TP1

Introduction:

Objectives
The objective of this first tutorial is to learn the basic elements to start a minimalist Linux system:

Linux kernel
init process
initramfs memory image using BusyBox
U-Boot bootloader

Prerequisites: All must do this part before starting the TP1
Convention: In the following text, when you have commands to type in a terminal, they are indicated with a $ character at the beginning of the line. This $ character at the beginning of the command lines represents the command prompt, i.e. the characters displayed by your command interpreter to invite you to type a command (they vary from one command interpreter to another and according to your configuration). This $ character (only at the beginning of the line) is therefore not to be typed.

Preparation of the TP environment

We recommend that you work in a new directory for the rest of the tutorial. In the following text (and in the following exercises) we will refer to it with the environment variable TPROOT.

If you are on your own machine, you can create the directory wherever you want and give it the name you want. For example, to create a directory seti-b4-tp in your home directory, you can type :

```
$ export TPROOT=~/seti-b4-tp
$ mkdir $TPROOT
$ cd $TPROOT
```

Be careful, if you open another terminal, you will need to redefine the TPROOT variable (using the first line).

If you are on a school's lab machine, be aware that the size of your home directory is very limited (a few gigabytes). But we will need more space. Fortunately, the machines have a local directory that you can use to store large files. However, be aware that the files in this directory are not saved and can only be accessed from the machine you are using (unlike your home directory which can be accessed from any TP machine). Use the following commands to create a local working directory (replace xxx with your user name):

```
$ export TPROOT=/home/users/xxx/seti-b4-tp
$ mkdir $TPROOT
```

```
$ cd $TPROOT
```

If the mkdir command fails, do not go any further.

Be careful, if you open another terminal, you will need to redefine the TPROOT variable (using the first line).

QEMU

To simulate a complete ARM-based embedded system, we will use QEMU. In addition to not needing physical hardware to experiment, Qemu also allows to easily test and debug your embedded system.

More precisely, we are going to use qemu-system-arm which is a full machine emulator, in this case, an Arm Versatile Express development board with a CoreTile Express A9x4 Cortex-A9 based expansion board.

The details of what QEMU simulates on this board are available in the documentation: https://www.qemu.org/docs/master/system/arm/vexpress.html.

We will use a particular version of qemu-system-arm (it adds an emulated device to QEMU for which you will write a device driver in the rest of the course).

Download :

For a Debian Buster distribution (especially the machines in the classrooms): qemu-system-arm (~90 MB)
For a Debian Bullseye distribution: qemu-system-arm (~90 MB)
For Ubuntu 20.04 distribution : qemu-system-arm (~90 Mb)
For Ubuntu 22.04 distribution : qemu-system-arm (~90 Mo)
Versions available here: https://perso.telecom-paristech.fr/duc/cours/linux/tp1.html

Put the downloaded executable in your $TPROOT directory, then check that the rights are correct for execution:

Place the downloaded executable in your $TPROOT directory, then check that the rights are correct for execution:

```
$ chmod u+x qemu-system-arm
```

Normally, this executable uses shared libraries that should be available on your machine. Nevertheless, check that all libraries are present by typing :

```
$ ldd qemu-system-arm
```

If you wish, the source code of QEMU and the changes made are available in the adxl345 branch of https://gitlab.telecom-paris.fr/guillaume.duc/qemu.

Cross-compiler

All the software layers we are going to compile (kernel, tools, etc.) are intended to be run on an ARM Cortex-A based machine. However, it is very likely that you will run the project on an x86 or x86_64 processor based machine.

So if you compile using the classical compiler of your machine, you will get machine code for x86 or x86_64 processor which will not be able to run on an ARM processor.

You will therefore have to use a cross-compiler, i.e. a compiler that runs on an A architecture (here x86_64) but produces code for a B architecture (here arm).

We will see how to build a cross-compiler in a later course, for now we will use the arm-linux-gnueabihf chain provided by Linaro.

