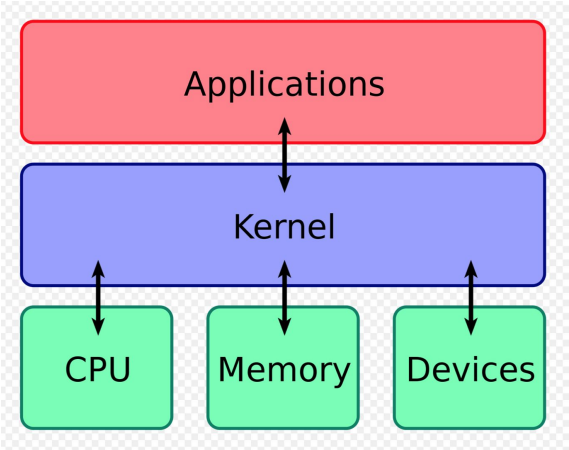
Embedded Linux Basics

Sara Sussman, 2/7/2020

These notes are based off of the 2/7/20 Doulos webinar "Anatomy of an Embedded Linux System". Most figures in these notes are taken from the slides in the webinar.



(Source: Wikipedia. "A **kernel** connects the application software to the hardware of a computer.")

1. **The Linux kernel**

In 1991 the **Linux kernel** was released by a Finnish man named Linus Torvalds. He wanted to make a kernel that was Unix-like. By the mid-90's, there was a large Linux developer community.

A typical **Linux distribution** (e.g. **Debian**) is composed of a Linux kernel and GNU tools and libraries. There are various Linux distros available. Some Linux distros are known as "commercial Linux"- these distros "make money out of free software" by offering paid support. One interesting example is the Red Hat company, which offers a community Linux distro (**Fedora**) that they use as a testbed for their commercial Linux distro (**Red Hat Enterprise**). IBM bought Red Hat for $34 billion USD in 2018- this is one of IBM's largest ever acquisitions.

Linux is often used on servers, but rarely used on desktop PCs. Increasingly, Linux is being used for embedded systems. 60% of embedded systems devices currently use Linux (cars, SoCs, cameras, etc.) Some embedded systems cannot use Linux because they require real-time performance and predictability- such systems use a **RTOS** "real time operating system" instead. One other difference between RTOSes and Linux is that RTOS has a low minimum memory footprint and can run on 100 kB of RAM, while Linux typically runs on 32 MB of RAM.

Embedded Linux distros differ in several ways from desktop Linux distros: the boot processes will be different, the kernel will have different drivers and features, and the filesystems will contain different things.

Linux pros and cons

Pros of using Linux:

- The bulk of Linux software is open source, which means a wide range of architectures is supported and a wide range of hardware is supported.

Cons of using Linux:

- The user needs to choose the kernel, the software and the libraries to put together their Linux system. This requires the user to have some background.
- Licensing conditions drive the use of open source software. If licensing is not managed properly in an open source project then that could become an issue for the developer (see example mentioned later).
- The real-time performance of Linux is suboptimal. Sometimes a system needs hard real-time performance and predictability and Linux might not be the best choice for that. Linux was not designed to be a RTOS. [Note: RTLinux is a thing]

Open source software pros

- It reduces software costs
- The source code is transparent to the user
- This transparency can improve the overall quality of the software
- It is easy to install and evaluate new open-source software on Linux
- "The open-source community provides a potentially huge 'virtual' development team for your project."
    - This community includes top industry people from Xilinx, ARM, Microsoft…

Open source licensing conditions (Linux)

Different licenses apply to different parts of Linux: the **GPL** (GNU general purpose license) applies to the Linux kernel and other main components of a Linux system. For instance, the GNU bootloader is under the GPL. The GPL is subject to a "copyleft" (instead of a copyright)- this means that users are entitled to access the source code of that device, and if the developer makes changes to the source code, they have to make these changes known to the user. The developer can satisfy that requirement by hosting their code on a website such as Github.

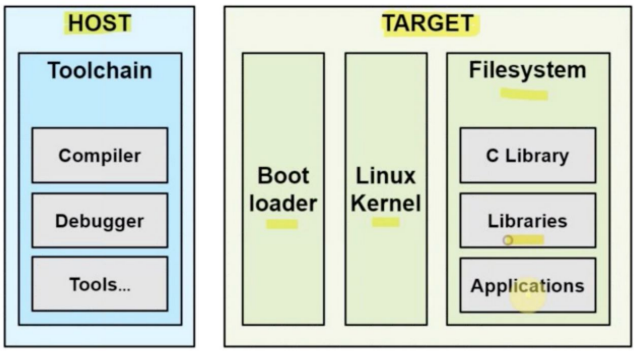
An extension to the GPL is the **LGPL** ("lesser GPL") which is used for many open source libraries that include additional features such as graphics. This license has a "weak copyleft", which means that the source code can link against (partially link) other libraries, without revealing the contents of those libraries to the user.

