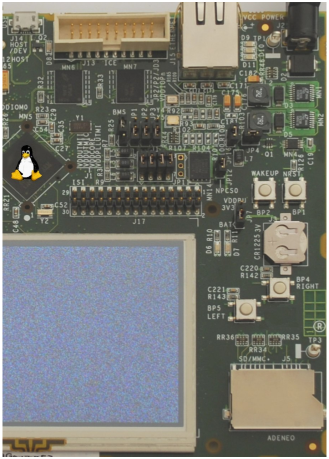
Embedded Linux Development

Free Electrons, Witekio



License

Witekio
EMBEDDING SUCCESS

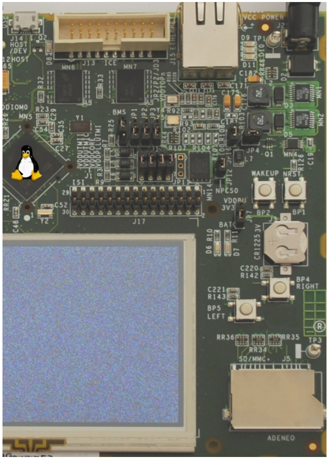
Rights and constraints

Witekio
EMBEDDING SUCCESS

Course Information

Course Information

Free Electrons, Witekio



About Witekio

- BSP and driver development
- Hardware Design and design reviews
- Systems optimization
- Embedded application development
- Support contract
- Training and Workshop
- Consulting and engineering services

Hardware for this session

IMX6Q Sabre Lite from Boundary Devices

- Freescale i.MX6Q (ARM Cortex-A9)
- Powerful CPU, with 3D acceleration and lots of peripherals.
- 1 GB of RAM
- 2 x USB host and 1 x USB device ports
- MicroSD slot
- HDMI/LVDS/RGB display ports
- SATA port
- 10/100 Ethernet

Participate!

During the lectures...

- Don't hesitate to ask questions. Other people in the audience may have similar questions too.
- This helps the trainer to detect any explanation that wasn't clear or detailed enough.
- Don't hesitate to share your experience, for example to compare Linux / Android with other operating systems used in your company.
- Your point of view is most valuable, because it can be similar to your colleagues' and different from the trainer's.
- Your participation can make our session more interactive and make the topics easier to learn.

Practical lab guidelines

During practical labs...

- We cannot support more than 8 workstations at once (each with its board and equipment). Having more would make the whole class progress slower, compromising the coverage of the whole training agenda (exception for public sessions: up to 10 people).
- So, if you are more than 8 participants, please form up to 8 working groups.
- Open the electronic copy of your lecture materials, and use it throughout the practical labs to find the slides you need again.
- Don't copy and paste from the PDF slides.
  The slides contain UTF-8 characters that look the same as ASCII ones, but won't be understood by shells or compilers.

Witekio
EMBEDDING SUCCESS

Cooperate!

As in the Free Software and Open Source community, cooperation during practical labs is valuable in this training session:

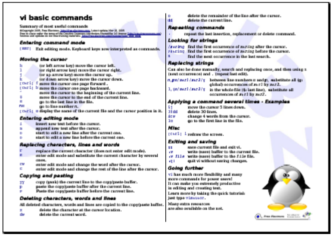
- If you complete your labs before other people, don't hesitate to help other people and investigate the issues they face. The faster we progress as a group, the more time we have to explore extra topics.
- Explain what you understood to other participants when needed. It also helps to consolidate your knowledge.
- Don't hesitate to report potential bugs to your instructor.
- Don't hesitate to look for solutions on the Internet as well.

Command memento sheet

- This memento sheet gives command examples for the most typical needs (looking for files, extracting a tar archive...)
- It saves us 1 day of UNIX / Linux command line training.
- Our best tip: in the command line shell, always hit the `Tab` key to complete command names and file paths. This avoids 95% of typing mistakes.
- Get an electronic copy on `https://bootlin.com/doc/legacy/command-line/command_memento.pdf`

vi basic commands

- The `vi` editor is very useful to make quick changes to files in an embedded target.
- Though not very user friendly at first, `vi` is very powerful and its main 15 commands are easy to learn and are sufficient for 99% of everyone's needs!
- Get an electronic copy on `https://bootlin.com/doc/legacy/command-line/vi_memento.pdf`
- You can also take the quick tutorial by running `vimtutor`. This is a worthy investment!



vi basic commands

Summary of most useful commands

©Copyright 2005, Free Electrons, http://free-electrons.com. Latest update: Mar 28, 2005
Free to share under the terms of the Creative Commons Attribution-ShareAlike 2.0 license (http://creativecommons.org)
Sources and updates on our free training materials: http://free-electrons.com/training/intro_unix_linux

Entering command mode

[ESC]	Exit editing mode. Keyboard keys now interpreted as commands.

Moving the cursor

h	(or left arrow key) move the cursor left.
l	(or right arrow key) move the cursor right.
j	(or up arrow key) move the cursor up.
k	(or down arrow key) move the cursor down.
[Ctrl] f	move the cursor one page forward .
[Ctrl] b	move the cursor one page backward.
^	move the cursor to the beginning of the current line.
$	move the cursor to the end of the current line.
G	go to the last line in the file.
nG	go to line number *n*.
[Ctrl] G	display the name of the current file and the cursor position in it.

Entering editing mode

i	insert new text before the cursor.
a	append new text after the cursor.
o	start to edit a new line after the current one.
O	start to edit a new line before the current one.

Replacing characters, lines and words

r	replace the current character (does not enter edit mode).
s	enter edit mode and substitute the current character by several ones.
cw	enter edit mode and change the word after the cursor.
C	enter edit mode and change the rest of the line after the cursor.

Copying and pasting

yy	copy (yank) the current line to the copy/paste buffer.
p	paste the copy/paste buffer after the current line.
P	Paste the copy/paste buffer before the current line.

Deleting characters, words and lines

All deleted characters, words and lines are copied to the copy/paste buffer.

x	delete the character at the cursor location.
dw	delete the current word.
D	delete the remainder of the line after the cursor.
dd	delete the current line.

Repeating commands

.	repeat the last insertion, replacement or delete command.

Looking for strings

/string	find the first occurrence of *string* after the cursor.
?string	find the first occurrence of *string* before the cursor.
n	find the next occurrence in the last search.

Replacing strings

Can also be done manually, searching and replacing once, and then using n (next occurrence) and . (repeat last edit).

:n,ps/str1/str2/g	between line numbers n and p, substitute all (g: global) occurrences of *str1* by *str2*.
:1,$s/str1/str2/g	in the whole file ($: last line), substitute all occurrences of *str1* by *str2*.

Applying a command several times - Examples

5j	move the cursor 5 lines down.
30dd	delete 30 lines.
4cw	change 4 words from the cursor.
1G	go to the first line in the file.

Misc

[Ctrl] l	redraw the screen.

Exiting and saving

ZZ	save current file and exit vi.
:w	write (save) buffer to the current file.
:w file	write (save) buffer to the *file* file.
:q!	quit vi without saving changes.

Going further

`vi` has much more flexibility and many more commands for power users!
It can make you extremely productive in editing and creating text.

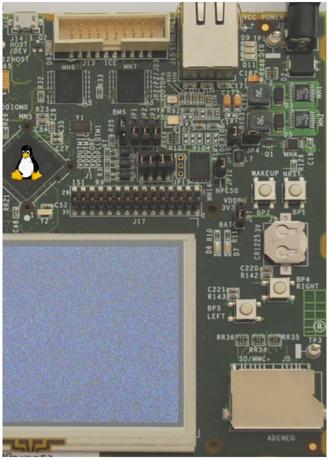
Learn more by taking the quick tutorial: just type `vimtutor`.

Many extra resources are also available on the net.

Free Electrons

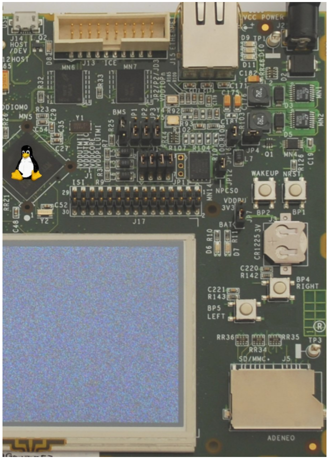
Building and running GNU Linux

Free Electrons, Witekio



Embedded Linux Introduction

Free Electrons, Witekio



Birth of free software

- 1983, Richard Stallman, **GNU project** and the **free software** concept. Beginning of the development of *gcc*, *gdb*, *glibc* and other important tools.
- 1991, Linus Torvalds, **Linux kernel project**, a Unix-like operating system kernel. Together with GNU software and many other open-source components: a completely free operating system, GNU/Linux.
- 1995, Linux is more and more popular on server systems.
- 2000, Linux is more and more popular on **embedded systems**.
- 2008, Linux is more and more popular on mobile devices.
- 2010, Linux is more and more popular on phones.

Witekio
EMBEDDING SUCCESS

Free software?

- A program is considered **free** when its license offers to all its users the following **four** freedoms:
  - Freedom to run the software for any purpose.
  - Freedom to study the software and to change it.
  - Freedom to redistribute copies.
  - Freedom to distribute copies of modified versions.
- These freedoms are granted for both commercial and non-commercial use.
- They imply the availability of source code, software can be modified and distributed to customers.
- **Good match for embedded systems !**

What is embedded Linux?

Embedded Linux is the usage of the **Linux kernel** and various **open-source** components in embedded systems

Embedded Linux work

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**
  - Custom Linux applications, generally using a given framework.

Embedded Linux Introduction

Strengths

Witekio
EMBEDDING SUCCESS

Re-using components

- The key advantage of Linux and open-source in embedded systems is the **ability** to re-use components
- The open-source ecosystem already provides many components for standard features, from hardware support to network protocols, going through multimedia, graphic, cryptographic libraries, etc.
- As soon as a hardware device, or a protocol, or a feature is wide-spread enough, high chance of having open-source components that support it.
- Allows to quickly design and develop complicated products, based on existing components.
- No-one should re-develop yet another operating system kernel, TCP/IP stack, USB stack or another graphical toolkit library.
- **Allows to focus on the added value of your product.**

Low cost

- Free software can be duplicated on as many devices as you want, free of charge.
- If your embedded system uses only free software, you can reduce the cost of software licenses to zero. Even the development tools are free, unless you choose a commercial embedded Linux edition.
- **Allows to have a higher budget for the hardware or to increase the company's skills and knowledge.**

Full control

- With open-source, you have the source code for all components in your system.
- Allows unlimited modifications, changes, tuning, debugging, optimization, for an unlimited period of time.
- Without lock-in or dependency from a third-party vendor.
  - To be true, non open-source components should be avoided when the system is designed and developed.
- **Allows to have full control over the software part of your system.**

Quality

- Many open-source components are widely used, on millions of systems.
- Usually higher quality than what an in-house development can produce, or even proprietary vendors.
- Of course, not all open-source components are of good quality, but most of the widely-used ones are.
- **Allows to design your system with high-quality components at the foundations.**

Eases testing of new features

- Open-source being freely available, it is easy to get a piece of software and evaluate it.
- Allows to easily study several options while making a choice.
- Much easier than purchasing and demonstration procedures needed with most proprietary products.
- **Allows to easily explore new possibilities and solutions.**

Community support

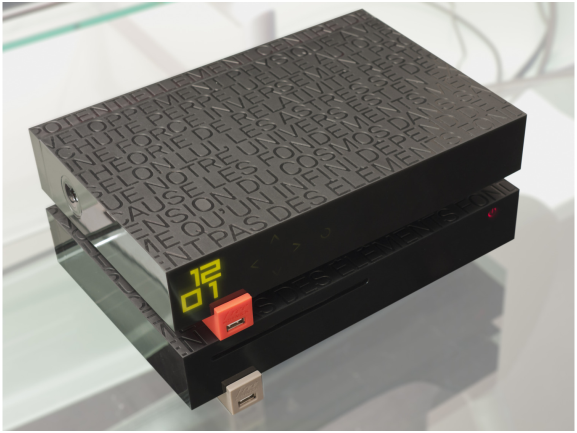
- Open-source software components are developed by communities of developers and users.
- This community can provide a high-quality support: you can directly contact the main developers of the component you are using. The likelyhood of getting an answer doesn't depend on which company you are working for.
- Often better than traditional support, but one needs to understand how the community works to properly use its support possibilities.
- **Allows to speed up the resolution of problems when developing your system.**

Taking part into the community

- Possibility of taking part into the development community of some of the components used in embedded systems: bug reporting, test of new versions or features, patches that fix bugs or add new features, etc.
- Most of the time the open-source components are not the core value of a product: it's the interest of everybody to contribute back.
- For the *engineers*: a very **motivating** way of being recognized outside the company, communication with others in the same field, **opening of new possibilities**, etc.
- For the *managers*: **motivation factor** for engineers, allows the company to be **recognized** in the open-source community and therefore get support more easily and be **more attractive** to open-source developers.

Examples

Personal routers

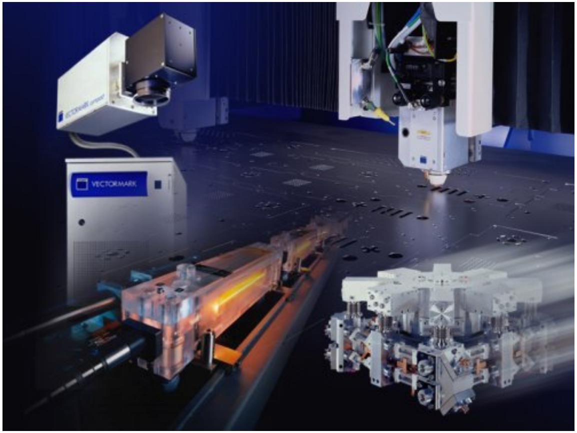


Television

Point of sale terminal

Witekio
EMBEDDING SUCCESS

Laser cutting machine



Witekio
EMBEDDING SUCCESS

Viticulture machine

Hardware

Processor and architecture (1)

- The Linux kernel and most other architecture-dependent components support a wide range of 32 and 64 bits architectures
  - x86 and x86-64, as found on PC platforms, but also embedded systems (multimedia, industrial)
  - ARM, with hundreds of different SoC (multimedia, industrial)
  - PowerPC (mainly real-time, industrial applications)
  - MIPS (mainly networking applications)
  - SuperH (mainly set top box and multimedia applications)
  - Blackfin (DSP architecture)
  - Microblaze (soft-core for Xilinx FPGA)
  - Coldfire, SCore, Tile, Xtensa, Cris, FRV, AVR32, M32R

Processor and architecture (2)

- Both MMU and no-MMU architectures are supported, even though no-MMU architectures have a few limitations.
- Linux is not designed for small microcontrollers.
- Besides the toolchain, the bootloader and the kernel, all other components are generally **architecture-independent**

RAM and storage

- **RAM**: a very basic Linux system can work within 8 MB of RAM, but a more realistic system will usually require at least 32 MB of RAM. Depends on the type and size of applications.
- **Storage**: a very basic Linux system can work within 4 MB of storage, but usually more is needed.
  - Flash storage is supported, both NAND and NOR flash, with specific filesystems
  - Block storage including SD/MMC cards and eMMC is supported
- Not necessarily interesting to be too restrictive on the amount of RAM/storage: having flexibility at this level allows to re-use as many existing components as possible.

Communication

- The Linux kernel has support for many common communication buses
  - I2C
  - SPI
  - CAN
  - 1-wire
  - SDIO
  - USB
- And also extensive networking support
  - Ethernet, Wifi, Bluetooth, CAN, etc.
  - IPv4, IPv6, TCP, UDP, SCTP, DCCP, etc.
  - Firewalling, advanced routing, multicast

Types of hardware platforms

- **Evaluation platforms** from the SoC vendor. Usually expensive, but many peripherals are built-in. Generally unsuitable for real products.
- **Component on Module**, a small board with only CPU/RAM/flash and a few other core components, with connectors to access all other peripherals. Can be used to build end products for small to medium quantities.
- **Community development platforms**, to make a particular SoC popular and easily available. These are ready-to-use and low cost, but usually have less peripherals than evaluation platforms. To some extent, can also be used for real products.
- **Custom platform**. Schematics for evaluation boards or development platforms are more and more commonly freely available, making it easier to develop custom platforms.