If you are on your personal machine:

Download it here: https://releases.linaro.org/components/toolchain/binaries/7.4-2019.02/arm-linux-gnueabihf/gcc-linaro-7.4.1-2019.02-x86_64_arm-linux-gnueabihf.tar.xz

Then unzip it somewhere (for example in your lab directory) and add the path `/.../gcc-linaro-7.4.1-2019.02-x86_64_arm-linux-gnueabihf/bin` (replacing /.../ with the correct path) in your PATH environment variable.

If you are on a machine in the classrooms:

Since the chain is already installed, simply add the directory `/comelec/softs/opt/gnu_tools_for_arm/gcc-linaro-7.4.1-2019.02-x86_64_arm-linux-gnueabihf/bin/` to your PATH environment variable:

```
$ export PATH=$PATH:/comelec/softs/opt/gnu_tools_for_arm/gcc-linaro-7.4.1-2019.02-x86_64_arm-linux-gnueabihf/bin/
```

Classic tools

If you are on your own machine, to do this and the following tasks, you will need a number of classical development tools, some of which are probably already installed on your distribution.

For Debian/Ubuntu, install the following packages:

build-essential
libncurses-dev and libncurses5-dev
file

cpio
unzip
rsync
bc

The Linux kernel
The most important component of a Linux-based system is... the Linux kernel itself.

Official sources
The official versions of the Linux kernel can be downloaded from https://www.kernel.org.
The home page allows quick access to :

The current stable version (stable)
The version under development (mainline)
The older versions still maintained (longterm)
The sources can be retrieved as an archive containing a given version (.tar.xz format). Once unpacked, the sources of a version take a little more than 1 Gio.

It is also possible (and even recommended if you want to develop), to get directly the official git repository including all the history (since the kernel developers use git!):

Note: do not type the commands below, they are only given as an indication for the moment.

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

or (for example if a paranoid firewall blocks access to the git protocol (TCP port 9418)) :

```
$ git clone https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

In December 2021, a copy of the official git repository takes about 3.8 Gio.

Alternative sources
Many manufacturers maintain their own versions of the kernel sources including support for their products when it is not yet integrated into the official versions.

For example, the Linux kernel sources for Altera's SoC FPGAs are available here: https://github.com/altera-opensource/linux-socfpga.

Also, some distributions maintain divergent kernel versions for various reasons.

Kernel configuration
Preliminary note: the commands in this section are given as examples and do not have to be executed.

The Linux kernel is extremely versatile (it is designed to run on many architectures, from small embedded systems to large supercomputers, it handles many different devices, many network protocols, etc.). It therefore has a configuration system that allows the kernel to be adapted to the user's needs by selecting only what is necessary.

Configuration management tools: For your knowledge

Several targets of the main kernel Makefile allow to create and modify this configuration:

make config : basic textual tool, all options are presented one after the other (very inconvenient)
make menuconfig : semi-graphical tool (requires the ncurses library and its headers)
make nconfig : semi-graphical tool (requires the ncurses library and its headers)
make xconfig : graphical tool based on Qt (requires Qt development libraries)
make gconfig : graphical tool based on GTK+ (requires GTK+ 2.0 development libraries)
If no configuration file exists (which is the case when retrieving the official sources), these tools will look for a file with a name of the form /boot/config-* and take it as a base. If not, the default values of the parameters are chosen.

If you already have a configuration file, for example from a previous kernel version, the make oldconfig command can be very useful. It removes options that no longer exist and asks for user input only for new options.

Finally, a number of default configuration files for many embedded boards are available in the arch/xxx/configs directory (where xxx represents an architecture, like arm). To use them, just type make yyy_defconfig.

Syntax and meaning of the configuration

The configuration is stored, in a simple textual format (key=value), in the .config file.

Several types are possible for a value :

bool: true (y) / false (not declared: # XXX is not set)
tristate : true (y) / modulus (m) / false (not declared : # XXX is not set)
string : string of characters
hex : numerical value represented in hexadecimal form (base 16)
int : numerical value represented in decimal form (base 10)
Some options may depend on other options (example : the support of USB storage devices depends on the support of USB).

For bool or tristate types, the value has the following meaning:

true : the functionality is hard-coded into the kernel
module : the functionality is compiled as a module (an independent file that can be loaded dynamically at runtime)
false : the functionality is not integrated in the kernel

Here is an extract from a kernel configuration file:

```
#
# USB Host Controller Drivers
#
# CONFIG_USB_C67X00_HCD is not set
CONFIG_USB_XHCI_HCD=m
# CONFIG_USB_XHCI_PLATFORM is not set
CONFIG_USB_EHCI_HCD=m
CONFIG_USB_EHCI_ROOT_HUB_TT=y
CONFIG_USB_EHCI_TT_NEWSCHED=y
CONFIG_USB_EHCI_PCI=m
```

Compiling the kernel (For your Knowledge only)
Preliminary note: the commands in this section are given as examples and do not have to be executed.

Native compilation

To compile the kernel for an architecture identical to the one the compiler is running on, you just need to run the make command at the kernel root.

It is possible to have all the files produced during the compilation stored in another directory (for example if the sources are read-only for the user or if one does not want to pollute them):

```
$ make O=/home/toto/build menuconfig
$ make O=/home/toto/build
```

Several files are produced by the compilation, including :

The final image, most often compressed, used to boot the kernel: arch/<arch>/boot/*Image
The uncompressed raw image, in ELF format, vmlinux. It is used in particular for debugging
The compiled modules (*.ko files) distributed in the whole kernel tree

Installation of the kernel and modules

The installation of the kernel is done with the install target of the Makefile :

```
# make install
```

or, if you compiled the kernel using a separate :

```
# make O=/home/toto/build install
```

This command performs the following operations:

Copy the compressed image to /boot/vmlinuz-<version>.
Copy the configuration file to /boot/config-<version>.
Copy the kernel symbol table to /boot/System.map-<version>.

As write access to the /boot directory is normally not possible for a normal user, the make command needs administrator privileges to be able to perform the installation.

The modules are installed using the modules_install target of the Makefile :

```
# make modules_install
```

It installs modules (*.ko files as well as module dependency, symbol table and alias information) in /lib/modules/<version>.

Cross-compilation

Sometimes it is necessary to compile the kernel for a different architecture than the one on which the compilation is done. This is called cross-compilation.

Two variables are used by the Makefile to allow cross-compilation:

ARCH: name of the target architecture (as it appears in arch/)
CROSS_COMPILE : prefix of the compilation chain (e.g. arm-linux-gnueabihf- if the compiler is called arm-linux-gnueabihf-gcc)
These two variables must be passed at each invocation of a Makefile target, either by specifying them on the command line or by exporting them as environment variables.

```
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- menuconfig
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
```

Or this:

```
$ export ARCH=arm
$ export CROSS_COMPILE=arm-linux-gnueabihf-
$ make menuconfig
$ make
```

1- Source cleaning

Three Makefile targets are available to allow cleaning of sources:

clean: Deletes most of the generated files, except for the configuration (.config) and the files needed to compile external modules
mrproper : Delete all generated files, including .config (be careful)
distclean : Like mrproper but deletes all the backup files of the editors (*~ for example), the patch reject files, etc.

Work to do (Practical Work Starts Here):

Get the latest stable kernel version (5.15.6 at the time of writing).

```
$ cd $TPROOT
$ wget https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.15.6.tar.xz
```

2- Uncompress it in your TP directory:

```
$ tar xJf linux-5.15.6.tar.xz
$ cd linux-5.15.6
```

3. Define the environment variable for kernel cross-compiling

```
$ export ARCH=arm
```

```
$ export CROSS_COMPILE=arm-linux-gnueabihf-
```

4. Run the configuration from the default configuration file for the Versatile Express card (we will use a separate build directory for compilation):

```
$ make O=build vexpress_defconfig
$ make O=build menuconfig
```

5. Quickly go through the configuration. In the following we will need an option that is not enabled in the default configuration: Support for uevent helper (in Device Drivers -> Generic Driver Options).

6. Compile the Kernel :