Examples of other open source licenses are MIT, Apache and BSD.

Some good practices for open-source licensing

- Keep a list of all the components of your system and your licenses.
- The goal for the developer is to ensure that no GPL/LGPL code exists in proprietary software, because then the developer could be forced to release the source code of that proprietary software.

2. The embedded Linux system

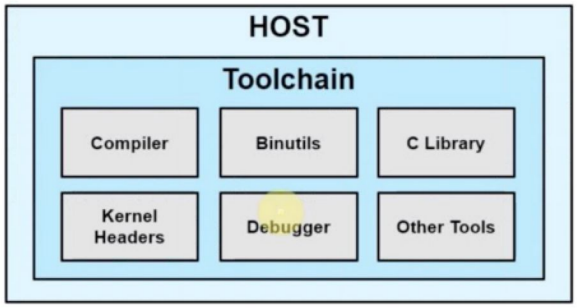


In an embedded Linux system, the host device architecture is used to generate the embedded system (target device) architecture. For example, the host device could be your PC and the target device could be a Xilinx SoC. The **toolchain** is the set of tools and software required for developing software on the target device architecture. The toolchain includes the blocks in the above diagram, like the compiler and the debugger. Target device users are generally not aware of the toolchain unless they encounter some bug in the target device architecture.

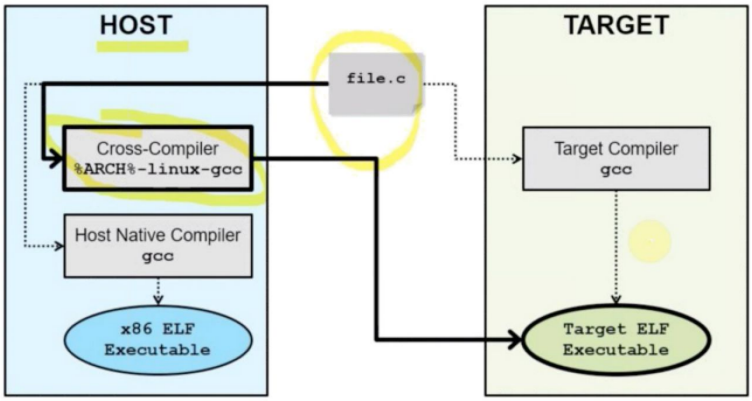
It is possible to emulate the embedded system hardware before deployment: **QEMU** is an emulator for different CPUs (e.g. x86, ARM, ARM64) which can run Linux applications.

[Note: beneath Linux you can add layers of hardware support for security such as TrustZone with a security kernel, virtual machine monitors, and hypervisors. Most SoC vendors provide support for this.]