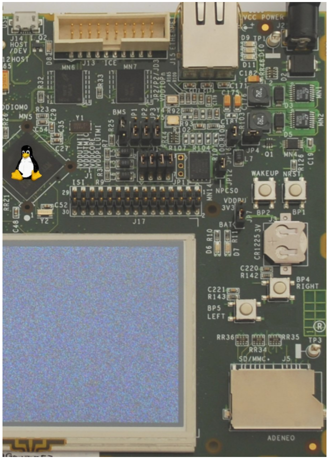
Criteria for choosing the hardware

- Make sure the hardware you plan to use is already supported by the Linux kernel, and has an open-source bootloader, especially the SoC you're targeting.
- Having support in the official versions of the projects (kernel, bootloader) is a lot better: quality is better, and new versions are available.
- Some SoC vendors and/or board vendors do not contribute their changes back to the mainline Linux kernel. Ask them to do so, or use another product if you can. A good measurement is to see the delta between their kernel and the official one.
- **Between properly supported hardware in the official Linux kernel and poorly-supported hardware, there will be huge differences in development time and cost.**

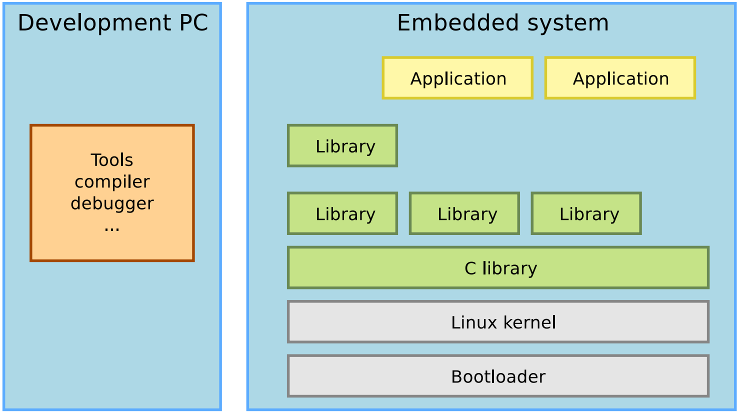
Embedded Linux System Architecture

Embedded Linux System Architecture

Free Electrons, Witekio



Global architecture



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.

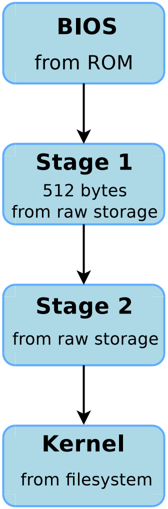
Bootloaders

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.
  - Loading of data from storage or network, memory inspection, hardware diagnostics and testing, etc.

Bootloaders on x86 (1)

- The x86 processors are typically bundled on a board with a non-volatile memory containing a program, the BIOS.
- This program gets executed by the CPU after reset, and is responsible for basic hardware initialization and loading of a small piece of code from non-volatile storage.
  - This piece of code is usually the first 512 bytes of a storage device
- This piece of code is usually a 1st stage bootloader, which will load the full bootloader itself.
- The bootloader can then offer all its features. It typically understands filesystem formats so that the kernel file can be loaded directly from a normal filesystem.

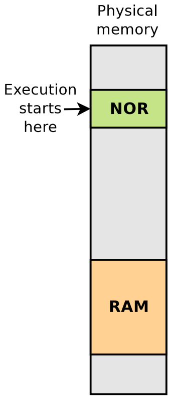


Bootloaders on x86 (2)

- GRUB, Grand Unified Bootloader, the most powerful one. `http://www.gnu.org/software/grub/`
  - Can read many filesystem formats to load the kernel image and the configuration, provides a powerful shell with various commands, can load kernel images over the network, etc.
  - See our dedicated presentation for details: `http://free-electrons.com/docs/grub/`
- Syslinux, for network and removable media booting (USB key, CD-ROM)
  `http://www.kernel.org/pub/linux/utils/boot/syslinux/`

Booting on embedded CPUs: case 1

- When powered, the CPU starts executing code at a fixed address
- There is no other booting mechanism provided by the CPU
- 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
- The first stage bootloader must be programmed at this address in the NOR
- NOR is mandatory, because it allows random access, which NAND doesn't allow
- **Not very common anymore** (unpractical, and requires NOR flash)

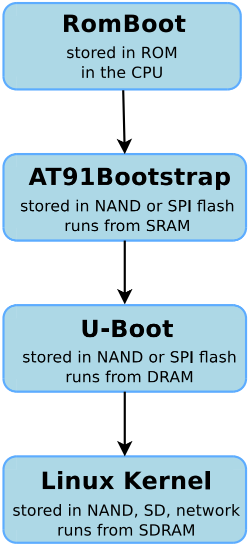


Booting on embedded CPUs: case 2

- The CPU has an integrated boot code in ROM
  - BootROM on AT91 CPUs, “ROM code” on OMAP, etc.
  - Exact details are CPU-dependent
- This boot code is able to load a first stage bootloader from a storage device into an internal SRAM (DRAM not initialized yet)
  - Storage device can typically be: MMC, NAND, SPI flash, UART, 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

Witekio
EMBEDDING SUCCESS

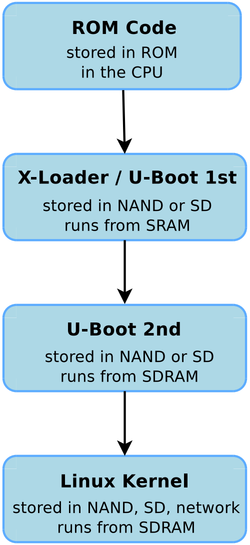
Booting on ARM Atmel AT91



- **RomBoot**: tries to find a valid bootstrap image from various storage sources, and load it into SRAM (DRAM not initialized yet). Size limited to 4 KB. No user interaction possible in standard boot mode.
- **AT91Bootstrap**: 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.
- **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.
- **Linux Kernel**: runs from RAM. Takes over the system completely (bootloaders no longer exists).

Witekio EMBEDDING SUCCESS

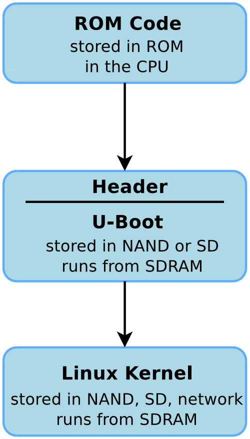
Booting on ARM TI OMAP3



- **ROM Code**: tries to find a valid bootstrap image from various storage sources, and load it into SRAM or RAM (RAM can be initialized by ROM code through a configuration header). Size limited to <64 KB. No user interaction possible.
- **X-Loader** or **U-Boot**: runs from SRAM. Initializes the DRAM, the NAND or MMC controller, and loads the secondary bootloader into RAM and starts it. No user interaction possible. File called `MLO`.
- **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.bin` or `u-boot.img`.
- **Linux Kernel**: runs from RAM. Takes over the system completely (bootloaders no longer exists).

Witekio
EMBEDDING SUCCESS

Booting on Marvell SoC



- **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.
- **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`.
- **Linux Kernel**: runs from RAM. Takes over the system completely (bootloaders no longer exists).

Generic bootloaders for embedded CPUs

- 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, NIOS, etc. The de-facto standard nowadays. We will study it in detail.
    `http://www.denx.de/wiki/U-Boot`
  - **Barebox**, a new architecture-neutral bootloader, written as a successor of U-Boot. Better design, better code, active development, but doesn't yet have as much hardware support as U-Boot.
    `http://www.barebox.org`
- There are also a lot of other open-source or proprietary bootloaders, often architecture-specific
  - RedBoot, Yaboot, PMON, etc.

Linux Kernel Overview

History

- The Linux kernel is one component of a system, which also requires libraries and applications to provide features to end users.
- The Linux kernel was created as a hobby in 1991 by a Finnish student, Linus Torvalds.
  - Linux quickly started to be used as the kernel for free software operating systems
- Linus Torvalds has been able to create a large and dynamic developer and user community around Linux.
- Nowadays, more than one thousand people contribute to each kernel release, individuals or companies big and small.

Linux license

- The whole Linux sources are Free Software released under the GNU General Public License version 2 (GPL v2).
- For the Linux kernel, this basically implies that:
  - When you receive or buy a device with Linux on it, you should receive the Linux sources, with the right to study, modify and redistribute them.
  - When you produce Linux based devices, you must release the sources to the recipient, with the same rights, with no restriction.

Witekio
EMBEDDING SUCCESS

Supported hardware architectures

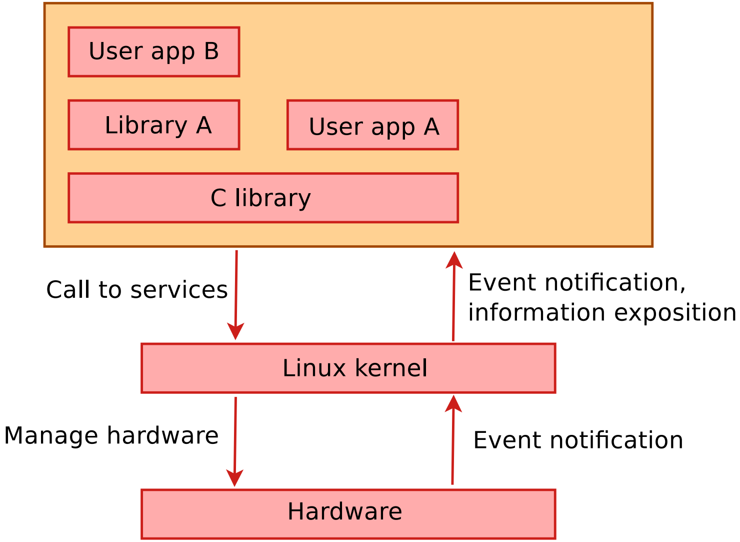
- See the `arch/` directory in the kernel sources
- Minimum: 32 bit processors, with or without MMU, and `gcc` support
- 32 bit architectures (`arch/` subdirectories)
  Examples: `arm`, `avr32`, `blackfin`, `c6x`, `m68k`, `microblaze`, `mips`, `score`, `sparc`, `um`
- 64 bit architectures:
  Examples: `alpha`, `arm64`, `ia64`, `tile`
- 32/64 bit architectures
  Examples: `powerpc`, `x86`, `sh`, `sparc`
- Find details in kernel sources: `arch/<arch>/Kconfig`, `arch/<arch>/README`, or `Documentation/<arch>/`

Witekio
EMBEDDING SUCCESS

Linux kernel key features

- Portability and hardware support. Runs on most architectures.
- Scalability. Can run on super computers as well as on tiny devices (4 MB of RAM is enough).
- Compliance to standards and interoperability.
- Exhaustive networking support.
- Security. It can't hide its flaws. Its code is reviewed by many experts.
- Stability and reliability.
- Modularity. Can include only what a system needs even at run time.
- Easy to program. You can learn from existing code. Many useful resources on the net.

Linux kernel in the system

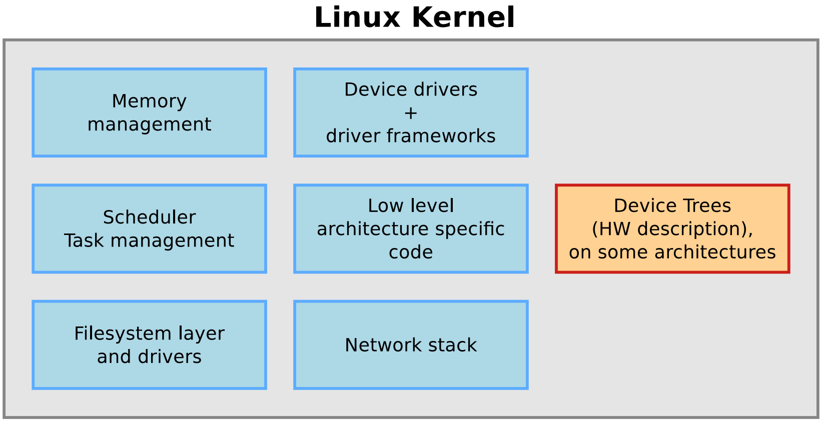


Linux kernel main roles

- **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.

Witekio
EMBEDDING SUCCESS

Inside the Linux kernel


Linux Kernel
Memory management
Device drivers + driver frameworks
Scheduler Task management
Low level architecture specific code
Device Trees (HW description), on some architectures
Filesystem layer and drivers
Network stack

🟦 Implemented mainly in C, a little bit of assembly.

🟧 Written in a Device Tree specific language.

Linux Root Filesystem

Witekio
EMBEDDING SUCCESS

Filesystems (1)

- Filesystems are used to organize data in directories and files on storage devices or on the network. The directories and files are organized as a hierarchy
- In Unix systems, applications and users see a **single global hierarchy** of files and directories, which can be composed of several filesystems.
- Filesystems are **mounted** in a specific location in this hierarchy of directories
  - When a filesystem is mounted in a directory (called *mount point*), the contents of this directory reflects the contents of the storage device
  - When the filesystem is unmounted, the *mount point* is empty again.
- This allows applications to access files and directories easily, regardless of their exact storage location

Filesystems (2)

- Create a mount point, which is just a directory

```
$ mkdir /mnt/usbkey
```

- It is empty

```
$ ls /mnt/usbkey
$
```

- Mount a storage device in this mount point

```
$ mount -t vfat /dev/sda1 /mnt/usbkey
$
```

- You can access the contents of the USB key

```
$ ls /mnt/usbkey
docs prog.c picture.png movie.avi
$
```

Witekio
EMBEDDING SUCCESS

mount / umount

- `mount` allows to mount filesystems
  - `mount -t type device mountpoint`
  - `type` is the type of filesystem
  - `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.

Witekio
EMBEDDING SUCCESS

Root filesystem

- 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 filesystem is available, the kernel panics

```
Please append a correct "root=" boot option
Kernel panic - not syncing: VFS: Unable to mount root fs on unknown block(0,0)
```

Location of the root filesystem

- It can be mounted from different locations
  - From the partition of a hard disk
  - From the partition of a USB key
  - From the partition of an SD card
  - From the partition of a NAND flash chip or similar type of storage device
  - From the network, using the NFS protocol
  - From memory, using a pre-loaded filesystem (by the bootloader)
  - etc.
- It is up to the system designer to choose the configuration for the system, and configure the kernel behaviour with `root=`

Witekio
EMBEDDING SUCCESS

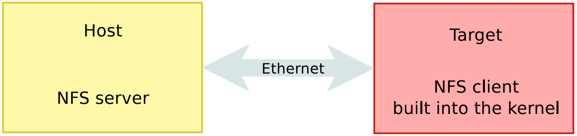
Mounting rootfs from storage devices

- 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 (1)

Once networking works, your root filesystem could be a directory on your GNU/Linux development host, exported by NFS (Network File System). This is very convenient for system development:

- Makes it very easy to update files on the root filesystem, without rebooting. Much faster than through the serial port.
- Can have a big root filesystem even if you don't have support for internal or external storage yet.
- The root filesystem can be huge. You can even build native compiler tools and build all the tools you need on the target itself (better to cross-compile though).