```
$ make O=build
```

Two important files are produced by the compilation:

- `build/arch/arm/boot/zImage` : the compressed image of the core

7. `build/arch/arm/boot/dts/vexpress-v2p-ca9.dtb` : the default device tree for the map
8.
9. Launch QEMU :

```
$ cd $TPROOT
$ ./qemu-system-arm -machine vexpress-a9 -nographic -kernel \
linux-5.15.6/build/arch/arm/boot/zImage -dtb \
linux-5.15.6/build/arch/arm/boot/dts/vexpress-v2p-ca9.dtb
```

This command starts QEMU by telling it the type of machine to emulate (vexpress-a9), the kernel image to use (linux-5.15.6/build/arch/arm/boot/zImage) and the tree of devices to use (linux-5.15.6/build/arch/arm/boot/dts/vexpress-v2p-ca9.dtb) In this mode, QEMU will place the kernel image and the DTB in RAM and start the kernel execution, as a bootloader would (see the course on booting a Linux system).

You should see the kernel boot messages appear (they appear because QEMU redirects to the terminal the first serial link of the emulated board, which happens to be the one used by the kernel to send its messages). The kernel will panic and stop. Why?

When QEMU is running, everything you type in your terminal is sent to your virtual machine through the emulated serial link. You can access the QEMU console by typing ctrl-a c. Then in this console, type quit (or q) to stop QEMU.

A first file system and a first init process
The kernel does not find a file system. So we will first provide it with an initramfs memory image and a first small init process.

```
$ cd $TPROOT
$ mkdir initramfs_simple
$ cd initramfs_simple
```

1. Create the file init.c with the following content (you are free to adapt if you want)

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
  printf("Hello world!\n");
  sleep(10);
}
```

2. Compile :

```
$ arm-linux-gnueabihf-gcc -static init.c -o init
```

Note: here we use -static to link the C library with the executable so that the latter contains everything it needs to run (otherwise we would have to copy the standard library, the dynamic linker and the dynamic loader into the archive).

3. Then create an initramfs image using cpio (take the opportunity to look at the cpio man page to understand how the following command line works):
   `$ echo init | cpio -o -H newc | gzip > test.cpio.gz`

4. Launch the simulation:

```
$ cd ..
$ ./qemu-system-arm -machine vexpress-a9 -nographic -kernel \
linux-5.15.6/build/arch/arm/boot/zImage \
-dtb linux-5.15.6/build/arch/arm/boot/dts/vexpress-v2p-ca9.dtb \
-initrd initramfs_simple/test.cpio.gz
```

You should see that your program is executed and then, when it ends (after a few seconds), the kernel panics (init is not supposed to end).

A more ambitious memory image with BusyBox

BusyBox

BusyBox describes itself as the Swiss Army knife for embedded Linux systems. It combines simplified versions of many common Unix utilities (init, mount, cp, ls, ifconfig...) into one small executable. So it provides, by itself, an almost complete environment for small embedded systems.

So we are going to use BusyBox to make an initial memory image a little more functional than the previous one.

1. Start by getting the latest version of BusyBox (1.34.1 at the time of writing this tutorial):`$ cd $TPROOT`

```
$ wget https://busybox.net/downloads/busybox-1.34.1.tar.bz2
```

2. Decompress :

```
$ tar xjf busybox-1.34.1.tar.bz2
```

3. configure *BusyBox* :

```
$ cd busybox-1.34.1
$ export ARCH=arm
$ export CROSS_COMPILE=arm-linux-gnueabihf-
$ make defconfig
```

```
$ make menuconfig
```

In the configuration, remember to activate the Build Static Binary option in the Settings menu (as before, we want to avoid having to manage dynamic libraries for the moment).

4. Then launch the compilation:

```
$ make
```

```
$ make install
```

Creation of the complete image

We are now going to prepare an initramfs image a little more complete integrating BusyBox.