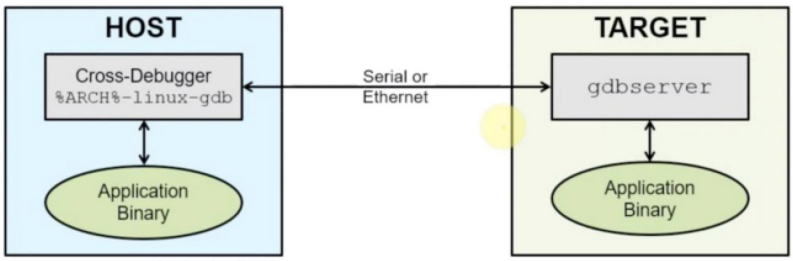
The GNU toolchain for embedded Linux



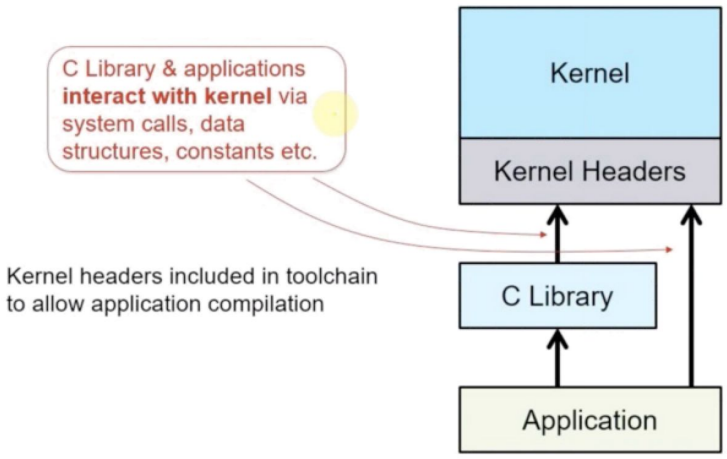
- You don’t create a toolchain by compiling it by hand. You could look for a pre-compiled toolchain (commercial or open source), or you could build the toolchain as part of a distribution management system.
- Here we will talk about the GNU toolchain for an embedded Linux system. One popular pre-compiled GNU toolchain is **Linaro**. Two popular GNU toolchain distribution management systems are **BuildRoot** and **yocto**- these will be introduced later. Here are some of the GNU toolchain components:
    - Compiler: in the GNU toolchain, the compiler is GCC.
        - Various versions of GCC exist for different system architectures. LLVM/CLANG are alternative compilers that are more application-specific than GCC.
    - Cross-compiler: There is also a separate GCC **cross-compiler** for the target device architecture. On the host machine, the GCC cross-compiler compiles all parts of the system which runs on the target device.



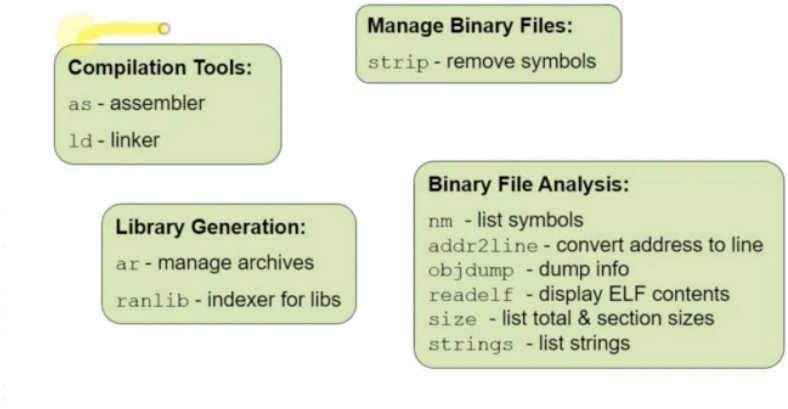
    - Cross-debugger: it is practical to **cross-debug** on the host machine.



    - Kernel headers: **kernel headers** allow programs like kernel modules and device drivers to build against the version of a kernel. You can generate the headers from the source code itself, too.

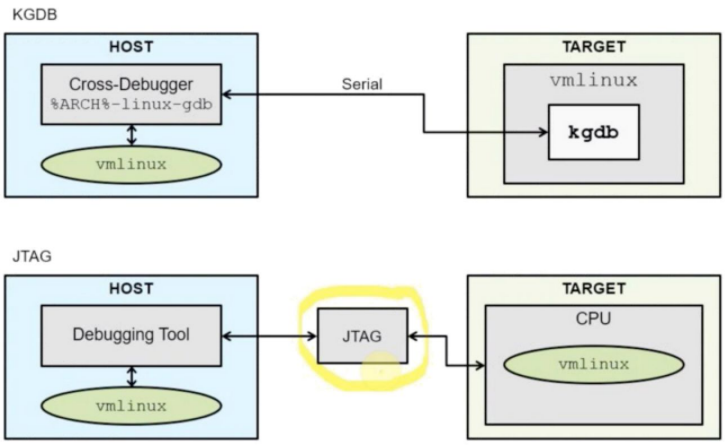


- Binutils: **binutils** is a collection of programs that allow the host device to manage the target device's binary files. binutils also contains "tracing and profiling" tools. These can be used for debugging or for analyzing an entire system or a process, such as which kernel functions are being called at any given moment, etc. Here are some examples of binutils programs.



- Putting a toolchain together by hand is hard.