Mounting rootfs over the network (2)

On the development workstation side, a NFS server is needed

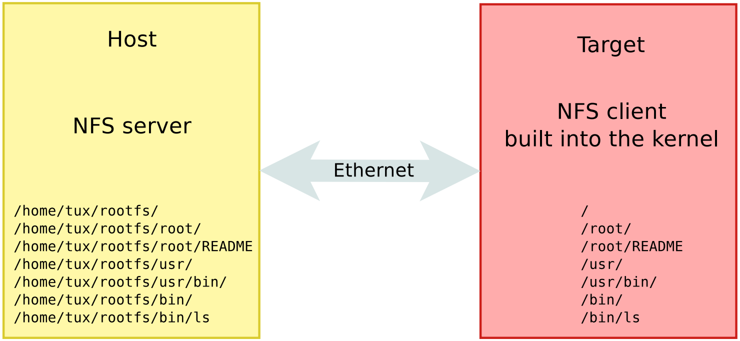
- Install an NFS server (example: Debian, Ubuntu)
  ```
  sudo apt-get install nfs-kernel-server
  ```
- Add the exported directory to your `/etc/exports` file:
  ```
  /home/tux/rootfs 192.168.1.111(rw,no_root_squash,
  no_subtree_check)
  ```
  - `192.168.1.111` is the client IP address
  - `rw,no_root_squash,no_subtree_check` 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
  sudo systemctl restart nfs-kernel-server.service
  ```

Witekio
EMBEDDING SUCCESS

Mounting rootfs over the network (3)

- On the target system
- The kernel must be compiled with
  - `CONFIG_NFS_FS=y` (NFS 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)

Mounting rootfs over the network (4)



Root filesystem organization

- The organization of a Linux root filesystem in terms of directories is well-defined by the **Filesystem Hierarchy Standard**
- `https://refspecs.linuxfoundation.org/fhs.shtml`
- Most Linux systems conform to this specification
  - Applications expect this organization
  - It makes it easier for developers and users as the filesystem organization is similar in all systems

Important directories (1)

/bin	Basic programs
/boot	Kernel image (only when the kernel is loaded from a filesystem, not common on non-x86 architectures)
/dev	Device files (covered later)
/etc	System-wide configuration
/home	Directory for the users home directories
/lib	Basic libraries
/media	Mount points for removable media
/mnt	Mount points for static media
/proc	Mount point for the proc virtual filesystem

Witekio
EMBEDDING SUCCESS

Important directories (2)

/root	Home directory of the `root` user
/sbin	Basic system programs
/sys	Mount point of the sysfs virtual filesystem
/tmp	Temporary files
/usr	/usr/bin Non-basic programs /usr/lib Non-basic libraries /usr/sbin Non-basic system programs
/var	Variable data files. This includes spool directories and files, administrative and logging data, and transient and temporary files

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 was down, some binaries and libraries are stored in `/bin`, `/sbin` and `/lib`
- `/bin` and `/sbin` contain programs like `ls`, `ifconfig`, `cp`, `bash`, etc.
- `/lib` contains the C library and sometimes a few other basic libraries
- All other programs and libraries are in `/usr`

proc virtual filesystem

- The `proc` virtual filesystem exists since the beginning of Linux
- It allows
  - The kernel to expose statistics about running processes in the system
  - The user to 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`
- `Documentation/filesystems/proc.rst` in the kernel sources
- `man proc`

proc contents

- One directory for each running process in the system
  - `/proc/<pid>`
  - `cat /proc/3840/cmdline`
  - 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 to adjust kernel parameters
  - They are called *sysctl*. See `Documentation/admin-guide/sysctl` in kernel sources.
  - Example
    `echo 3 > /proc/sys/vm/drop_caches`

sysfs filesystem

- The sysfs filesystem is a feature integrated in the 2.6 Linux kernel
- It allows 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
  ```

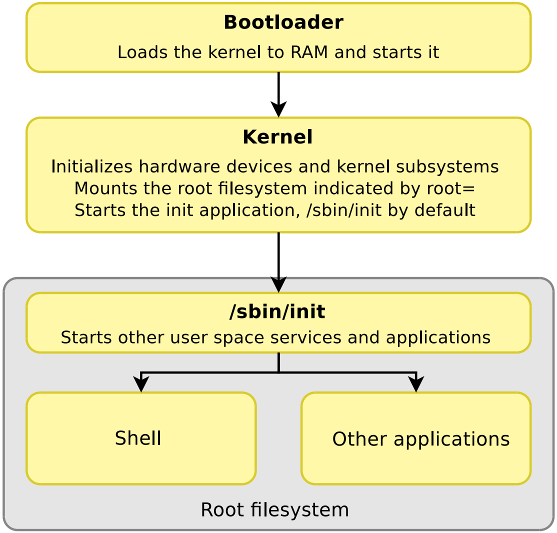
-
  ```
  $ ls /sys/
  block bus class dev devices firmware
  fs kernel module power
  ```

Basic applications

- In order to work, a Linux system needs at least a few 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 `/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 them are found, the kernel panics and the boot process is stopped.
  - The `init` application is responsible for starting all other user space applications and services
- Usually a shell, to allow a user to interact with the system
- Basic Unix applications, to copy files, move files, list files (commands like `mv`, `cp`, `mkdir`, `cat`, etc.)
- These basic components have to be integrated into the root filesystem to make it usable

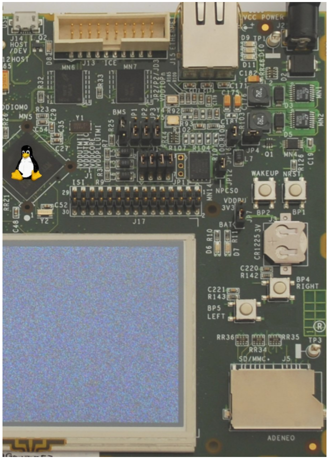
Witekio
EMBEDDING SUCCESS

Overall booting process


Bootloader
Loads the kernel to RAM and starts it
Kernel
Initializes hardware devices and kernel subsystems
Mounts the root filesystem indicated by root=
Starts the init application, /sbin/init by default
/sbin/init
Starts other user space services and applications
Shell
Other applications
Root filesystem

Embedded Linux Development Environment

Free Electrons, Witekio



Host Environment

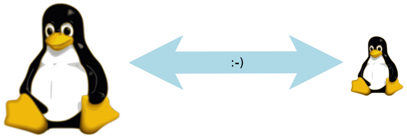
Embedded Linux solutions

- Two ways to switch to embedded Linux
  - Use **solutions provided and supported by vendors** like MontaVista, Wind River or TimeSys. These solutions come with their own development tools and environment. They use a mix of open-source components and proprietary tools.
  - Use **community solutions**. They are completely open, supported by the community.
- In our training sessions, we do not promote a particular vendor, and therefore use community solutions.
  - However, knowing the concepts, switching to vendor solutions will be easy.

Witekio EMBEDDING SUCCESS

OS for Linux development

- We strongly recommend to use Linux as the desktop operating system to embedded Linux developers, for multiple reasons.
- All community tools are developed and designed to run on Linux. Trying to use them on other operating systems (Windows, Mac OS X) will lead to trouble, and their usage on these systems is generally not supported by community developers.
- As Linux also runs on the embedded device, all the knowledge gained from using Linux on the desktop will apply similarly to the embedded device.



Desktop Linux distribution

- **Any good and sufficiently recent Linux desktop distribution** can be used for the development workstation
  - Ubuntu, Debian, Fedora, openSUSE, Red Hat, etc.
- We have chosen Ubuntu, as it is a **widely used and easy to use** desktop Linux distribution
- The Ubuntu setup on the training laptops has intentionally been left untouched after the normal installation process. Learning embedded Linux is also about learning the tools needed on the development workstation!

Linux root and non-root users

- Linux is a multi-user operating system
  - The **root user is the administrator**, and it can do privileged operations such as: mounting filesystems, configuring the network, creating device files, changing the system configuration, installing or removing software
  - All **other users are unprivileged**, and cannot perform these administrator-level operations
- On an Ubuntu system, it is not possible to log in as `root`, only as a normal user.
- The system has been configured so that the user account created first is allowed to run privileged operations through a program called `sudo`.
  - Example: `sudo mount /dev/sda2 /mnt/disk`

Software packages

- The distribution mechanism for software in GNU/Linux is different from the one in Windows
- Linux distributions provides a central and coherent way of installing, updating and removing applications and libraries: **packages**
- Packages contains the application or library files, and associated meta-information, such as the version and the dependencies
  - `.deb` on Debian and Ubuntu, `.rpm` on Red Hat, Fedora, openSUSE
- Packages are stored in **repositories**, usually on HTTP or FTP servers
- You should only use packages from official repositories for your distribution, unless strictly required.

Managing software packages (1)

Instructions for Debian based GNU/Linux systems (Debian, Ubuntu...)

- Package repositories are specified in `/etc/apt/sources.list`
- To update package repository lists:
  ```
  sudo apt-get update
  ```
- To find the name of a package to install, the best is to use the search engine on `http://packages.debian.org` or on `http://packages.ubuntu.com`. You may also use:
  ```
  apt-cache search <keyword>
  ```

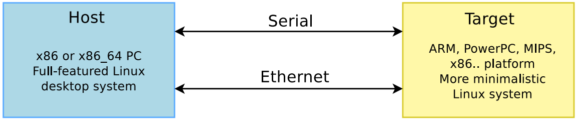
Managing software packages (2)

- To install a given package:
  `sudo apt-get install <package>`
- To remove a given package:
  `sudo apt-get remove <package>`
- To install all available package updates:
  `sudo apt-get dist-upgrade`
- Get information about a package:
  `apt-cache show <package>`
- Graphical interfaces
  - Synaptic for GNOME
  - KPackageKit for KDE

Further details on package management:
`http://www.debian.org/doc/manuals/apt-howto/`

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, etc.
- In this training session, we recommend using the simplest of them: `picocom`
  - Installation with `sudo apt-get install picocom`
  - Run with `picocom -b BAUD_RATE /dev/SERIAL_DEVICE`
  - Exit with `Control-A Control-X`
- `SERIAL_DEVICE` is typically
  - `ttyUSBx` for USB to serial converters
  - `ttySx` for real serial ports

Command line tips

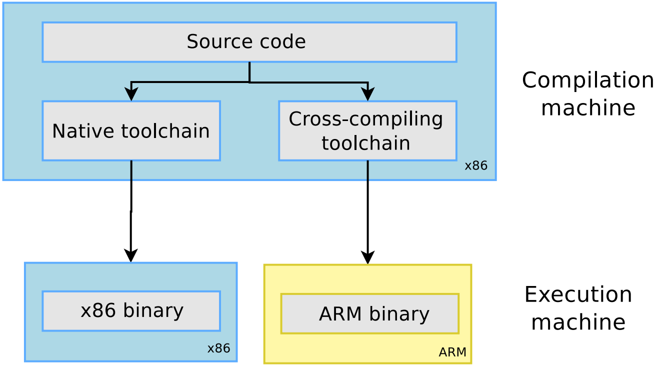
- Using the command line is mandatory for many operations needed for embedded Linux development
- It is a very powerful way of interacting with the system, with which you can save a lot of time.
- Some useful tips
  - You can use several tabs in the Gnome Terminal
  - Remember that you can use relative paths (for example: `../../linux`) in addition to absolute paths (for example: `/home/user`)
  - In a shell, hit `[Control] [r]`, then a keyword, will search through the command history. Hit `[Control] [r]` again to search backwards in the history
  - You can copy/paste paths directly from the file manager to the terminal by drag-and-drop.

Cross-compiling Toolchains

Definition (1)

- 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.

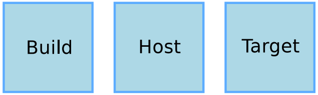
Definition (2)



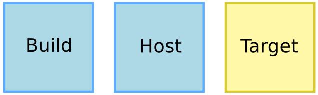
Machines in build procedures

- 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.
- Four common build types are possible for toolchains

Different toolchain build procedures

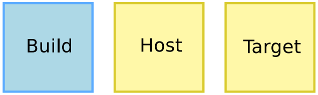


Native build
used to build the normal gcc
of a workstation

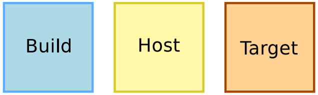


Cross build
used to build a toolchain that runs
on your workstation but generates
binaries for the target

The most common case in embedded development

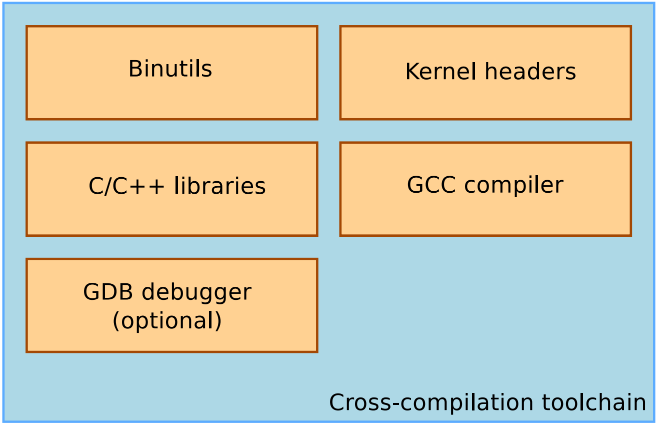


Cross-native build
used to build a toolchain that runs on your
target and generates binaries for the target



Canadian build
used to build on architecture A a
toolchain that runs on architecture B
and generates binaries for architecture C

Components

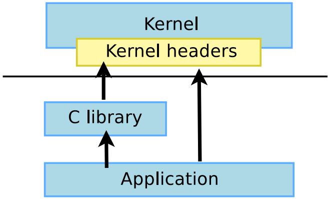

Binutils
Kernel headers
C/C++ libraries
GCC compiler
GDB debugger (optional)
Cross-compilation toolchain

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, used for libraries
  - `objdump`, `readelf`, `size`, `nm`, `strings`, to inspect binaries. Very useful analysis tools!
  - `strip`, to strip useless parts of binaries in order to reduce their size
- `http://www.gnu.org/software/binutils/`
- GPL license

Kernel headers (1)

- The C library and compiled programs needs to interact with the kernel
  - Available system calls and their numbers
  - Constant definitions
  - Data structures, etc.
- Therefore, compiling the C library requires kernel headers, and many applications also require them.
- Available in `<linux/...>` and `<asm/...>` and a few other directories corresponding to the ones visible in `include/` in the kernel sources



Kernel headers (2)

- System call numbers, in `<asm/unistd.h>`

```
#define __NR_exit           1
#define __NR_fork           2
#define __NR_read           3
```

- Constant definitions, here in `<asm-generic/fcntl.h>`, included from `<asm/fcntl.h>`, included from `<linux/fcntl.h>`

```
#define O_RDWR 00000002
```

- Data structures, here in `<asm/stat.h>`

```
struct stat {
    unsigned long st_dev;
    unsigned long st_ino;
    [...]
};
```

Witekio
EMBEDDING SUCCESS

Kernel headers (3)

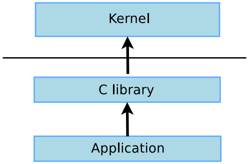
- The kernel to user space ABI is **backward compatible**
  - Binaries generated with a toolchain using kernel headers older than the running kernel will work without problem, but won't be able to use the new system calls, data structures, etc.
  - Binaries generated with a toolchain using kernel headers newer than the running kernel might work on if they don't use the recent features, otherwise they will break
  - Using the latest kernel headers is not necessary, unless access to the new kernel features is needed
- The kernel headers are extracted from the kernel sources using the `headers_install` kernel Makefile target.

GCC

- GNU Compiler Collection, the famous free software compiler
- Can compile C, C++, Ada, Fortran, Java, Objective-C, Objective-C++, and generate code for a large number of CPU architectures, including ARM, AVR, Blackfin, CRIS, FRV, M32, MIPS, MN10300, PowerPC, SH, v850, i386, x86_64, IA64, Xtensa, etc.
- `http://gcc.gnu.org/`
- Available under the GPL license, libraries under the LGPL.

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*, *eglibc*, *dietlibc*, *newlib*, etc.
- The choice of the C library must be made at the time of the cross-compiling toolchain generation, as the GCC compiler is compiled against a specific C library.



glibc

- License: LGPL
- C library from the GNU project
- Designed for performance, standards compliance and portability
- Found on all GNU / Linux host systems
- Of course, actively maintained
- Quite big for small embedded systems: approx 2.5 MB on ARM (version 2.9 - `libc`: 1.5 MB, `libm`: 750 KB)
- `http://www.gnu.org/software/libc/`

uClibc

- License: LGPL
- Lightweight C library for small embedded systems
  - High configurability: many features can be enabled or disabled through a menuconfig interface
  - Works only with Linux/uClinux, works on most embedded architectures
  - No guaranteed binary compatibility. May need to recompile applications when the library configuration changes.
  - Focus on size rather than performance
  - Small compile time
- `http://www.uclibc.org/`

uClibc (2)

- Most of the applications compile with uClibc. This applies to all applications used in embedded systems.
- Size (arm): 4 times smaller than glibc!
  - uClibc 0.9.30.1: approx. 600 KB (libuClibc: 460 KB, libm: 96KB)
  - glibc 2.9: approx 2.5 MB
- Some features not available or limited: priority-inheritance mutexes, NPTL support is very new, fixed Name Service Switch functionality, etc.
- Used on a large number of production embedded products, including consumer electronic devices
- Supported by all commercial embedded Linux providers (proof of maturity).
- Warning: though some development is still happening, the maintainers have stopped making releases since May 2012. The project is in trouble.

eglibc

- *Embedded glibc*, LGPL license too
- Was a variant of glibc, better adapted to the needs of embedded systems, because of past disagreement with glibc maintainers.
- eglibc's goals included reduced footprint, configurable components, better support for cross-compilation and cross-testing.
- Could be built without support for NIS, locales, IPv6, and many other features.
- Fortunately for eglibc, the glibc maintainer has changed and its features are now included in glibc. Version 2.19 (Feb. 2014) was the last release.
- `http://www.eglibc.org/home`

Witekio
EMBEDDING SUCCESS

Honey, I shrunk the programs!

