*- Allows read/write block-level access. However bad blocks are not handled, and no wear leveling is done for writes.*

*- For historical reasons, JFFS2 and YAFFS2 filesystems require a block device for the mount command.*

*- Do not write on mtdblock devices*

***Linux: ubiblock***

*- CONFIG_MTD_UBI_BLOCK*

*- The ubiblock layer creates read-only block devices on demand. The user specifies which static volumes (s)he would like to attach to ubiblock.*

- *Through the kernel command line: by passing ubi.block=<ubi-dev-id>,<volume-name>*
  - ***Example:*** *ubi.block=0,rootfs*
- *In Linux, using the ubiblock utility provided by mtd-utils: ubiblock --create <ubi-volume-dev-file>*

*- Usually named* ***/dev/ubiblockX_Y****: where X is the UBI device id and Y is the UBI volume id (example: /dev/ubiblock0_3)*

***Real-time in embedded Linux systems***

*Linux and real-time approaches:*

*-Approach 1*

- *Improve the Linux kernel itself so that it matches real-time requirements, by providing bounded latencies, real-time APIs, etc.*
- *Approach taken by the mainline Linux kernel and the* ***PREEMPT_RT*** *project.*

*-Approach 2*

- *Add a layer below the Linux kernel that will handle all the real-time requirements, sothat the behavior of Linux doesn't affect real-time tasks.*
- *Approach taken by RTLinux, RTAI and Xenomai*

***Improving the mainline Linux kernel with PREEMPT_RT***

*Understanding latency:*

- *1. An event from the physical world happens and gets notified to the CPU by means of an interrupt.*
- *2. The interrupt handler recognizes and handles the event, and then wake-up the user space task that will react to this event.*
- *3. Some time later, the user space task will run and be able to react to the physical world event.*

*Sources of interrupt latency*

*There can be multiple causes delaying the execution of an interrupt handler.*

- *Interrupts disabled by evil drivers. Such drivers may still exist but can be fixed.*
- *Already running interrupt handlers. In Linux, there are no nested interrupts. Only once interrupt handler can run at the same time on a give CPU core. Therefore, your critical interrupt handler may wait for the completon of another, non-critical handler.*
- *Sections of kernel code when disabling interrupts is necessary, typically in some spinlocks.*

*Disabling interrupts in spinlocks*

- *One of the concurrency prevention mechanism used in the kernel is the spinlock*
- *It has several variants, but one commonly used is to prevent concurrent accesses between a process context and an interrupt context works by disabling interrupts*
- *We want to avoid situations in which the interrupt handler waits forever for a spinlock that was held by the code it interrupted*

⇒ *To avoid such issues, when there can be concurrency between regular kernel code and interrupt code, spinlocks are held with interrupts disabled. This can delay the execution of a critical interrupt handler.*

*-The duration of these critical sections in which spinlocks are held with interrupts disabled is unbounded.*

*Interrupt handler implementation*

*-In Linux, many interrupt handlers are split in two parts*

- ***A top-half:** started by the CPU as soon as interrupts are enabled. It runs with the interrupt line disabled and is supposed to complete as quickly as possible.*
- ***A bottom-half:** scheduled by the top-half, which starts after all pending top-halves have completed their execution.*

⇒ *Therefore, for real-time critical interrupts, bottom-halves shouldn't be used: their execution is delayed by all other interrupts in the system.*

*Understanding preemption:*

*-The Linux kernel is a preemptive operating system:*

- *When a task runs in user space mode and gets interrupted by an interruption, if the interrupt handler wakes up another task, this task can be scheduled as soon as we return from the interrupt handler.*
- *However, when the interrupt comes while the task is executing a system call, this system call has to finish before another task can be scheduled.*

*- By default, the Linux kernel does not do kernel preemption.*

- *This means that the time before which the scheduler will be called to schedule another task is unbounded.*

*Other non-deterministic mechanisms*

*The non-deterministic mechanisms of Linux can affect the execution time of real-time tasks:*

- *Linux is highly based on virtual memory, as provided by an MMU, so that memory is allocated on demand. Whenever an application accesses code or data for the first time, it is loaded on demand, which can creates huge delays.*

⇒ *To avoid such sources of non-determinism, your system should allocate all the resources it needs ahead of time, before it is ready to react to events in a real-time way. For the virtual memory needs, it will be done through the mlock() and mlockall() system calls.*