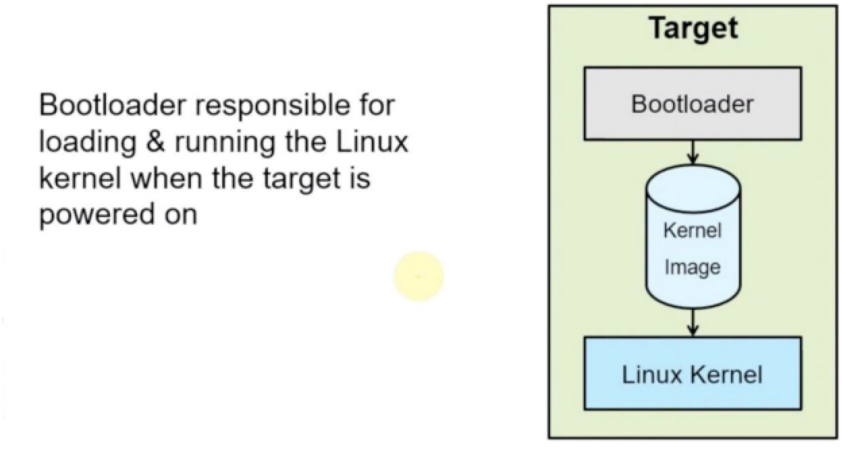
Debugging the embedded Linux system kernel with JTAG



a **JTAG controller** allows the host device to talk directly to the target device's CPU: it accesses on-chip debug modules inside the target CPU that allow the host device to (among other things) halt the CPU, inspect its registers and memory, and single step through code. A solution inside the target device's kernel is called kgdb but that might not perform as well as JTAG.

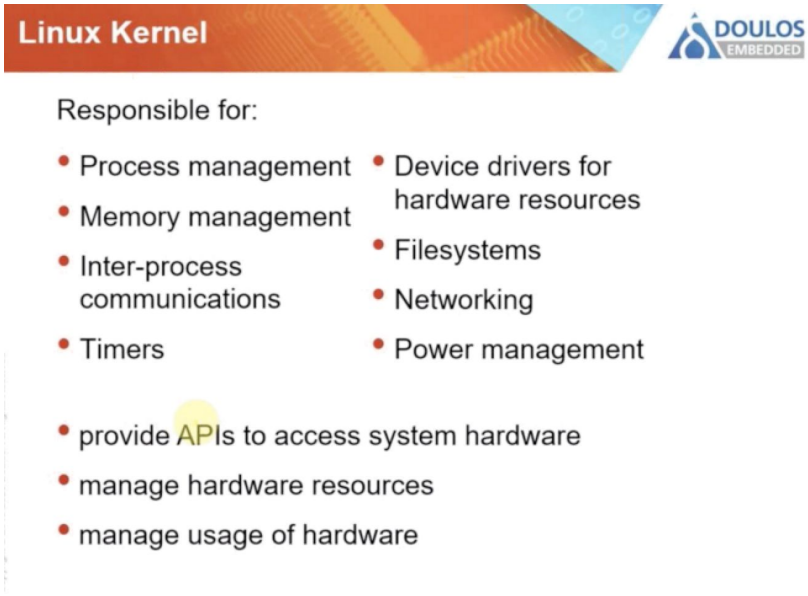
The embedded Linux system

- **Bootloader:** The **bootloader** loads the kernel image into system memory. A typical open-source bootloader is **U-Boot**, which is available for embedded system architectures such as ARM, RISC-V and MicroBlaze. The bootloader also provides drivers that allows you to talk to different types of storage and it has an interactive shell. So you can program in U-Boot to set up your boot process. The bootloader file is 100 kB so it can fit into a small system. Each SoC has its own methodology for loading the bootloader, often using some small 'pre-bootloader' on a ROM.



- **Linux kernel:** The kernel is the heart of a computer's OS- it more or less "is" the OS. The kernel is a set of APIs that allow a user to access the system hardware without knowledge of the hardware itself. E.g. if you open up a disk drive, you do not know

where that drive is, whether it is connected by USB, etc. The kernel also supports different power management modes including suspending to ram vs. suspending to disk. That is the job of the OS- to hide the system hardware knowledge and contain process management.




```
host$ ls kernel/
arch            firmware  lib             scripts
block           fs        MAINTAINERS     security
certs           include   Makefile        sound
COPYING         init      mm              tools
CREDITS         ipc       net             usr
crypto          Kbuild    README          virt
Documentation   Kconfig   REPORTING-BUGS
drivers         kernel    samples
```

The kernel is a big piece of software. Above is a picture of its top-level makefile. The arch directory is where the architecture-specific code is, it takes about 20% of the code base. The drivers directory is around 55% of the source base- the kernel supports many drivers. The fs directory holds the filesystem, and the mm directory handles memory management.