- Executable size comparison on ARM, tested with *eglibc* 2.15 and *uClibc* 0.9.33.2
- Plain "hello world" program (stripped):

helloworld	static	dynamic
uClibc	18kB	2.5kB
uClibc with Thumb-2	14kB	2.4kB
eglibc with Thumb-2	361kB	2.7kB

- Busybox (stripped):

busybox	static	dynamic
uClibc	750kB	603kB
uClibc with Thumb-2	533kB	439kB
eglibc with Thumb-2	934kB	444kB

Other smaller C libraries

- Several other smaller C libraries have been developed, but none of them have the goal of allowing the compilation of large existing applications
- They need specially written programs and applications
- Choices:
  - Dietlibc, `http://www.fefe.de/dietlibc/`. Approximately 70 KB.
  - Newlib, `http://sourceware.org/newlib/`
  - Klibc, `http://www.kernel.org/pub/linux/libs/klibc/`, designed for use in an *initramfs* or *initrd* at boot time.

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 must be 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*
- `http://en.wikipedia.org/wiki/Application_Binary_Interface`

Witekio
EMBEDDING SUCCESS

Floating point support

- Some processors have a floating point unit, some others do not.
  - For example, many ARMv4 and ARMv5 CPUs do not have a floating point unit. Since ARMv7, a VFP unit is mandatory.
- 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
- Decision taken at toolchain configuration time
- Also possible to configure which floating point unit should be used

Witekio
EMBEDDING SUCCESS

CPU optimization flags

- A set of cross-compiling tools is specific to a CPU architecture (ARM, x86, MIPS, PowerPC)
- However, with the `-march=`, `-mcpu=`, `-mtune=` options, one can select more precisely the target CPU type
  - For example, `-march=armv7 -mcpu=cortex-a8`
- At the toolchain compilation time, values can be chosen. They are used:
  - As the default values for the cross-compiling tools, when no other `-march`, `-mcpu`, `-mtune` options are passed
  - To compile the C library
- Even if the C library has been compiled for armv5t, it doesn't prevent from compiling other programs for armv7

Building a toolchain manually

Building a cross-compiling toolchain by yourself is a difficult and painful task! Can take days or weeks!

- Lots of details to learn: many components to build, complicated configuration
- Lots of decisions to make (such as C library version, ABI, floating point mechanisms, component versions)
- Need kernel headers and C library sources
- Need to be familiar with current `gcc` issues and patches on your platform
- Useful to be familiar with building and configuring tools
- See the *Crosstool-NG* `docs/` directory for details on how toolchains are built.

Witekio
EMBEDDING SUCCESS

Get a pre-compiled toolchain

- Solution that many people choose
  - Advantage: it is the simplest and most convenient solution
  - Drawback: you can't fine tune the toolchain to your needs
- Determine what toolchain you need: CPU, endianism, C library, component versions, ABI, soft float or hard float, etc.
- Check whether the available toolchains match your requirements.
- Possible choices
  - Sourcery CodeBench toolchains
  - Linaro toolchains
  - Toolchains packaged by your distribution
    Ubuntu example:
    `sudo apt-get install gcc-arm-linux-gnueabi`
  - More references at `http://elinux.org/Toolchains`

Sourcery CodeBench

- *CodeSourcery* was a company with an extended expertise on free software toolchains: gcc, gdb, binutils and glibc. It has been bought by *Mentor Graphics*, which continues to provide similar services and products
- They sell toolchains with support, but they also provide a "*Lite*" version, which is free and usable for commercial products
- They have toolchains available for
  - ARM
  - MIPS
  - PowerPC
  - SuperH
  - x86
- Be sure to use the GNU/Linux versions. The EABI versions are for bare-metal development (no operating system)

Linaro toolchains

- Linaro contributes to improving mainline gcc on ARM, in particular by hiring CodeSourcery developers.
- For people who can't wait for the next releases of gcc, Linaro releases modified sources of stable releases of gcc, with these optimizations for ARM (mainly for recent Cortex A CPUs).
- As any gcc release, these sources can be used by build tools to build their own binary toolchains (Buildroot, OpenEmbedded...). This allows to support glibc, uClibc and eglibc.

Installing and using a pre-compiled toolchain

- Follow the installation procedure proposed by the vendor
- Usually, it is simply a matter of extracting a tarball wherever you want.
- Then, add the path to toolchain binaries in your `PATH`:
  `export PATH=/path/to/toolchain/bin/:$PATH`
- Finally, compile your applications
  `PREFIX-gcc -o foobar foobar.c`
- `PREFIX` depends on the toolchain configuration, and allows to distinguish cross-compilation tools from native compilation utilities

Toolchain building utilities

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

- Same advantage as the pre-compiled toolchains: you don't need to mess up with all the details of the build process
- But also offers more flexibility in terms of toolchain configuration, component version selection, etc.
- They also usually contain several patches that fix known issues with the different components on some architectures
- Multiple tools with identical principle: shell scripts or Makefile that automatically fetch, extract, configure, compile and install the different components

Witekio
EMBEDDING SUCCESS

Toolchain building utilities (2)

- **Crosstool-ng**
  - Rewrite of the older Crosstool, with a menuconfig-like configuration system
  - Feature-full: supports uClibc, glibc, eglibc, hard and soft float, many architectures
  - Actively maintained
  - `http://crosstool-ng.org/`

Toolchain building utilities (3)

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

- **Buildroot**
  - Makefile-based. Can build (e)glibc, uClibc and musl based toolchains, for a wide range of architectures.
  - `http://www.buildroot.net`
- **PTXdist**
  - Makefile-based, uClibc or glibc, maintained mainly by *Pengutronix*
  - `https://www.ptxdist.org/`
- **OpenEmbedded / Yocto**
  - A featureful, but more complicated build system
  - `http://www.openembedded.org/`
  - `https://www.yoctoproject.org/`

Witekio
EMBEDDING SUCCESS

Crosstool-NG: installation and usage

- 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

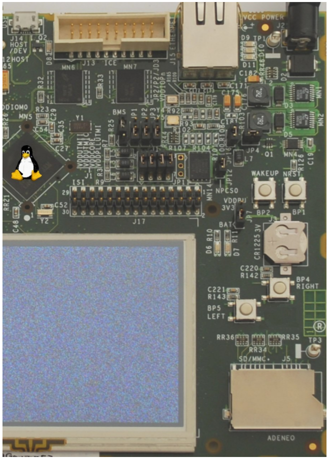
```
./ct-ng build
```

Toolchain contents

- The cross compilation tool binaries, in `bin/`
  - This directory can 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.)
  - CodeSourcery ARM toolchain are multilib, the sysroots are 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`

Embedded Linux System Building

Free Electrons, Witekio



System building: goal and solutions

- Goal:
  - Integrate all the needed software components, both third-party and in-house, into a working root filesystem.
  - Involves download, extraction, configuration, compilation and installation of all these components, possibly having to fix issues and to adapt configurations (at build and/or at run time).
- Several solutions:
  - Manually, package per package.
  - Distributions or ready-made filesystems.
  - System bakers.

Manual Build

System building: manually (1)

- Manually building a target system involves downloading, configuring, compiling and installing all the components of the system.
- All the libraries and dependencies must be configured, compiled and installed in the right order.
- Sometimes, the build system used by libraries or applications is not very cross-compile friendly, so some adaptations are necessary.
- There is no infrastructure to reproduce the build from scratch, which might cause problems if one component needs to be changed, if somebody else takes over the project, etc.

System building: manually (2)

- Manual system building is not recommended for production projects
- However, using automated tools often requires the developer to dig into specific issues
- Having a basic understanding of how a system can be built manually is therefore very useful to fix issues encountered with automated tools
  - We will first study manual system building, and during a practical lab, create a system using this method
  - Then, we will study the automated tools available, and use one of them during a lab

System foundations

- A basic root file system needs at least
  - A traditional directory hierarchy, with `/bin`, `/etc`, `/lib`, `/root`, `/usr/bin`, `/usr/lib`, `/usr/share`, `/usr/sbin`, `/var`, `/sbin`
  - A set of basic utilities, providing at least the `init` program, a shell and other traditional Unix command line tools. This is usually provided by *Busybox*
  - The C library and the related libraries (thread, math, etc.) installed in `/lib`
  - A few configuration files, such as `/etc/inittab`, and initialization scripts in `/etc/init.d`
- On top of this foundation common to most embedded Linux system, we can add third-party or in-house components

Target and build spaces

- The system foundation, Busybox and C library, are the core of the target root filesystem
- However, when building other components, one must distinguish two directories
  - The *target* space, which contains the target root filesystem, everything that is needed for **execution** of the application
  - The *build* space, which will contain a lot more files than the *target* space, since it is used to keep everything needed to **compile** libraries and applications. So we must keep the headers, documentation, and other configuration files

Witekio
EMBEDDING SUCCESS

Package builders

- Each open-source component comes with a mechanism to configure, compile and install it
  - A basic `Makefile`
    - Need to read the `Makefile` to understand how it works and how to tweak it for cross-compilation
  - A build system based on the *Autotools*
    - As this is the most common build system, we will study it in details
  - CMake, `http://www.cmake.org/`
    - Newer and simpler than the *autotools*. Used by large projects such as KDE or Second Life
  - Scons, `http://www.scons.org/`
  - Other manual build systems

Ready Made Filesystems

Distributions - Debian

Debian GNU/Linux, `http://www.debian.org`

- Provides the easiest environment for quickly building prototypes and developing applications. Countless runtime and development packages available.
- But probably too costly to maintain and unnecessarily big for production systems.
- Available on ARM, MIPS and PowerPC architectures
- Software is compiled natively by default.

Distributions - Others

Fedora

- `http://fedoraproject.org/wiki/Architectures/ARM`
- Supported on various recent ARM boards (such as Beaglebone Black). Pidora supports Raspberry Pi too.
- Supports QEMU emulated ARM boards too (Versatile Express board)
- Shipping the same version as for desktops!

Ubuntu

- Had some releases for ARM mobile multimedia devices, but stopped at version 12.04. Now focusing on ARM servers only.

Embedded distributions

Distributions designed for specific types of devices

- **Android**: `http://www.android.com/`
  Google's distribution for phones and tablet PCs. Except the Linux kernel, very different userspace than other Linux distributions. Very successful, lots of applications available (many proprietary).
- **Ångström**:
  Targets PDAs and webpads (Siemens Simpad...)
  Binaries available for arm little endian.

Application frameworks

Not real distributions you can download. Instead, they implement middleware running on top of the Linux kernel and allowing to develop applications.

- **Mer**: `http://merproject.org/`
  Fork from the Meego project.
  Targeting mobile devices.
  Supports x86, ARM and MIPS.
  See `http://en.wikipedia.org/wiki/Mer_(software_distribution)`
- **Tizen**: `https://www.tizen.org/`
  Targeting smartphones, tablets, netbooks, smart TVs and In Vehicle Infotainment devices.
  Supported by big phone manufacturers and operators
  HTML5 base application framework.
  Likely to compete against Android!
  See `http://en.wikipedia.org/wiki/Tizen`

Commercial embedded Linux solutions

Caution: *commercial* doesn't mean *proprietary!*

- Vendors play fair with the GPL and do make their source code available to their users, and most of the time, eventually to the community.
  - As long as they distribute the sources to their users, the GPL doesn't require vendors to share their sources with any third party.
- Graphical toolkits developed by the vendors are usually proprietary, trying to make it easier to create and embedded Linux systems.
- Major players: Wind River, Montavista, TimeSys

Commercial solution strengths

- Technical advantages
  - Well tested and supported kernel and tool versions
  - Often supporting patches not supported by the mainline kernel yet (example: real-time patches)
- Complete development tool sets: kernels, toolchains, utilities, binaries for impressive lists of target platforms
- Integrated utilities for automatic kernel image, initramfs and filesystem generation.
- Graphical developments tools
- Development tools available on multiple platforms: GNU / Linux, Solaris, Windows...
- Support services
  - Useful if you don't have your own support resources
  - Long term support commitment, even for versions considered as obsolete by the community, but not by your users!

Commercial or community solutions?

Commercial distributions and tool sets

- Best if you don't have your own support resources and have a sufficient budget
- Help focusing on your primary job: making an embedded device.
- You can even subcontract driver development to the vendor

Community distributions and tools

- Best if you are on a tight budget
- Best if you are willing to build your own embedded Linux expertise, investigate issues by yourselves, and train your own support resources.

In any case, your products are based on Free Software!

System Bakers

Principle

- Different tools are available to automate the process of building a target system, including the kernel, and sometimes the toolchain.
- They automatically download, configure, compile and install all the components in the right order, sometimes after applying patches to fix cross-compiling issues.
- They already contain a large number of packages, that should fit your main requirements, and are easily extensible.
- The build becomes reproducible, which allows to easily change the configuration of some components, upgrade them, fix bugs, etc.

Available system building tools

Large choice of tools

- **Buildroot**, developed by the community
  `http://www.buildroot.net`
- **PTXdist**, developed by Pengutronix
  `https://www.ptxdist.org/`
- **OpenWRT**, originally a fork of Buildroot for wireless routers, now a more generic project
  `http://www.openwrt.org`
- **LTIB**. Good support for Freescale boards, but small community
  `http://ltib.org/`
- **OpenEmbedded**, more flexible but also far more complicated
  `http://www.openembedded.org`, its industrialized version **Yocto Project** and vendor-specific derivatives such as **Arago**.
- Vendor specific tools (silicon or embedded Linux vendors)

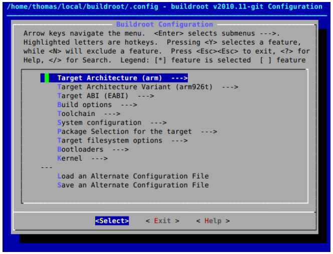
Witekio
EMBEDDING SUCCESS

Buildroot (1)

- Allows to build a toolchain, a root filesystem image with many applications and libraries, a bootloader and a kernel image.
  - Or any combination of the previous items
- Supports building uClibc, glibc, eglibc and musl toolchains, either built by Buildroot, or external.
- Over 1200+ applications or libraries integrated, from basic utilities to more elaborate software stacks: X.org, GStreamer, Qt, Gtk, WebKit, Python, PHP, etc.
- Good for small to medium embedded systems, with a fixed set of features.
  - No support for generating packages (`.deb` or `.ipk`).
  - Needs complete rebuild for most configuration changes.
- Active community, releases published every 3 months.

Buildroot (2)

- Configuration takes place through a `*config` interface similar to the kernel:
  `make menuconfig`
- Allows to define:
  - Architecture and specific CPU.
  - Toolchain configuration.
  - Set of applications and libraries to integrate.
  - Filesystem images to generate.
  - Kernel and bootloader configuration.
- Build by just running:
  `make`



```
/home/thomas/local/buildroot/.config - buildroot v2010.11-git Configuration

                          Buildroot Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus --->.
  Highlighted letters are hotkeys.  Pressing <Y> selectes a feature,
  while <N> will exclude a feature.  Press <Esc><Esc> to exit, <?> for
  Help, </> for Search.  Legend: [*] feature is selected  [ ] feature

        Target Architecture (arm)  --->
        Target Architecture Variant (arm926t)  --->
        Target ABI (EABI)  --->
        Build options  --->
        Toolchain  --->
        System configuration  --->
        Package Selection for the target  --->
        Target filesystem options  --->
        Bootloaders  --->
        Kernel  --->
    ---
        Load an Alternate Configuration File
        Save an Alternate Configuration File

                  <Select>    < Exit >    < Help >
```

Buildroot: adding a new package (1)

- A package allows to integrate a user application or library to Buildroot.
- Each package has its own directory (such as `package/gqview`). This directory contains:
  - A `Config.in` file (mandatory), describing the configuration options for the package. At least one is needed to enable the package. This file must be sourced from `package/Config.in`.
  - A `gqview.mk` file (mandatory), describing how the package is to be built.
  - Patches (optional). Each file of the form `gqview-*.patch` will be applied as a patch.