*Priority inversion*

*- A process with a low priority might hold a lock needed by a higher priority process, effectively reducing the priority of this process. Things can be even worse if a middle priority process uses the CPU.*

*Interrupt handler priority*

*- In Linux, interrupt handlers are executed directly by the CPU interrupt mechanisms, and not under control of the Linux scheduler. Therefore, all interrupt handlers have a higher priority than all tasks running on the system.*

***The PREEMPT_RT project:***

*1st option: no forced preemption: CONFIG_PREEMPT_NONE*

*⇒ Kernel code (interrupts, exceptions, system calls) never preempted. Default behavior in standard kernels.*

*2nd option: voluntary kernel preemption: CONFIG_PREEMPT_VOLUNTARY*

*⇒ Kernel code can preempt itself, Adds explicit rescheduling points (might_sleep()) throughout kernel code.*

*3rd option: preemptible kernel: CONFIG_PREEMPT*

*⇒ Most kernel code can be involuntarily preempted at any time. When a process becomes runnable, no more need to wait for kernel code (typically a system call) to return before running the scheduler. Exception: kernel critical sections (holding spinlocks):*

## ***Priority inheritance***

*-One classical solution to the priority inversion problem is called priority inheritance.*

- *In the Linux kernel, mutexes support priority inheritance*
- *In user space, priority inheritance must be explicitly enabled on a per-mutex basis.*

## ***High resolution timers***

*-The high-resolution timers infrastructure allows to use the available hardware timers to program interrupts at the right moment.*

- *Hardware timers are multiplexed, so that a single hardware timer is sufficient to handle a large number of software-programmed timers.*
- *Usable directly from user space using the usual timer APIs*

## ***Threaded interrupts***

***⇒ To solve the interrupt inversion problem, PREEMPT_RT has introduced the concept of threaded interrupts.***

*- The interrupt handlers run in normal kernel threads, so that the priorities of the different interrupt handlers can be configured.*

*- The real interrupt handler, as executed by the CPU, is only in charge of masking the interrupt and waking-up the corresponding thread.*

*- The idea of threaded interrupts also allows to use sleeping spinlocks.*

*- The conversion of interrupt handlers to threaded interrupts is not automatic: drivers must be modified.*

*- In PREEMPT_RT, all interrupt handlers are switched to threaded interrupts.*

## ***CONFIG_PREEMPT_RT***

*- This level of preemption replaces all kernel spinlocks by mutexes (or so-called sleeping spinlocks)*

*- Instead of providing mutual exclusion by disabling interrupts and preemption, they are just normal locks: when contention happens, the process is blocked and another one is selected by the scheduler.*

*- Works well with threaded interrupts, since threads can block, while usual interrupt handlers could not.*

*- Some core, carefully controlled, kernel spinlocks remain as normal spinlocks*

*- With CONFIG_PREEMPT_RT, virtually all kernel code becomes preemptible*

*- An interrupt can occur at any time, when returning from the interrupt handler, the woken up process can start immediately.*

## ***PREEMPT_RT setup***

*- PREEMPT_RT is delivered as a patch against the mainline kernel*

*-In the kernel configuration, be sure to enable*

- *CONFIG_PREEMPT_RT*
- *High-resolution timers*

*- Compile your kernel, and boot*

*- You are now running the real-time Linux kernel*

*- Of course, some system configuration remains to be done, in particular setting appropriate priorities to the interrupt threads, which depend on your application.*

## ***Real-time application development with PREEMPT_RT***

### *Development and compilation*

*- No special library is needed, the POSIX real-time API is part of the standard C library.*

*- The glibc C library is recommended, as support for some real-time features is not mature in other C libraries.*

*- Compile a program*

- *ARCH-linux-gcc -o myprog myprog.c -lrt*

### *Process, thread: kernel point of view*

*- The kernel represents each thread running in the system by a struct task_struct structure.*

*- From a scheduling point of view, it makes no difference between the initial thread of a process and all additional threads created dynamically using pthread_create().*

***Scheduling classes***

*-The Linux kernel scheduler support different scheduling classes*

- *The default class (SCHED_OTHER), in which processes are started by default, is a time-sharing class*