Here is an example of a user compiling their target device kernel on their host device. First, they set the environment variable ARCH which tells the kernel makefile which environment it is going to build for. They also set the environment variable CROSS_COMPILE, which tells the kernel makefile which version of GCC it is going to compile the kernel for. Next, the user configures the kernel for their specified target device (board). Finally, they compile the kernel with ‘make vmlinux’. Here vm = virtual memory.

Kernel makefile needs to know architecture & cross-compiler prefix:

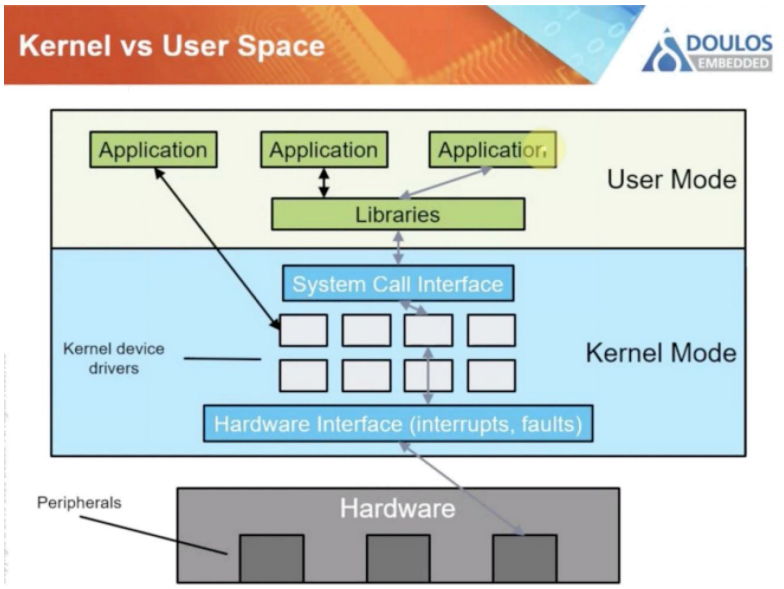
```
host$ echo $ARCH
arm
host$ echo $CROSS_COMPILE
arm-poky-linux-gnueabi-
```

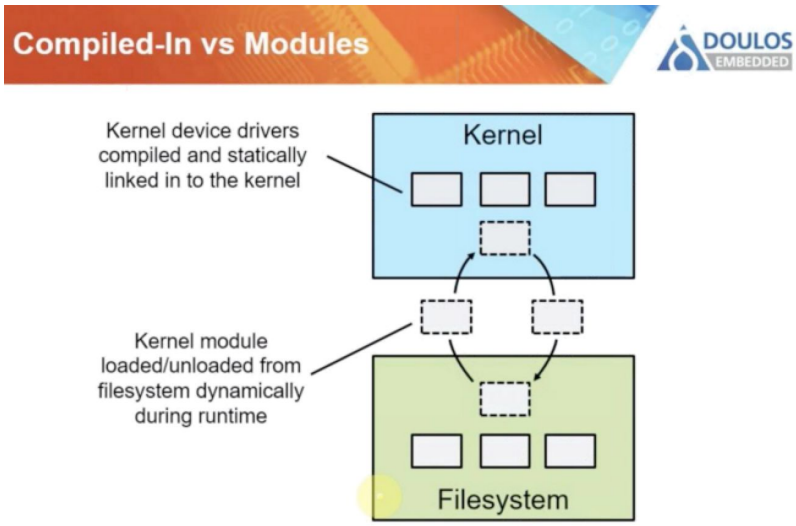
Configure and customize kernel:

```
host$ make <board>_defconfig
host$ make menuconfig
```

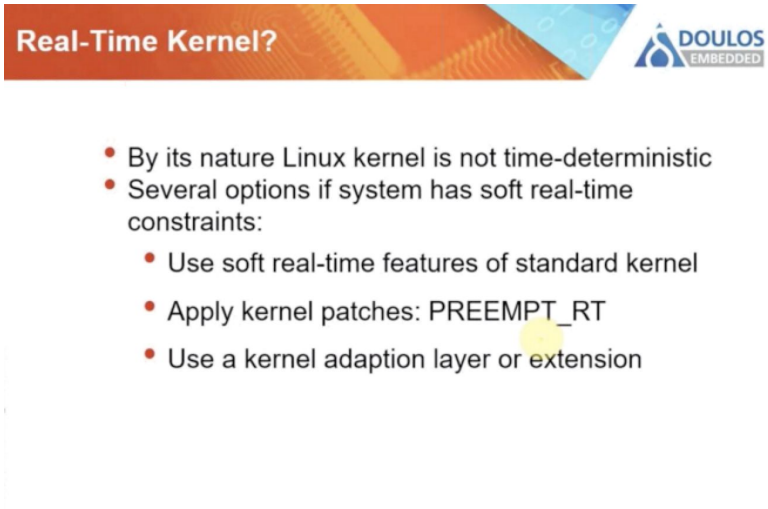
Compile kernel:

```
host$ make vmlinux
```