Witekio
EMBEDDING SUCCESS

Buildroot: adding a new package (2)

- For a simple package with a single configuration option to enable/disable it, the `Config.in` file looks like:

```
config BR2_PACKAGE_GQVIEW
        bool "gqview"
        depends on BR2_PACKAGE_LIBGTK2
        help
          GQview is an image viewer for Unix operating systems

          http://prdownloads.sourceforge.net/gqview
```

- It must be sourced from `package/Config.in`:

```
source "package/gqview/Config.in"
```

Witekio
EMBEDDING SUCCESS

Buildroot: adding new package (3)

- Create the gqview.mk file to describe the build steps:

```
GQVIEW_VERSION = 2.1.5
GQVIEW_SOURCE = gqview-$(GQVIEW_VERSION).tar.gz
GQVIEW_SITE = http://prdownloads.sourceforge.net/gqview
GQVIEW_DEPENDENCIES = host-pkgconf libgtk2
GQVIEW_CONF_ENV = LIBS="-lm"

$(eval $(autotools-package))
```

- The package directory and the prefix of all variables must be identical to the suffix of the main configuration option `BR2_PACKAGE_GQVIEW`.
- The `autotools-package` infrastructure knows how to build autotools packages. A more generic `generic-package` infrastructure is available for packages not using autotools as their build system.

Witekio
EMBEDDING SUCCESS

OpenEmbedded / Yocto Project

- The most versatile and powerful embedded Linux build system
  - A collection of recipes (`.bb` files).
  - A tool that processes the recipes: `bitbake`.
- Integrates 2000+ application and libraries, is highly configurable, can generate binary packages to make the system customizable, supports multiple versions/variants of the same package, no need for full rebuild when the configuration is changed.
- Configuration takes place by editing various configuration files.
- Good for larger embedded Linux systems, or people looking for more configurability and extensibility.
- Drawbacks: very steep learning curve, very long first build.

Yocto Project (1)

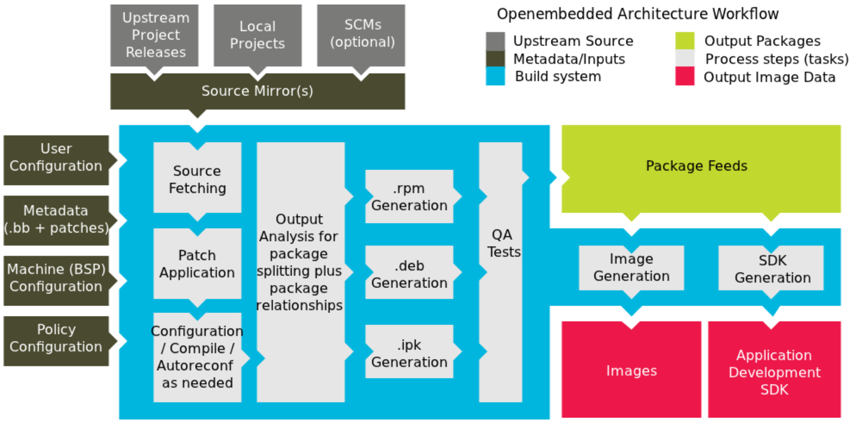
- `www.yoctoproject.org`
  - "It's not an embedded Linux distribution - it creates a custom one for you".
  - Contains a lot of "recipes", defining how to fetch, build and install packages into the target root filesystem.
  - Also contains scripts to generate support tools such as toolchains, sysroots, etc.
  - Versioning: allows to rely on a stable and validated version of the system.
  - Graphical tools to configure and build images (experimental).

Witekio
EMBEDDING SUCCESS

Yocto Project (2)

- Since 2012, the preferred way to go for dealing with complex images.
  - Well supported by many silicon vendors providing support for their SoCs.
  - Based on OpenEmbedded's extensive set of recipes.
  - Easy to customize and add new recipes.
  - Default image configurations available for quick start on evaluation boards
- For quick simple image generation, Buildroot may be easier to use.

Yocto architecture



Credits: `http://www.yoctoproject.org`

Concept of recipes

- A "recipe" is a sort of Makefile that defines several steps for building a package:
  - Standard steps: fetch, unpack, patch, compile, install, etc.
    - Generally inherit from generic steps defined into "classes" (autotools, kernel, etc).
    - Can be overriden by recipes for customization.
  - Custom steps: can be defined in recipes, order needs to be specified.
- Naming convention: `<package name>_<version>.bb`.
  - In the recipe, variables are automatically generated out of the recipe filename:
    - PN = `<package name>`.
    - PV = `<version>`.
- Recipes can inherit from "classes" and include ".inc" recipe files.

Witekio
EMBEDDING SUCCESS

Sample recipe: flex (1)

flex.inc

```
SUMMARY = "Flex (The Fast Lexical Analyzer)"
DESCRIPTION = "Flex is a fast lexical analyser generator. \
Flex is a tool for generating programs that recognize \
lexical patterns in text."
HOMEPAGE = "http://sourceforge.net/projects/flex/"

SECTION = "devel"
LICENSE = "BSD"

SRC_URI = "${SOURCEFORGE_MIRROR}/flex/flex-${PV}.tar.bz2"

inherit autotools gettext
```

Sample recipe: flex (2)

flex_2.5.35.bb

```
require flex.inc
PR = "r3"
LICENSE="BSD"
LIC_FILES_CHKSUM = " \
file://COPYING;md5=e4742cf92e89040b39486a6219b68067"

BBCLASSEXTEND = "native nativesdk"

SRC_URI += "file://avoid-FORTIFY-warnings.patch \
            file://int-is-not-the-same-size-as-size_t.patch"

SRC_URI[md5sum] = "10714e50cea54dc7a227e3eddcd44d57"
SRC_URI[sha256sum] = " \
0becbd4b2b36b99c67f8c22ab98f7f80c9860aec70f0350a0018f29"
```

Image configuration

- Machine: Select the configuration of certain recipes depending on the target platform:
  - Kernel/U-Boot: select the version and `.config` file to use.
- Image: Select the packages to install depending of the type of image (console, demo, Qt based, etc).
- Use variables from configuration files:
  - `{DISTRO,MACHINE,IMAGE}_FEATURES`: select generic features (pci, wifi, bluetooth, etc) that will automatically affect dependency trees.
  - `PREFERRED_PROVIDER_<PN>`: Define preferred version to build for the specified package.
  - `IMAGE_FSTYPES`: Format of the root filesystem image to generate (tar.bz2, ext3, sdcard, etc).

Tools

- `bitbake`
  - Command-line tool used to parse configration and recipe files and perfom related appropriate actions.
  - Written in Python.
- `hob`
  - Graphical front-end for bitbake.
  - User friendly way of configuring images.
  - Still experimental, cannot replace a good understanding of Yocto internals for managing images efficiently.

Useful tricks

- Speed-up the build process:
  - Use `BB_NUMBER_THREADS` to define the number of concurrent threads to launch when running the building tools.
  - Use `PARALLEL_MAKE` on multi-core hosts to parallelize the work when compiling packages with the `make` command.
- Save up some precious disk space:
  - `INHERIT += "rm_work"`: Perform the "rm_work" step at the end of each recipe. This will clean up the useless sources and files that have just been built.
  - Use `SSTATE_MIRRORS` to point to a remote network location. This directory contains lots of files used to save time when processing a recipe that has already been built, and grows very quickly. This directory can be shared over the network between several workstations to save space.

Concept of layers

- Customization of Yocto is done through "`layers`".
- Yocto comes with a default set of `layers` containing recipes and configuration files.
- Silicon Vendors supporting Yocto provide specific layers for supporting their architectures.
- Best way to customize or add new recipes is to create a custom layer which may contain:
  - Custom machine configuration files.
  - New or custom `.bb` recipe files.
  - Modifications targeting recipes in other layers using `.bbappend` files.
  - Anything else that may be missing in other layers.
- A custom layer should have a high priority to take precedence over the default ones.

Example

- Customize the "netbase" package to add Ethernet gadget network interface:
  - Create a custom `interfaces` file containing your config and place it under the right path in your layer: `meta-layer/recipes-core/netbase/netbase-5.0/`

    ```
    auto usb0
    iface usb0 inet dhcp
    ```

  - Create a `netbase_5.0.bbappend` customization file under `meta-layer/recipes-core/netbase/`.
  - Add the following line in `netbase_5.0.bbappend` to instruct `bitbake` to use your custom recipe directory with highest priority:

    ```
    FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}-${PV}:"
    ```

Witekio
EMBEDDING SUCCESS

Bitbake usage (1)

- Bitbake is the main tool to handle recipes and build images.
- It can be used to compile single packages:
  - `bitbake <package-name>`: Process the package in its last or preferred version (specified in distro, image, or machine specific configuration files).
  - `bitbake -b <path>/<package-name>_<package-version>.bb`: Process a specific package in a given version.
- Can process specific steps on a package:
  - `bitbake <package-name> -c <step>`: Will perform all steps off the corresponding recipe up to the specified one.
  - `bitbake virtual/kernel -c patch`: Will fetch, unpack, and patch the preferred kernel sources for the selected machine.

Bitbake usage (2)

- Generate full images:
  - `bitbake <image-name>`: Will process all the packages required by the image, then generate the bootloader, kernel, and rootfs images.
- Generate specific toolchains and SDKs:
  - `bitbake meta-toolchain`: Will generate a self-extractable shell script containing a toolchain that can be deployed on other development machines.
  - `bitbake meta-toolchain-qte`: Will generate a self-extractable shell script containing a SDK that can be directly installed and used from QtCreator on other development machines.

Witekio
EMBEDDING SUCCESS

Remote package installation using OPKG

- opkg is a tool running on the target that allows downloading and installing .ipk packages from remote repositories, much like apt-get on Debian/Ubuntu.
- Configuration of the repositories is done under /etc/opkg/*-feeds.conf files:

```
src/gz <arch> http://<ip-of-the-repo>/<arch>
src/gz <machine> http://<ip-of-the-repo>/<machine>
src/gz all http://<ip-of-the-repo>/all
```

- From the target, simple commands are called to automatically install a package and its dependencies:
  - opkg update: Fetch the list of available packages from the remote repositories.
  - opkg list: Display the list of all packages available for installation.
  - opkg install <package-name>: Fetch and install a given package and all its dependencies.

Using OPKG during development

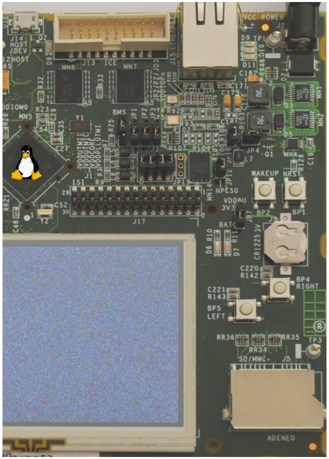
- Can be very useful to quickly add new packages to the target without re-generating and re-deploying the complete image on the target.
- There must be a TCP/IP connection between the host and the target (Ethernet over USB is fine).
- Packages must be compiled as `.ipk` packages. Set `PACKAGE_CLASSES ?= "package_ipk"` in your `local.conf` file.
- `opkg` must be part of the target image. Set `IMAGE_INSTALL_append = " opkg"` in your image configuration file.
- The host PC must expose a webserver from where to fetch the `.ipk` packages, busybox builtin `httpd` command will work just fine.
- After rebuilding a package on the host, make sure you update the package list before running `opkg update` on the target.
  - `bitbake package-index`

Witekio
EMBEDDING SUCCESS

More On Bootloaders

More On Bootloaders

Free Electrons, Witekio



More On Bootloaders

Das U-Boot

U-Boot

U-Boot is a typical free software project

- License: GPLv2 (same as Linux)
- Freely available at `http://www.denx.de/wiki/U-Boot`
- Documentation available at `https://u-boot.readthedocs.io/en/latest/`
- The latest development source code is available in a Git repository: `http://git.denx.de/?p=u-boot.git;a=summary`
- Development and discussions happen around an open mailing-list `http://lists.denx.de/pipermail/u-boot/`
- Since the end of 2008, it follows a fixed-interval release schedule. Every three months, a new version is released. Versions are named `YYYY.MM`.

Witekio
EMBEDDING SUCCESS

U-Boot configuration

- Get the source code from the website, and uncompress it
- The `include/configs/` directory contains one configuration file 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.
  - It is a simple `.h` file that sets C pre-processor constants. See the `README` file for the documentation of these constants. This file can also be adjusted to add or remove features from U-Boot (commands, etc.).
- Assuming that your board is already supported by U-Boot, there should be one entry corresponding to your board in the `boards.cfg` file.
  - Run `./tools/genboardscfg.py` to generate it.
  - Or just look in the `configs/` directory.

Witekio
EMBEDDING SUCCESS

U-Boot configuration file excerpt

```
/* CPU configuration */
#define CONFIG_ARMV7 1
#define CONFIG_OMAP 1
#define CONFIG_OMAP34XX 1
#define CONFIG_OMAP3430 1
#define CONFIG_OMAP3_IGEP0020 1
[...]
/* Memory configuration */
#define CONFIG_NR_DRAM_BANKS 2
#define PHYS_SDRAM_1 OMAP34XX_SDRC_CS0
#define PHYS_SDRAM_1_SIZE (32 <<  20)
#define PHYS_SDRAM_2 OMAP34XX_SDRC_CS1
[...]
/* USB configuration */
#define CONFIG_MUSB_UDC 1
#define CONFIG_USB_OMAP3 1
#define CONFIG_TWL4030_USB 1
[...]
```

```
/* Available commands and features */
#define CONFIG_CMD_CACHE
#define CONFIG_CMD_EXT2
#define CONFIG_CMD_FAT
#define CONFIG_CMD_I2C
#define CONFIG_CMD_MMC
#define CONFIG_CMD_NAND
#define CONFIG_CMD_NET
#define CONFIG_CMD_DHCP
#define CONFIG_CMD_PING
#define CONFIG_CMD_NFS
#define CONFIG_CMD_MTDPARTS
[...]
```

Configuring and compiling U-Boot

- U-Boot must be configured before being compiled
  - `make BOARDNAME_config`
  - Where `BOARDNAME` is the name of the board, as visible in the `boards.cfg` file (first column).
- 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, there may be other specialized images: `u-boot.img`, `u-boot.kwb`, `MLO`, etc.

Witekio
EMBEDDING SUCCESS

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 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 to write to the flash memory remotely, without any system running on the board. It also allows to rescue a board if the bootloader doesn't work.

Witekio
EMBEDDING SUCCESS

U-boot prompt

- Connect the target to the host through a serial console
- Power-up the board. On the serial console, you will see something like:

```
U-Boot 2013.04 (May 29 2013 - 10:30:21)

OMAP36XX/37XX-GP ES1.2, CPU-OPP2, L3-165MHz, Max CPU Clock 1 Ghz
IGEPv2 + LPDDR/NAND
I2C:   ready
DRAM:  512 MiB
NAND:  512 MiB
MMC:   OMAP SD/MMC: 0

Die ID #255000029ff80000168580212029011
Net:   smc911x-0
U-Boot #
```

- The U-Boot shell offers a set of commands. We will study the most important ones, see the documentation for a complete reference or the `help` command.