1. Create a working directory

```
$ cd $TPROOT
```

```
$ mkdir initramfs_busybox
$ cd initramfs_busybox
```

2. Only if you are on the machines in the classrooms: we will have to create special files (with mknod), which is normally forbidden to a normal user. So we will do these operations in a virtual environment allowing us to do it only in order to create a correct archive:

```
$ fakeroot /bin/bash
```

3. Copy installation of *BusyBox*

```
$ cp -r ../busybox-1.34.1/_install/* .
```

You end up with bin, sbin, usr/bin and usr/sbin directories that contain symbolic links bearing the names of the classic Linux commands (cp, insmod, mount...) to the only real executable: bin/busybox. Busybox uses the name used to run the command (which, as a reminder, is the first argument retrieved from the command line) to identify what it should do. For example, if it is called via bin/cp, it will behave like the command cp.

4. There is a linuxrc symbolic link at the root of your image. As a reminder, it is this executable that the kernel will launch from an initrd image. Now we are building an initramfs image, so let's rename this symbolic link :`$ rm linuxrc`

```
$ ln -s bin/sh init
```

5. Then create the rest of the classic tree structure

```
$ mkdir proc sys tmp root var mnt dev etc
```

6. You must also create in dev some special files for the terminals (if you are on your personal machine, run these commands as root with su or sudo)

```
$ mknod dev/console c 5 1
$ mknod dev/tty1 c 4 1
$ mknod dev/tty2 c 4 2
$ mknod dev/tty3 c 4 3
$ mknod dev/tty4 c 4 4
```

7. It remains to put some configuration files in etc:

```
$ cd etc
$ cp ../../busybox-1.34.1/examples/inittab .
$ mkdir init.d
$ cd init.d
$ cat <<EOF > rcS
#!/bin/sh
mount -a
mkdir -p /dev/pts
mount -t devpts devpts /dev/pts
echo /sbin/mdev > /proc/sys/kernel/hotplug
mdev -s
mkdir -p /var/lock
ifconfig lo 127.0.0.1
EOF
$ chmod 755 rcS
$ cd ..
$ cat <<EOF > fstab
proc            /proc           proc    defaults                0       0
tmpfs           /tmp            tmpfs   defaults                0       0
sysfs           /sys            sysfs   defaults                0       0
tmpfs           /dev            tmpfs   defaults                0       0
EOF
```

The etc/inittab file is taken directly from the example provided by BusyBox. It tells it what it must execute at system startup and shutdown and which terminals to open.

The etc/init.d/rcS file is a script executed by BusyBox at startup. As filled in above, it mounts the file systems described in /etc/fstab, activates mdev which will statically and dynamically populate /dev and configures the loopback network interface.

Finally, the etc/fstab file lists the file systems to be mounted at boot time.

8. All that remains is to create the image with cpio$ cd ..

```
$ find . | cpio -o -H newc -R +0:+0 | gzip > initramfs.gz
```

Then, if you are in the lab, leave the fakeroot environment:

```
$ exit
```

9. You can then launch it with QEMU:

```
$ cd ..
$ ./qemu-system-arm -machine vexpress-a9 -nographic -kernel \
linux-5.15.6/build/arch/arm/boot/zImage \
-dtb linux-5.15.6/build/arch/arm/boot/dts/vexpress-v2p-ca9.dtb \
-initrd initramfs_busybox/initramfs.gz
```

Once Linux is started, it is enough to press a key to access a command interpreter.

Bootloader :

Once Linux is started, you just have to press a key to access a command interpreter.

The last thing we would need on a real system is a bootloader (as we have seen, QEMU knows how to boot the Linux kernel correctly and has therefore acted as a bootloader so far).

We will use Das U-Boot which is one of the most used bootloaders in the embedded world with Linux.