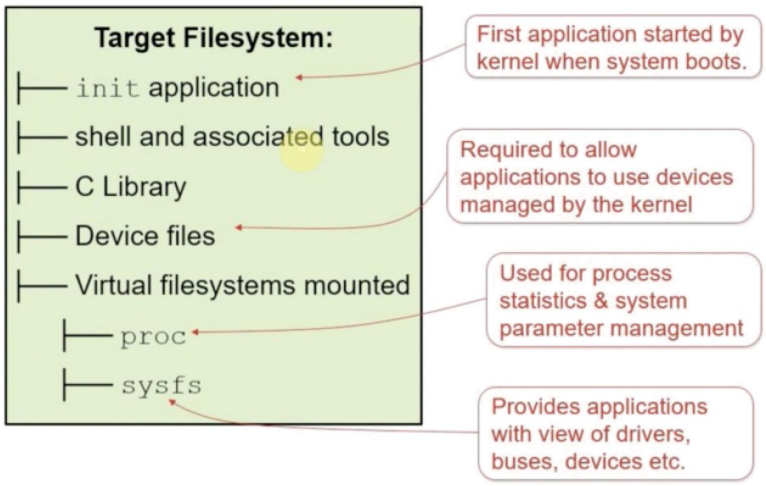
The kernel can be highly configured. For example, one could build certain drivers into the kernel binary so that when the kernel boots then those drivers are activated.You can also dynamically load a kernel module during runtime, for instance if you want the system to boot as fast as possible so you make the kernel as small as possible by loading drivers only when they are required (e.g. only load USB drivers when the USB device is plugged in). You can also use kernel modules to update drivers without rebooting the system.



The Linux kernel is not a RTOS. A Xilinx SoC typically has 2 CPUs- one running Linux, and one running a RTOS. For example, the Avnet Ultra96 eval board has a Xilinx Zynq Ultrascale+ MPSoC with two processors: a dual-core ARM cortex A53 (a low-power processor with 64-bit capabilities) and a dual-core ARM cortex R5 (a real time processor).

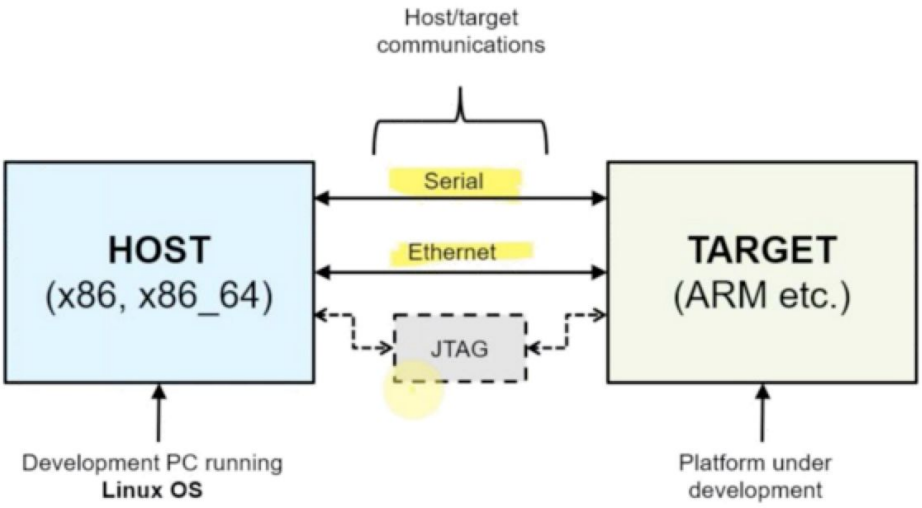
- **Filesystem**

- Here is an outline of the target Linux filesystem. Note that “device files” contains hardware. In Linux and Unix, hardware is represented as a file.