Witekio
EMBEDDING SUCCESS

Information commands

Flash information (NOR and SPI flash)

```
U-Boot> flinfo
DataFlash:AT45DB021
Nb pages: 1024
Page Size: 264
Size= 270336 bytes
Logical address: 0xC0000000
Area 0: C0000000 to C0001FFF (RO) Bootstrap
Area 1: C0002000 to C0003FFF Environment
Area 2: C0004000 to C0041FFF (RO) U-Boot
```

NAND flash information

```
U-Boot> nand info
Device 0: nand0, sector size 128 KiB
  Page size      2048 b
  OOB size         64 b
  Erase size   131072 b
```

Version details

```
U-Boot> version
U-Boot 2013.04 (May 29 2013 - 10:30:21)
```

Witekio
EMBEDDING SUCCESS

Important commands (1)

- The exact set of commands depends on the U-Boot configuration
- `help` and `help command`
- `boot`, runs the default boot command, stored in `bootcmd`
- `bootm <address>`, starts a kernel image loaded at the given address in RAM
- `ext2load`, loads a file from an ext2 filesystem to RAM
  - And also `ext2ls` to list files, `ext2info` for information
- `fatload`, loads a file from a FAT filesystem to RAM
  - And also `fatls` and `fatinfo`
- `tftp`, loads a file from the network to RAM
- `ping`, to test the network

Witekio
EMBEDDING SUCCESS

Important commands (2)

- `loadb`, `loads`, `loady`, load a file from the serial line to RAM
- `usb`, to initialize and control the USB subsystem, mainly used for USB storage devices such as USB keys
- `mmc`, to initialize and control the MMC subsystem, used for SD and microSD cards
- `nand`, to erase, read and write contents to NAND flash
- `erase`, `protect`, `cp`, to erase, modify protection and write to NOR flash
- `md`, displays memory contents. Can be useful to check the contents loaded in memory, or to look at hardware registers.
- `mm`, modifies memory contents. Can be useful to modify directly hardware registers, for testing purposes.

Environment variables commands (1)

- U-Boot can be configured through environment variables, which affect the behavior of the different commands.
- Environment variables are loaded from flash to RAM at U-Boot startup, can be modified and saved back to flash for persistence
- There is a dedicated location in flash (or in MMC storage) to store the U-Boot environment, defined in the board configuration file

Environment variables commands (2)

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, only in RAM
- `editenv <variable-name>`
  Edits the value of a variable, only in RAM
- `saveenv`
  Saves the current state of the environment to flash

Witekio
EMBEDDING SUCCESS

Environment variables commands - Example

```
u-boot # printenv
baudrate=19200
ethaddr=00:40:95:36:35:33
netmask=255.255.255.0
ipaddr=10.0.0.11
serverip=10.0.0.1
stdin=serial
stdout=serial
stderr=serial
u-boot # printenv serverip
serverip=10.0.0.1
u-boot # setenv serverip 10.0.0.100
u-boot # saveenv
```

Witekio
EMBEDDING SUCCESS

Important U-Boot env variables

- `bootcmd`, contains the command that U-Boot will automatically execute at boot time after a configurable delay, 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
- `bootdelay`, the delay in seconds before which U-Boot runs `bootcmd`
- `autostart`, if yes, U-Boot starts automatically an image that has been loaded into memory

Witekio
EMBEDDING SUCCESS

Scripts in environment variables

- Environment variables can contain small scripts, to execute several commands and test the results of commands.
  - Useful to automate booting or upgrade processes
  - Several commands can be chained using the `;` operator
  - Tests can be done using
    `if command ; then ... ; else ... ; fi`
  - Scripts are executed using `run <variable-name>`
  - You can reference other variables using `${variable-name}`
- Example
  - `setenv mmc-boot 'if fatload mmc 0 80000000 boot.ini; then source; else if fatload mmc 0 80000000 uImage; then run mmc-bootargs; bootm; fi; fi'`

Witekio
EMBEDDING SUCCESS

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 if the target has an Ethernet connection, and U-Boot contains a driver for the Ethernet chip. 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

TFTP

- Network transfer from the development workstation to U-Boot on the target takes place through TFTP
  - *Trivial File Transfer Protocol*
  - Somewhat similar to FTP, but without authentication and over UDP
- A TFTP server is needed on the development workstation
  - `sudo apt-get install tftpd-hpa`
  - All files in `/var/lib/tftpboot` are then visible through TFTP
  - A TFTP client is available in the `tftp-hpa` package, for testing
- A TFTP client is integrated into U-Boot
  - Configure the `ipaddr` and `serverip` environment variables
  - Use `tftp <address> <filename>` to load a file

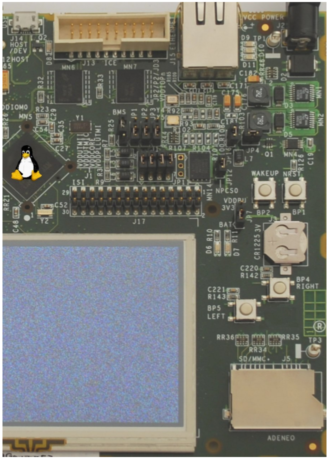
U-boot mkimage

- The kernel image that U-Boot loads and boots must be prepared, so that a U-Boot specific header is added in front of the image
  - This header gives details such as the image size, the expected load address, the compression type, etc.
- This is done with a tool that comes in U-Boot, `mkimage`
- Debian / Ubuntu: just install the `u-boot-tools` package.
- Or, compile it by yourself: simply configure U-Boot for any board of any architecture and compile it. Then install `mkimage`:
  `cp tools/mkimage /usr/local/bin/`
- The special target `uImage` of the kernel Makefile can then be used to generate a kernel image suitable for U-Boot.

More On Filesystems

More On Filesystems

Free Electrons, Witekio



More On Filesystems

Reminder

Filesystems (1)

- Filesystems are used to organize data in directories and files on storage devices or on the network. The directories and files are organized as a hierarchy
- In Unix systems, applications and users see a **single global hierarchy** of files and directories, which can be composed of several filesystems.
- Filesystems are **mounted** in a specific location in this hierarchy of directories
  - When a filesystem is mounted in a directory (called *mount point*), the contents of this directory reflects the contents of the storage device
  - When the filesystem is unmounted, the *mount point* is empty again.
- This allows applications to access files and directories easily, regardless of their exact storage location

Witekio
EMBEDDING SUCCESS

Filesystems (2)

- Create a mount point, which is just a directory

```
$ mkdir /mnt/usbkey
```

- It is empty

```
$ ls /mnt/usbkey
$
```

- Mount a storage device in this mount point

```
$ mount -t vfat /dev/sda1 /mnt/usbkey
$
```

- You can access the contents of the USB key

```
$ ls /mnt/usbkey
docs prog.c picture.png movie.avi
$
```

Witekio
EMBEDDING SUCCESS

mount / umount

- `mount` allows to mount filesystems
  - `mount -t type device mountpoint`
  - `type` is the type of filesystem
  - `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 filesystem

- 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 filesystem is available, the kernel panics

```
Please append a correct "root=" boot option
Kernel panic - not syncing: VFS: Unable to mount root fs on unknown block(0,0)
```

Location of the root filesystem

- It can be mounted from different locations
  - From the partition of a hard disk
  - From the partition of a USB key
  - From the partition of an SD card
  - From the partition of a NAND flash chip or similar type of storage device
  - From the network, using the NFS protocol
  - From memory, using a pre-loaded filesystem (by the bootloader)
  - etc.
- It is up to the system designer to choose the configuration for the system, and configure the kernel behaviour with `root=`

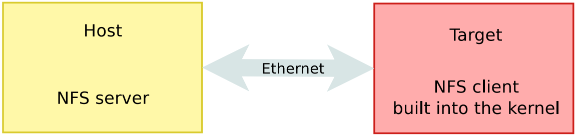
Mounting rootfs from storage devices

- 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 (1)

Once networking works, your root filesystem could be a directory on your GNU/Linux development host, exported by NFS (Network File System). This is very convenient for system development:

- Makes it very easy to update files on the root filesystem, without rebooting. Much faster than through the serial port.
- Can have a big root filesystem even if you don't have support for internal or external storage yet.
- The root filesystem can be huge. You can even build native compiler tools and build all the tools you need on the target itself (better to cross-compile though).



Witekio
EMBEDDING SUCCESS

Mounting rootfs over the network (2)

On the development workstation side, a NFS server is needed

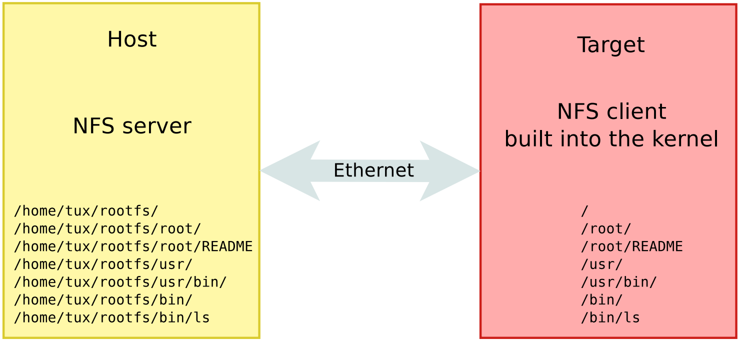
- Install an NFS server (example: Debian, Ubuntu)
  ```
  sudo apt-get install nfs-kernel-server
  ```
- Add the exported directory to your `/etc/exports` file:
  ```
  /home/tux/rootfs 192.168.1.111(rw,no_root_squash,
  no_subtree_check)
  ```
  - `192.168.1.111` is the client IP address
  - `rw,no_root_squash,no_subtree_check` 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
  sudo systemctl restart nfs-kernel-server.service
  ```

Witekio
EMBEDDING SUCCESS

Mounting rootfs over the network (3)

- On the target system
- The kernel must be compiled with
  - `CONFIG_NFS_FS=y` (NFS 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)

Mounting rootfs over the network (4)



More On Filesystems

Root Filesystem

Root filesystem organization

- The organization of a Linux root filesystem in terms of directories is well-defined by the **Filesystem Hierarchy Standard**
- `https://refspecs.linuxfoundation.org/fhs.shtml`
- Most Linux systems conform to this specification
  - Applications expect this organization
  - It makes it easier for developers and users as the filesystem organization is similar in all systems

Important directories (1)

/bin	Basic programs
/boot	Kernel image (only when the kernel is loaded from a filesystem, not common on non-x86 architectures)
/dev	Device files (covered later)
/etc	System-wide configuration
/home	Directory for the users home directories
/lib	Basic libraries
/media	Mount points for removable media
/mnt	Mount points for static media
/proc	Mount point for the proc virtual filesystem

Witekio
EMBEDDING SUCCESS

Important directories (2)

/root	Home directory of the `root` user
/sbin	Basic system programs
/sys	Mount point of the sysfs virtual filesystem
/tmp	Temporary files
/usr	/usr/bin Non-basic programs /usr/lib Non-basic libraries /usr/sbin Non-basic system programs
/var	Variable data files. This includes spool directories and files, administrative and logging data, and transient and temporary files

Witekio
EMBEDDING SUCCESS

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 was down, some binaries and libraries are stored in `/bin`, `/sbin` and `/lib`
- `/bin` and `/sbin` contain programs like `ls`, `ifconfig`, `cp`, `bash`, etc.
- `/lib` contains the C library and sometimes a few other basic libraries
- All other programs and libraries are in `/usr`

proc virtual filesystem

- The `proc` virtual filesystem exists since the beginning of Linux
- It allows
  - The kernel to expose statistics about running processes in the system
  - The user to 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`
- `Documentation/filesystems/proc.rst` in the kernel sources
- `man proc`

Witekio EMBEDDING SUCCESS

proc contents

- One directory for each running process in the system
  - `/proc/<pid>`
  - `cat /proc/3840/cmdline`
  - 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 to adjust kernel parameters
  - They are called *sysctl*. See `Documentation/admin-guide/sysctl` in kernel sources.
  - Example
    `echo 3 > /proc/sys/vm/drop_caches`

Witekio
EMBEDDING SUCCESS

sysfs filesystem

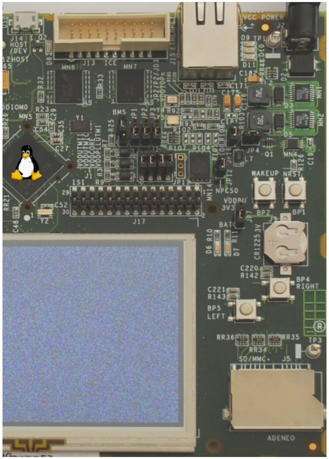
- The `sysfs` filesystem is a feature integrated in the 2.6 Linux kernel
- It allows to represent in user space the vision that the kernel has of the buses, devices and drivers in the system
- It is useful for various user space applications that need to list and query the available hardware, for example `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
  ```

-
  ```
  $ ls /sys/
  block bus class dev devices firmware
  fs kernel module power
  ```

The Linux Kernel

The Linux Kernel

Free Electrons, Witekio



The Linux Kernel

Reminder

History

- The Linux kernel is one component of a system, which also requires libraries and applications to provide features to end users.
- The Linux kernel was created as a hobby in 1991 by a Finnish student, Linus Torvalds.
  - Linux quickly started to be used as the kernel for free software operating systems
- Linus Torvalds has been able to create a large and dynamic developer and user community around Linux.
- Nowadays, more than one thousand people contribute to each kernel release, individuals or companies big and small.

Linux license

- The whole Linux sources are Free Software released under the GNU General Public License version 2 (GPL v2).
- For the Linux kernel, this basically implies that:
  - When you receive or buy a device with Linux on it, you should receive the Linux sources, with the right to study, modify and redistribute them.
  - When you produce Linux based devices, you must release the sources to the recipient, with the same rights, with no restriction.

Witekio
EMBEDDING SUCCESS

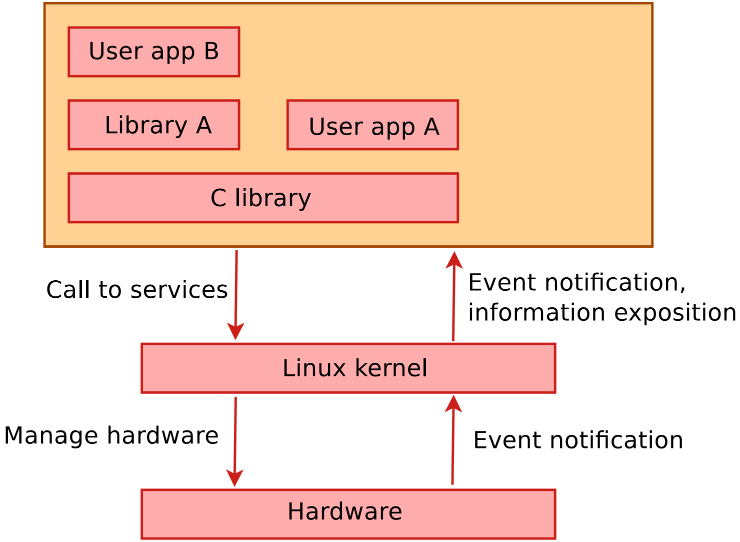
Supported hardware architectures

- See the `arch/` directory in the kernel sources
- Minimum: 32 bit processors, with or without MMU, and `gcc` support
- 32 bit architectures (`arch/` subdirectories)
  Examples: `arm`, `avr32`, `blackfin`, `c6x`, `m68k`, `microblaze`, `mips`, `score`, `sparc`, `um`
- 64 bit architectures:
  Examples: `alpha`, `arm64`, `ia64`, `tile`
- 32/64 bit architectures
  Examples: `powerpc`, `x86`, `sh`, `sparc`
- Find details in kernel sources: `arch/<arch>/Kconfig`, `arch/<arch>/README`, or `Documentation/<arch>/`

Linux kernel key features

- Portability and hardware support. Runs on most architectures.
- Scalability. Can run on super computers as well as on tiny devices (4 MB of RAM is enough).
- Compliance to standards and interoperability.
- Exhaustive networking support.
- Security. It can't hide its flaws. Its code is reviewed by many experts.
- Stability and reliability.
- Modularity. Can include only what a system needs even at run time.
- Easy to program. You can learn from existing code. Many useful resources on the net.

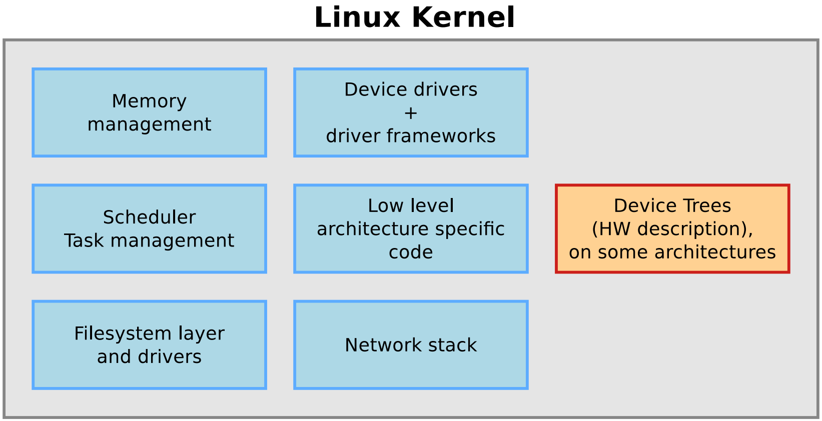
Linux kernel in the system



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.