1. Download U-Boot :`$ cd $TPROOT`

```
$ wget https://ftp.denx.de/pub/u-boot/u-boot-2020.10.tar.bz2
```

2. Decompress archive

```
$ tar xjf u-boot-2020.10.tar.bz2
```

3. Configure and compile U-Boot (using the default configuration provided for the Versatile Express card)

```
$ cd u-boot-2020.10
$ make distclean
$ export ARCH=arm
$ export CROSS_COMPILE=arm-linux-gnueabihf-
$ make vexpress_ca9x4_defconfig
$ make
```

4. You can then start U-Boot with QEMU to test

```
$ cd ..
$ ./qemu-system-arm -machine vexpress-a9 -nographic -kernel u-boot-2020.10/u-boot
```

Remember to press a key during the U-Boot countdown. Otherwise, it will try to find a Linux image to load from the devices it supports (SD card, flash memory, network). Since none of these emulated devices contain the kernel, the boot process will fail.

All together

This section is not feasible on the machines in the classrooms, you can just read it.

In a real system, the bootloader (U-Boot) would most likely be stored on an SD card or in a flash memory of the system and would be put in memory and launched by a preloader (this part is very system dependent). In this tutorial, this preloader is simulated by QEMU which places the U-Boot image in memory and launches its execution.

Then, in a real system, U-Boot would fetch the Linux kernel image, the initial memory file system (initramfs orinitrd) and the device tree from a device (most often SD card or internal flash memory, but it can also be the network via TFTP for example) and place them in memory.

We will simulate this situation by generating an SD card image containing these elements and asking QEMU to use it so that U-Boot can load the elements it contains.

WARNING: Be very careful when using the following commands (parted, mkfs, etc., especially when preceded by sudo). You could end up overwriting your data if you are not careful about the arguments you pass to these commands.

1. Create a file sdcard that will contain the image of this SD card`$ cd $TPROOT`

```
$ mkdir sdcard
$ cd sdcard
$ dd if=/dev/zero of=sd bs=1M count=64
```

2. Partition this SD card (for the moment only one FAT partition covering the whole card)

```
$ parted sd
(parted) mklabel msdos
(parted) mkpart primary fat32 1MiB 100%
(parted) print
Modèle :  (file)
Disque : 67,1MB
Taille des secteurs (logiques/physiques) : 512B/512B
Table de partitions : msdos
Drapeaux de disque :

Numéro  Début   Fin     Taille  Type     Système de fichiers  Drapeaux
 1      1049kB  67,1MB  66,1MB  primary  fat32                lba

(parted) quit
```

3. With losetup make this partition appear as a device in /dev

```
$ sudo losetup -f --show -P sd
/dev/loop0
```

(if the command returns a number other than loop0, change the following lines)

4. Create the FAT file system on the partition`$ sudo mkfs.fat -F 32 /dev/loop0p1`

5. Mount this partition

```
$ mkdir mnt
$ sudo mount /dev/loop0p1 mnt
```

6. Copy the files containing the kernel, the DTB and the initramfs image

```
$ sudo cp ../linux-5.15.6/build/arch/arm/boot/zImage mnt
$ sudo cp ../linux-5.15.6/build/arch/arm/boot/dts/vexpress-v2p-ca9.dtb mnt
$ sudo cp ../initramfs_busybox/initramfs.gz mnt
```

7. The initial memory image must be encapsulated with a special header for U-Boot (the same could be done with the kernel image and the DTB but it is not necessary)

```
$ cd mnt
$ sudo ../../u-boot-2020.10/tools/mkimage -A arm -O linux -T ramdisk -d initramfs.gz
uinitramfs
```

8. Unmount the image of SD card

```
$ cd ..
$ sudo umount mnt
$ sudo losetup -d /dev/loop0
```

9. Start QEMU using the SD card and stop the countdown

```
$ cd $TPROOT
$ qemu-system-arm -machine vexpress-a9 -nographic -kernel \
u-boot-2020.10/u-boot -sd sdcard/sd
```

10. With U-Boot, load the files in memory

```
=> fatload mmc 0:1 0x62000000 zImage
=> fatload mmc 0:1 0x63000000 vexpress-v2p-ca9.dtb
=> fatload mmc 0:1 0x63100000 uinitramfs
```

11. Start the kernel

```
=> bootz 0x62000000 0x63100000 0x63000000
```

And that's it!

In a real system, these commands could be automated to be executed automatically. This automation can be done in a hard way during the compilation of U-Boot.