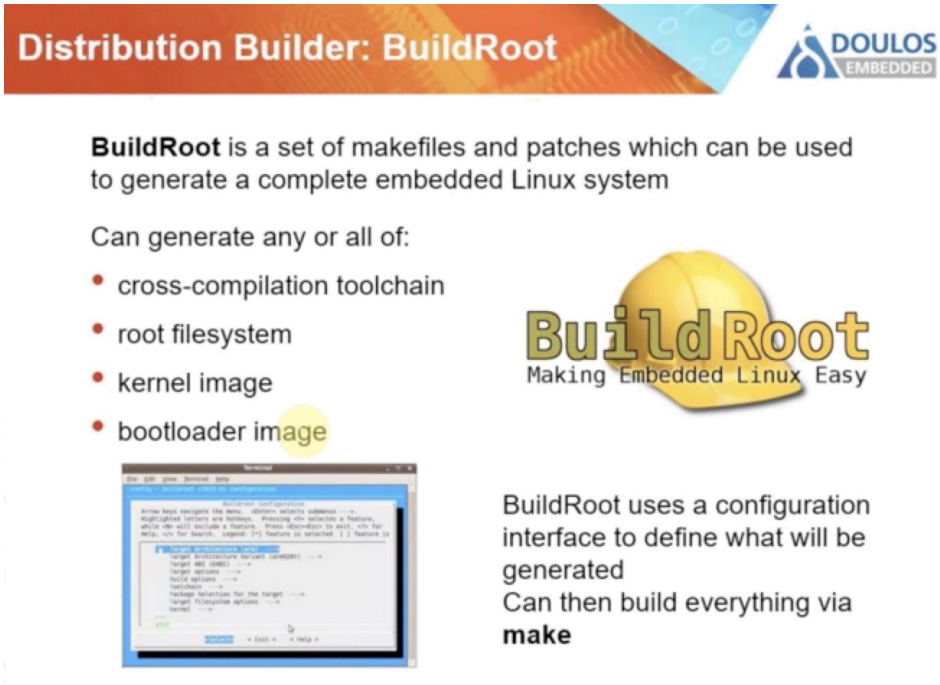
- The most common init systems daemon for Linux is systemd. sysvinit is commonly used on many embedded systems. Another alternative is a package called Busybox which provides various Linux tools including a simpler init.
- C library: a C library interfaces between the kernel and some applications. There are several C libraries available for embedded Linux: the standard GNU/Linux C library glibc, or more lightweight libraries such as musl and uClibc. glibc could be too big for your embedded system, in which case you would consider using a lightweight library.
- If you don’t want to program in C/C++ on your embedded system, other popular choices are Python, **Go** or GoLang, and **Rust**.
- A popular choice of IDE for embedded Linux is **Eclipse**. Another is **KDevelop**.
- Putting together a filesystem by hand is very, very hard.