Inside the Linux kernel


Linux Kernel
Memory management
Device drivers + driver frameworks
Scheduler Task management
Low level architecture specific code
Device Trees (HW description), on some architectures
Filesystem layer and drivers
Network stack

🟦	Implemented mainly in C, a little bit of assembly.
🟧	Written in a Device Tree specific language.

User Interfaces

System calls

- The main interface between the kernel and user space is a set of system calls
- About 300 system calls that provide the main kernel services:
  - File and device operations, networking operations, inter-process communication, process management, memory mapping, timers, threads, synchronization primitives, etc.
- This interface is stable over time: only new system calls can be added by the kernel developers
- This system call interface is wrapped by the C library, and user space applications usually never call a system call directly but rather use the corresponding C library function.

Virtual filesystems

- Linux makes system and kernel information available in user space through **virtual/pseudo filesystems**
- Virtual 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 most important virtual filesystems are:
  - `proc`, usually mounted on `/proc`:
    Operating system related information (processes, memory, network ...).
  - `sysfs`, usually mounted on `/sys`:
    Representation of the system as a tree of buses and devices. Related hardware and drivers information.
  - `devfs`, usually mounted on `/dev`:
    Device nodes (files).
  - `tmpfs`, usually used for `/tmp`:
    RAM based filesystem (volatile).
  - `debugfs`, usually mounted on `/sys/kernel/debug`:
    Kernel debug information and control.

Kernel Sources

Location of kernel sources

- The official versions of the Linux kernel, as released by Linus Torvalds, are available at `http://www.kernel.org`
  - These versions follow the development model of the kernel
  - However, they may not contain the latest development from a specific area yet. Some features in development might not be ready for mainline inclusion yet.
- Many chip vendors supply their own kernel sources
  - Focusing on hardware support first
  - Can have a very important delta with mainline Linux
  - Useful only when mainline hasn't caught up yet.
- Many kernel sub-communities maintain their own kernel, with usually newer but less stable features
  - Architecture communities (ARM, MIPS, PowerPC, etc.), device drivers communities (I2C, SPI, USB, PCI, network, etc.), other communities (real-time, etc.)
  - No official releases, only development trees are available.

Witekio
EMBEDDING SUCCESS

Getting Linux sources

- The kernel sources are available from `http://kernel.org/pub/linux/kernel` as **full tarballs** (complete kernel sources) and **patches** (differences between two kernel versions).
- However, more and more people use the `git` version control system. Absolutely needed for kernel development!
  - Fetch the entire kernel sources and history
    `git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git`
  - Create a branch that starts at a specific stable version
    `git checkout -b <name-of-branch> v3.11`
  - Web interface available at `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/`.
  - Read more about Git at `http://git-scm.com/`

Linux kernel size (1)

- Linux 3.10 sources:
  Raw size: 573 MB (43,000 files, approx 15,800,000 lines)
  `gzip` compressed tar archive: 105 MB
  `bzip2` compressed tar archive: 83 MB (better)
  `xz` compressed tar archive: 69 MB (best)
- Minimum Linux 3.17 compiled kernel size, booting on the ARM Versatile board (hard drive on PCI, ext2 filesystem, ELF executable support, framebuffer console and input devices): 876 KB (compressed), 2.3 MB (raw)
- Why are these sources so big?
  Because they include thousands of device drivers, many network protocols, support many architectures and filesystems...
- The Linux core (scheduler, memory management...) is pretty small!

Witekio
EMBEDDING SUCCESS

Linux kernel size (2)

As of kernel version 3.10.

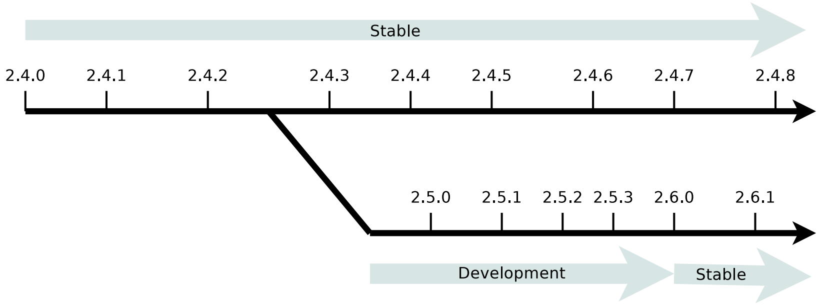
- `drivers/`: 49.4%
- `arch/`: 21.9%
- `fs/`: 6.0%
- `include/`: 4.7%
- `sound/`: 4.4%
- `Documentation/`: 4.0%
- `net/`: 3.9%
- `firmware/`: 1.0%
- `kernel/`: 1.0%
- `tools/`: 0.9%
- `scripts/`: 0.5%
- `mm/`: 0.5%
- `crypto/`: 0.4%
- `security/`: 0.4%
- `lib/`: 0.4%
- `block/`: 0.2%
- ...

Kernel Development Process

Until 2.6 (1)

- One stable major branch every 2 or 3 years
  - Identified by an even middle number
  - Examples: `1.0.x, 2.0.x, 2.2.x, 2.4.x`
- One development branch to integrate new functionalities and major changes
  - Identified by an odd middle number
  - Examples: `2.1.x, 2.3.x, 2.5.x`
  - After some time, a development version becomes the new base version for the stable branch
- Minor releases once in while: `2.2.23, 2.5.12`, etc.

Until 2.6 (2)


Stable
2.4.0
2.4.1
2.4.2
2.4.3
2.4.4
2.4.5
2.4.6
2.4.7
2.4.8
2.5.0
2.5.1
2.5.2
2.5.3
2.6.0
2.6.1
Development
Stable

Changes since Linux 2.6

- Since `2.6.0`, kernel developers have been able to introduce lots of new features one by one on a steady pace, without having to make disruptive changes to existing subsystems.
- Since then, there has been no need to create a new development branch massively breaking compatibility with the stable branch.
- Thanks to this, **more features are released to users at a faster pace**.

Witekio
EMBEDDING SUCCESS

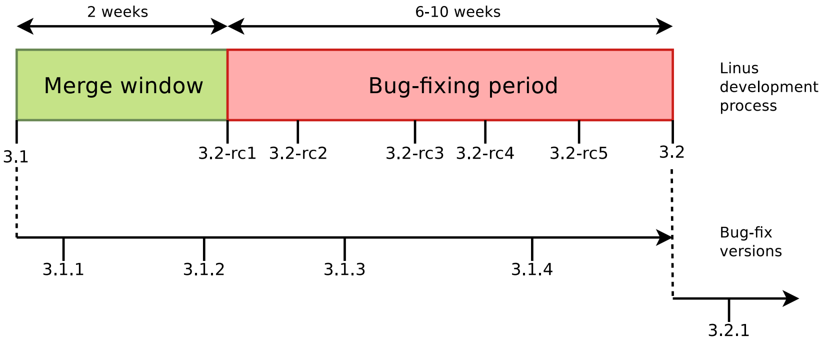
3.x stable branch

- From 2003 to 2011, the official kernel versions were named `2.6.x`.
- Linux `3.0` was released in July 2011
- This is only a change to the numbering scheme
  - Official kernel versions are now named `3.x` (`3.0, 3.1, 3.2`, etc.)
  - Stabilized versions are named `3.x.y` (`3.0.2, 3.4.3`, etc.)
  - It effectively only removes a digit compared to the previous numbering scheme

Witekio
EMBEDDING SUCCESS

New development model

Using merge and bug fixing windows



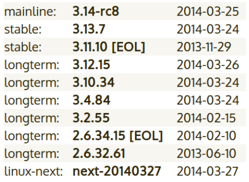
New development model - Details

- After the release of a `3.x` version (for example), a two-weeks merge window opens, during which major additions are merged.
- The merge window is closed by the release of test version `3.(x+1)-rc1`
- The bug fixing period opens, for 6 to 10 weeks.
- At regular intervals during the bug fixing period, `3.(x+1)-rcY` test versions are released.
- When considered sufficiently stable, kernel `3.(x+1)` is released, and the process starts again.

Witekio
EMBEDDING SUCCESS

More stability for the kernel source tree

- Issue: bug and security fixes only released for most recent stable kernel versions.
- Some people need to have a recent kernel, but with long term support for security updates.
- You could get long term support from a commercial embedded Linux provider.
- You could reuse sources for the kernel used in Ubuntu Long Term Support releases (5 years of free security updates).
- The `http://kernel.org` front page shows which versions will be supported for some time (up to 2 or 3 years), and which ones won't be supported any more ("EOL: End Of Life")



mainline:	**3.14-rc8**	2014-03-25
stable:	**3.13.7**	2014-03-24
stable:	**3.11.10 [EOL]**	2013-11-29
longterm:	**3.12.15**	2014-03-26
longterm:	**3.10.34**	2014-03-24
longterm:	**3.4.84**	2014-03-24
longterm:	**3.2.55**	2014-02-15
longterm:	**2.6.34.15 [EOL]**	2014-02-10
longterm:	**2.6.32.61**	2013-06-10
linux-next:	**next-20140327**	2014-03-27

What's new in each Linux release?

- The official list of changes for each Linux release is just a huge list of individual patches!

```
commit aa6e52a35d388e730f4df0ec2ec48294590cc459
Author: Thomas Petazzoni <thomas.petazzoni@free-electrons.com>
Date:   Wed Jul 13 11:29:17 2011 +0200

    at91: at91-ohci: support overcurrent notification

    Several USB power switches (AIC1526 or MIC2026) have a digital output
    that is used to notify that an overcurrent situation is taking
    place. This digital outputs are typically connected to GPIO inputs of
    the processor and can be used to be notified of these overcurrent
    situations.

    Therefore, we add a new overcurrent_pin[] array in the at91_usbh_data
    structure so that boards can tell the AT91 OHCI driver which pins are
    used for the overcurrent notification, and an overcurrent_supported
    boolean to tell the driver whether overcurrent is supported or not.

    The code has been largely borrowed from ohci-da8xx.c and
    ohci-s3c2410.c.

    Signed-off-by: Thomas Petazzoni <thomas.petazzoni@free-electrons.com>
    Signed-off-by: Nicolas Ferre <nicolas.ferre@atmel.com>
```

  - Very difficult to find out the key changes and to get the global picture out of individual changes.
- Fortunately, there are some useful resources available
  - `http://wiki.kernelnewbies.org/LinuxChanges`
  - `http://lwn.net`
  - `http://linuxfr.org`, for French readers

Legal Issues

Witekio
EMBEDDING SUCCESS

Linux kernel licensing constraints

- The Linux kernel is licensed under the GNU General Public License version 2
  - This license gives you the right to use, study, modify and share the software freely
- However, when the software is redistributed, either modified or unmodified, the GPL requires that you redistribute the software under the same license, with the source code
  - If modifications are made to the Linux kernel (for example to adapt it to your hardware), it is a derivative work of the kernel, and therefore must be released under GPLv2
  - The validity of the GPL on this point has already been verified in courts
- However, you're only required to do so
  - At the time the device starts to be distributed
  - To your customers, not to the entire world

Proprietary code and the kernel

- It is illegal to distribute a binary kernel that includes statically compiled proprietary drivers
- The kernel modules are a gray area: are they derived works of the kernel or not?
  - The general opinion of the kernel community is that proprietary drivers are bad: `https://lkml.org/lkml/2006/12/13/370`
  - From a legal point of view, each driver is probably a different case
  - Is it really useful to keep your drivers secret?
- There are some examples of proprietary drivers, like the Nvidia graphics drivers
  - They use a wrapper between the driver and the kernel
  - Unclear whether it makes it legal or not

Advantages of GPL drivers

- You don't have to write your driver from scratch. You can reuse code from similar free software drivers.
- You could get free community contributions, support, code review and testing, though this generally only happens with code submitted for the mainline kernel.
- Your drivers can be freely and easily shipped by others (for example by Linux distributions or embedded Linux build systems).
- Pre-compiled drivers work with only one kernel version and one specific configuration, making life difficult for users who want to change the kernel version.
- Legal certainty, you are sure that a GPL driver is fine from a legal point of view.

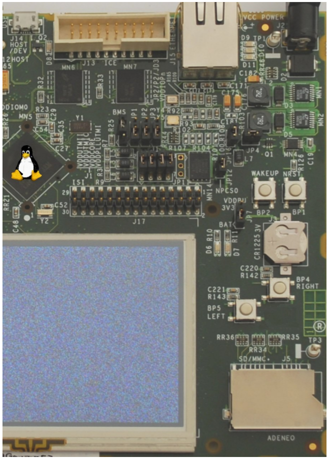
Advantages of in-tree kernel drivers

- Once your sources are accepted in the mainline tree, they are maintained by people making changes.
- Near cost-free maintenance, security fixes and improvements.
- Easy access to your sources by users.
- Many more people reviewing your code.

Kernel Source Code

Kernel Source Code

Free Electrons, Witekio



Kernel Source Code

Linux Code and Device Drivers

Programming language

- Implemented in C like all Unix systems. (C was created to implement the first Unix systems)
- A little Assembly is used too:
  - CPU and machine initialization, exceptions
  - Critical library routines.
- No C++ used, see `https://lore.kernel.org/lkml/Pine.LNX.4.58.0401192241080.2311@home.osdl.org/`
- All the code compiled with gcc
  - Many gcc specific extensions used in the kernel code, any ANSI C compiler will not compile the kernel
  - A few alternate compilers are supported (Intel and Marvell)
  - See `http://gcc.gnu.org/onlinedocs/gcc-4.9.0/gcc/C-Extensions.html`

No C library

- The kernel has to be standalone and can't use user space code.
- User space is implemented on top of kernel services, not the opposite.
- Kernel code has to supply its own library implementations (string utilities, cryptography, uncompression ...)
- So, you can't use standard C library functions in kernel code. (`printf()`, `memset()`, `malloc()`,...).
- Fortunately, the kernel provides similar C functions for your convenience, like `printk()`, `memset()`, `kmalloc()`, ...

Witekio
EMBEDDING SUCCESS

Portability

- The Linux kernel code is designed to be portable
- All code outside `arch/` should be portable
- To this aim, the kernel provides macros and functions to abstract the architecture specific details
  - Endianness
    - `cpu_to_be32()`
    - `cpu_to_le32()`
    - `be32_to_cpu()`
    - `le32_to_cpu()`
  - I/O memory access
  - Memory barriers to provide ordering guarantees if needed
  - DMA API to flush and invalidate caches if needed

No floating point computation

- Never use floating point numbers in kernel code. Your code may be run on a processor without a floating point unit (like on certain ARM CPUs).
- Don't be confused with floating point related configuration options
  - They are related to the emulation of floating point operation performed by the user space applications, triggering an exception into the kernel.
  - Using soft-float, i.e. emulation in user space, is however recommended for performance reasons.

No stable Linux internal API

- The internal kernel API to implement kernel code can undergo changes between two releases.
- In-tree drivers are updated by the developer proposing the API change: works great for mainline code.
- An out-of-tree driver compiled for a given version may no longer compile or work on a more recent one.
- See `Documentation/process/stable-api-nonsense.rst` in kernel sources for reasons why.
- Of course, the kernel to userspace API does not change (system calls, `/proc`, `/sys`), as it would break existing programs.

Witekio
EMBEDDING SUCCESS

Kernel memory constraints

- No memory protection
- Accessing illegal memory locations result in (often fatal) kernel oopses.
- Fixed size stack (8 or 4 KB). Unlike in user space, there's no way to make it grow.
- Kernel memory can't be swapped out (for the same reasons).

User space device drivers 1/3

- In some cases, it is possible to implement device drivers in user space!
- Can be used when
    - The kernel provides a mechanism that allows userspace applications to directly access the hardware.
    - There is no need to leverage an existing kernel subsystem such as the networking stack or filesystems.
    - There is no need for the kernel to act as a “multiplexer” for the device: only one application accesses the device.

User space device drivers 2/3

- Possibilities for userspace device drivers:
  - USB with *libusb*, `https://libusb.info/`
  - SPI with *spidev*, `Documentation/spi/spidev`
  - I2C with *i2cdev*, `Documentation/i2c/dev-interface`
  - Memory-mapped devices with *UIO*, including interrupt handling, `Documentation/driver-api/uio-howto.rst`
- Certain classes of devices (printers, scanners, 2D/3D graphics acceleration) are typically handled partly in kernel space, partly in user space.