*-The real-time classes SCHED_FIFO, SCHED_RR and SCHED_DEADLINE*

- *RT Priorities ranging from 0 (lowest) to 99 (highest)*
- *The highest RT priority process gets all the CPU time, until it blocks.*
- *With SCHED_FIFO, First In, First Out, each additional process with the same RT priority has to wait for the first ones to block.*
- *In SCHED_RR, Round-Robin scheduling between the processes with the same RT priority. All must block before lower priority processes get CPU time.*
- *In SCHED_DEADLINE: guarantees that an RT task will be given an exact amount of cpu time every period.*

***Using scheduling classes***

*- An existing program can be started in a specific scheduling class with a specific priority using the chrt command line:*

- *Example: chrt -f 99 ./myprog*

  *-f: SCHED_FIFO*
  *-r: SCHED_RR*
  *-d: SCHED_DEADLINE*

*- The sched_setscheduler() API can be used to change the scheduling class and priority of a process:*

*int sched_setscheduler(pid_t pid, int policy, const struct sched_param *param);*

- *policy can be SCHED_OTHER, SCHED_FIFO, SCHED_RR, SCHED_DEADLINE, etc. (others exist).*
- *param is a structure containing the priority*

*- The priority can be set on a per-thread basis when a thread is created*

```
struct sched_param parm;
pthread_attr_t attr;

pthread_attr_init(&attr);
pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
parm.sched_priority = 42;
pthread_attr_setschedparam(&attr, &parm);
```

*⇒ Then the thread can be created using pthread_create(), passing the attr structure.*

***Memory locking***

*- In order to solve the non-determinism introduced by virtual memory, memory can be locked.*

*- mlockall(MCL_CURRENT | MCL_FUTURE);*

- *Locks all the memory of the current address space, for currently mapped pages and pages mapped in the future*

*- Other, less useful parts of the API: munlockall, mlock, munlock.*

***Mutexes***

*- Allows mutual exclusion between two threads in the same address space*

- *Initialization/destruction*

  ```
  pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);
  pthread_mutex_destroy(pthread_mutex_t *mutex);
  ```

- *Lock/unlock*

  ```
  pthread_mutex_lock(pthread_mutex_t *mutex);
  pthread_mutex_unlock(pthread_mutex_t *mutex);
  ```

- *Priority inheritance must be activated explicitly*

  ```
  pthread_mutexattr_t attr;
  pthread_mutexattr_init (&attr);
  pthread_mutexattr_setprotocol(&attr, PTHREAD_PRIO_INHERIT);
  ```

***Timer:***

*Timer creation*

*timer_create(clockid_t clockid, struct sigevent *evp, timer_t *timerid);*

- *clockid is usually CLOCK_MONOTONIC.*
- *sigevent defines what happens upon timer expiration: send a signal or start a function in a new thread.*
- *timerid is the returned timer identifier.*

*Configure the timer for expiration at a given time*

*timer_settime(timer_t timerid, int flags, struct itimerspec *newvalue, struct itimerspec *oldvalue);*

*Delete a timer*

*timer_delete(timer_t timerid);*

***Signal:***

*- Signals are asynchronous notification mechanisms, Notification occurs either:*

- *By the call of a signal handler. Be careful with the limitations of signal handlers!*
- *By being unblocked from the sigwait(), sigtimedwait() or sigwaitinfo() functions. Usually better.*

*- Signal behavior can be configured using sigaction()*

*- The mask of blocked signals can be changed with pthread_sigmask()*

*- Delivery of a signal using pthread_kill() or tgkill()*

*- All signals between SIGRTMIN and SIGRTMAX, 32 signals under Linux.*

***Inter-process communication***

*Semaphores*

- ▸ *Usable between different processes using named semaphores*
- ▸ *sem_open(), sem_close(), sem_unlink(), sem_init(), sem_destroy(), sem_wait(), sem_post(), etc.*

*Message queues*

- ▸ *Allows processes to exchange data in the form of messages.*
- ▸ *mq_open(), mq_close(), mq_unlink(), mq_send(), mq_receive(), etc.*

*Shared memory*

- ▸ *Allows processes to communicate by sharing a segment of memory*
- ▸ *shm_open(), ftruncate(), mmap(), munmap(), close(), shm_unlink()*

***Debugging latencies in PREEMPT_RT***

*ftrace - Kernel function tracer:*

- *Infrastructure that can be used for debugging or analyzing latencies and performance issues in the kernel.*

*- Tracing information available through the tracefs virtual fs. Mount this filesystem as follows:*

- *mount -t tracefs nodev /sys/kernel/tracing*

*- Check available tracers in /sys/kernel/tracing/available_tracers.*