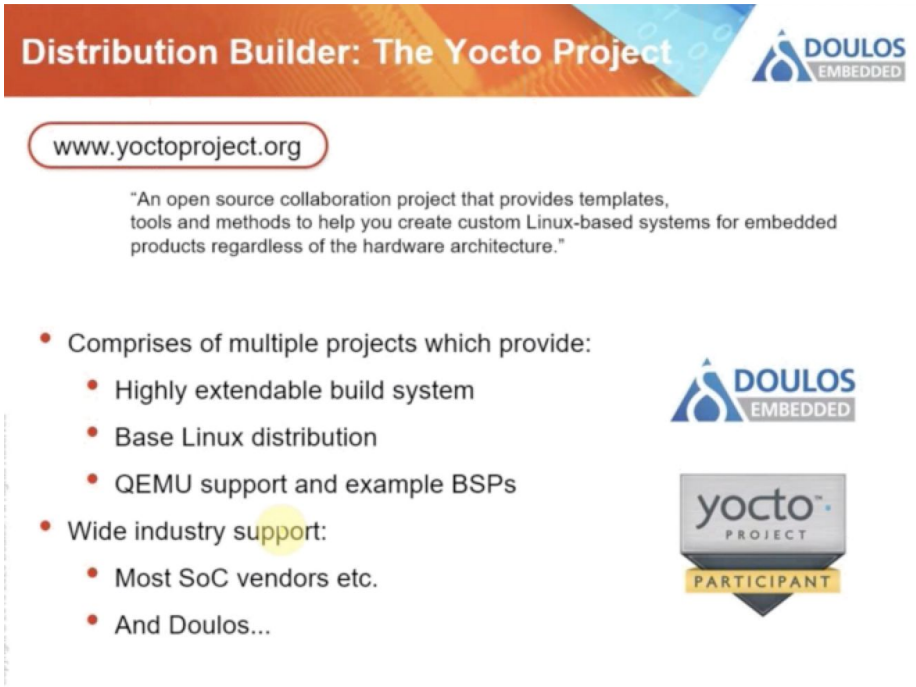
Host/Target communications



3. Distribution builders

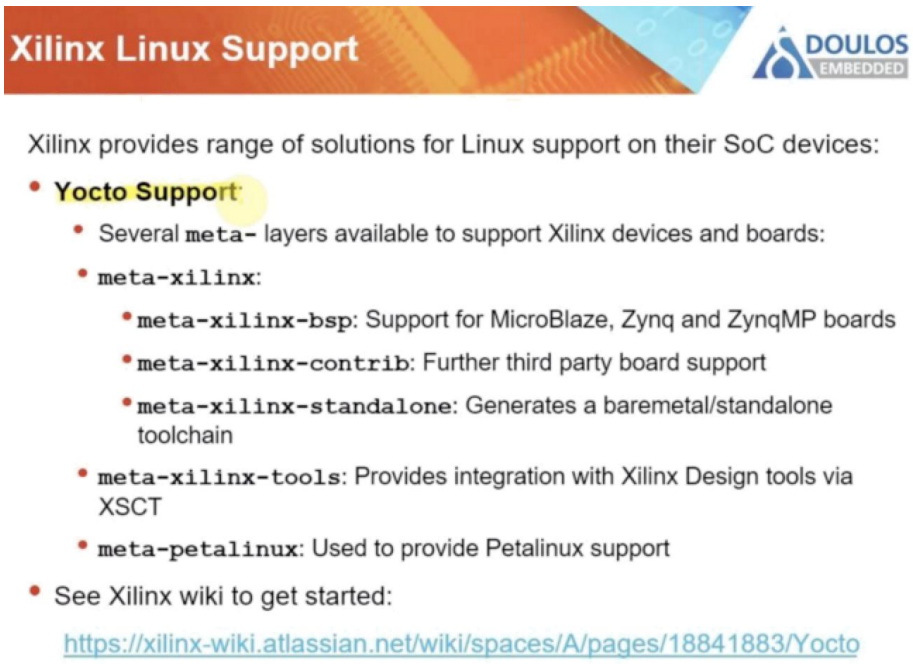
If you don't want to run a proprietary/desktop Linux distro on your embedded system, you could consider building your distro with BuildRoot or yocto. Buildroot is apparently quicker and easier to get started with than Yocto but less configurable and less scalable.





Running Linux on Xilinx devices

Xilinx SoCs are very popular. There is full Linux support for Xilinx HW accelerators, FPGAs (programmable logic), and real time processors. Xilinx even has a framework called **PetaLinux** for building a Linux distro on a Xilinx device.



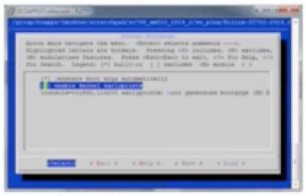
Xilinx Linux Support

DOULOS EMBEDDED

Xilinx provides range of solutions for Linux support on their SoC devices:

- **Pre-built SD card images** created using PetaLinux:
  - for Zynq & Zynq UltraScale+ MPSoC reference boards
  - Matched to Xilinx development tools releases
  - Contain all boot files including initramfs filesystem
- **PetaLinux** provides a build system framework for generating a Linux distribution for your Xilinx device:
  - Built upon Yocto but abstracted from the lower level build system
  - Less flexible than using Yocto directly but simpler to configure
  - Integrated with Xilinx design tools
  - Uses `menuconfig` style interface

https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html



Xilinx Linux Support

DOULOS EMBEDDED

Xilinx provides range of solutions for Linux support on their SoC devices:

- **Xilinx Git**: https://github.com/Xilinx
  - Open source repos for Xilinx U-Boot, ARM Trusted Firmware, Linux Kernel and more

  - Documentation and support via Xilinx Wiki:

https://xilinx-wiki.atlassian.net/wiki/spaces/A/overview

- **Ecosystem partners**:
  - Xilinx devices are supported by various commercial Linux providers

Thanks Doulos for such a great workshop!

Further resources:

-Embedded Linux wiki: https://elinux.org/Main_Page