User space device drivers 3/3

- Advantages
  - No need for kernel coding skills. Easier to reuse code between devices.
  - Drivers can be written in any language, even Perl!
  - Drivers can be kept proprietary.
  - Driver code can be killed and debugged. Cannot crash the kernel.
  - Can be swapped out (kernel code cannot be).
  - Can use floating-point computation.
  - Less in-kernel complexity.
  - Potentially higher performance, especially for memory-mapped devices, thanks to the avoidance of system calls.
- Drawbacks
  - Less straightforward to handle interrupts.
  - Increased interrupt latency vs. kernel code.

Linux Sources Layout

Linux sources structure 1/5

- `arch/<ARCH>`
  - Architecture specific code
  - `arch/<ARCH>/mach-<machine>`, machine/board specific code
  - `arch/<ARCH>/include/asm`, architecture-specific headers
  - `arch/<ARCH>/boot/dts`, Device Tree source files, for some architectures
- `block/`
  - Block layer core
- `COPYING`
  - Linux copying conditions (GNU GPL)
- `CREDITS`
  - Linux main contributors
- `crypto/`
  - Cryptographic libraries

Linux sources structure 2/5

- `Documentation/`
  - Kernel documentation. Don't miss it!
- `drivers/`
  - All device drivers except sound ones (usb, pci...)
- `firmware/`
  - Legacy: firmware images extracted from old drivers
- `fs/`
  - Filesystems (`fs/ext3/`, etc.)
- `include/`
  - Kernel headers
- `include/linux/`
  - Linux kernel core headers
- `include/uapi/`
  - User space API headers
- `init/`
  - Linux initialization (including `main.c`)
- `ipc/`
  - Code used for process communication

Witekio
EMBEDDING SUCCESS

Linux sources structure 3/5

- Kbuild
  - Part of the kernel build system
- Kconfig
  - Top level description file for configuration parameters
- kernel/
  - Linux kernel core (very small!)
- lib/
  - Misc library routines (zlib, crc32...)
- MAINTAINERS
  - Maintainers of each kernel part. Very useful!
- Makefile
  - Top Linux Makefile (sets arch and version)
- mm/
  - Memory management code (small too!)

Witekio
EMBEDDING SUCCESS

Linux sources structure 4/5

- `net/`
  - Network support code (not drivers)
- `README`
  - Overview and building instructions
- `REPORTING-BUGS`
  - Bug report instructions
- `samples/`
  - Sample code (markers, kprobes, kobjects...)
- `scripts/`
  - Scripts for internal or external use
- `security/`
  - Security model implementations (SELinux...)
- `sound/`
  - Sound support code and drivers
- `tools/`
  - Code for various user space tools (mostly C)

Linux sources structure 5/5

- `usr/`
  - Code to generate an initramfs cpio archive
- `virt/`
  - Virtualization support (KVM)

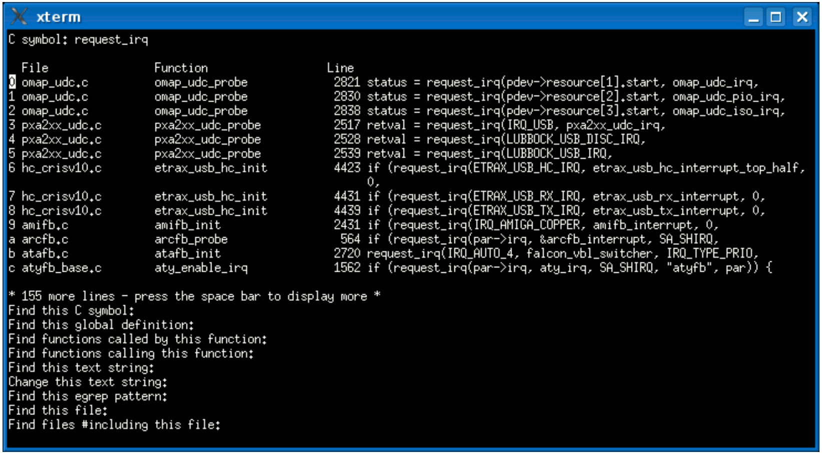
Kernel Source Code

Kernel Source Management Tools

Cscope

- Tool to browse source code (mainly C, but also C++ or Java)
- Supports huge projects like the Linux kernel. Typically takes less than 1 min. to index the whole Linux sources.
- In Linux kernel sources, two ways of running it:
  - `cscope -Rk`
    All files for all architectures at once
  - `make cscope`
    `cscope -d cscope.out`
    Only files for your current architecture
- Allows searching for a symbol, a definition, functions, strings, files, etc.
- Integration with editors like `vim` and `emacs`.
- Dedicated graphical front-end: `KScope`
- `http://cscope.sourceforge.net/`

Cscope screenshot



```
C symbol: request_irq

  File                Function                 Line
0 omap_udc.c          omap_udc_probe           2821 status = request_irq(pdev->resource[1].start, omap_udc_irq,
1 omap_udc.c          omap_udc_probe           2830 status = request_irq(pdev->resource[2].start, omap_udc_pio_irq,
2 omap_udc.c          omap_udc_probe           2838 status = request_irq(pdev->resource[3].start, omap_udc_iso_irq,
3 pxa2xx_udc.c        pxa2xx_udc_probe         2517 retval = request_irq(IRQ_USB, pxa2xx_udc_irq,
4 pxa2xx_udc.c        pxa2xx_udc_probe         2528 retval = request_irq(LUBBOCK_USB_DISC_IRQ,
5 pxa2xx_udc.c        pxa2xx_udc_probe         2539 retval = request_irq(LUBBOCK_USB_IRQ,
6 hc_crisv10.c        etrax_usb_hc_init        4423 if (request_irq(ETRAX_USB_HC_IRQ, etrax_usb_hc_interrupt_top_half,
                                                    0,
7 hc_crisv10.c        etrax_usb_hc_init        4431 if (request_irq(ETRAX_USB_RX_IRQ, etrax_usb_rx_interrupt, 0,
8 hc_crisv10.c        etrax_usb_hc_init        4439 if (request_irq(ETRAX_USB_TX_IRQ, etrax_usb_tx_interrupt, 0,
9 amifb.c             amifb_init               2431 if (request_irq(IRQ_AMIGA_COPPER, amifb_interrupt, 0,
a arcfb.c             arcfb_probe               564 if (request_irq(par->irq, &arcfb_interrupt, SA_SHIRQ,
b atafb.c             atafb_init               2720 request_irq(IRQ_AUTO_4, falcon_vbl_switcher, IRQ_TYPE_PRIO,
c atyfb_base.c        aty_enable_irq           1562 if (request_irq(par->irq, aty_irq, SA_SHIRQ, "atyfb", par)) {

* 155 more lines - press the space bar to display more *
Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

[Tab]: move the cursor between search results and commands

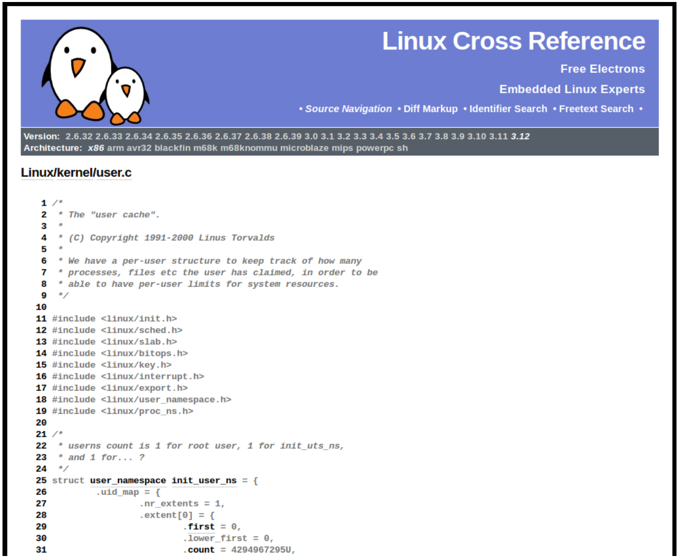
[Ctrl] [D]: exit cscope

Witekio
EMBEDDING SUCCESS

LXR: Linux Cross Reference

- Generic source indexing tool and code browser
- Web server based, very easy and fast to use
- Very easy to find the declaration, implementation or usage of symbols
- Supports C and C++
- Supports huge code projects such as the Linux kernel (431 MB of source code in version 3.0).
- Takes a little time and patience to setup (configuration, indexing, web server configuration)
- You don't need to set up LXR by yourself. Use our `http://lxr.free-electrons.com` server!
- `http://sourceforge.net/projects/lxr`

LXR screenshot


Linux Cross Reference
Free Electrons
Embedded Linux Experts
• Source Navigation • Diff Markup • Identifier Search • Freetext Search •
Version: 2.6.32 2.6.33 2.6.34 2.6.35 2.6.36 2.6.37 2.6.38 2.6.39 3.0 3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9 3.10 3.11 3.12
Architecture: x86 arm avr32 blackfin m68k m68knommu microblaze mips powerpc sh

Linux/kernel/user.c

```
/*
 * The "user cache".
 *
 * (C) Copyright 1991-2000 Linus Torvalds
 *
 * We have a per-user structure to keep track of how many
 * processes, files etc the user has claimed, in order to be
 * able to have per-user limits for system resources.
 */

#include <linux/init.h>
#include <linux/sched.h>
#include <linux/slab.h>
#include <linux/bitops.h>
#include <linux/key.h>
#include <linux/interrupt.h>
#include <linux/export.h>
#include <linux/user_namespace.h>
#include <linux/proc_ns.h>

/*
 * userns count is 1 for root user, 1 for init_uts_ns,
 * and 1 for... ?
 */
struct user_namespace init_user_ns = {
        .uid_map = {
                .nr_extents = 1,
                .extent[0] = {
                        .first = 0,
                        .lower_first = 0,
                        .count = 4294967295U,
```

Kernel Configuration

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.). Run `make help` to see all available targets.
- Example
  - `cd linux-3.6.x/`
  - `make <target>`

Kernel configuration (1)

- The kernel contains thousands of device drivers, filesystem drivers, network protocols and other configurable items
- Thousands of options are available, that are used to selectively compile parts of the kernel source code
- 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 your hardware (for device drivers, etc.)
  - On the capabilities you would like to give to your kernel (network capabilities, filesystems, real-time, etc.)

Kernel configuration (2)

- The configuration is stored in the `.config` file at the root of kernel sources
  - Simple text file, `key=value` style
- 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
- To modify a kernel in a GNU/Linux distribution: the configuration files are usually released in `/boot/`, together with kernel images: `/boot/config-3.2.0-31-generic`

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
  - This is not possible in the early boot procedure of the kernel, because no filesystem is available

Kernel option types

- There are different types of options
  - `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
  - `string` options, to specify string values

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 A that depends on option B is not visible until option B is enabled
  - `select` dependencies. In this case, with option A depending on option B, when option A is enabled, option B is automatically enabled
  - `make xconfig` allows to see all options, even the ones that cannot be selected because of missing dependencies. In this case, they are displayed in gray

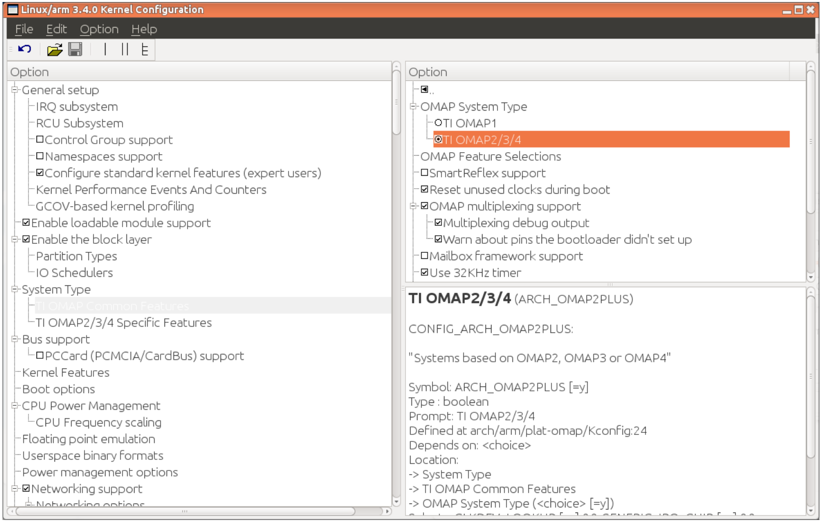
make xconfig

`make xconfig`

- The most common graphical interface to configure the kernel.
- Make sure you read
  `help -> introduction: useful options!`
- File browser: easier to load configuration files
- Search interface to look for parameters
- Required Debian / Ubuntu packages: `libqt4-dev g++` (`libqt3-mt-dev` for older kernel releases)

Witekio
EMBEDDING SUCCESS

make xconfig screenshot



- General setup
  - IRQ subsystem
  - RCU Subsystem
  - ☐ Control Group support
  - ☐ Namespaces support
  - ☑ Configure standard kernel features (expert users)
  - Kernel Performance Events And Counters
  - GCOV-based kernel profiling
- ☑ Enable loadable module support
- ☑ Enable the block layer
  - Partition Types
  - IO Schedulers
- System Type
  - TI OMAP Common Features
  - TI OMAP2/3/4 Specific Features
- Bus support
  - ☐ PCCard (PCMCIA/CardBus) support
- Kernel Features
- Boot options
- CPU Power Management
  - CPU Frequency scaling
- Floating point emulation
- Userspace binary formats
- Power management options
- ☑ Networking support
  - Networking options

Option

- ..
- OMAP System Type
  - ○ TI OMAP1
  - ◉ TI OMAP2/3/4
- OMAP Feature Selections
- ☐ SmartReflex support
- ☑ Reset unused clocks during boot
- ☑ OMAP multiplexing support
  - ☑ Multiplexing debug output
  - ☑ Warn about pins the bootloader didn't set up
- ☐ Mailbox framework support
- ☑ Use 32KHz timer

TI OMAP2/3/4 (ARCH_OMAP2PLUS)

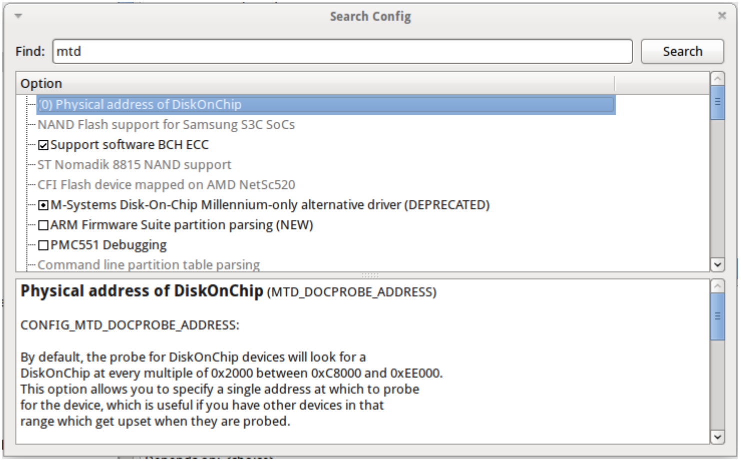
CONFIG_ARCH_OMAP2PLUS:

"Systems based on OMAP2, OMAP3 or OMAP4"

Symbol: ARCH_OMAP2PLUS [=y]
Type : boolean
Prompt: TI OMAP2/3/4
Defined at arch/arm/plat-omap/Kconfig:24
Depends on: <choice>
Location:
-> System Type
-> TI OMAP Common Features
-> OMAP System Type (<choice> [=y])

make xconfig search interface

Looks for a keyword in the parameter name. Allows to select or unselect found parameters.



mtd

(0) Physical address of DiskOnChip
NAND Flash support for Samsung S3C SoCs
☑ Support software BCH ECC
ST Nomadik 8815 NAND support
CFI Flash device mapped on AMD NetSc520
● M-Systems Disk-On-Chip Millennium-only alternative driver (DEPRECATED)
☐ ARM Firmware Suite partition parsing (NEW)
☐ PMC551 Debugging
Command line partition table parsing

Physical address of DiskOnChip (MTD_DOCPROBE_ADDRESS)

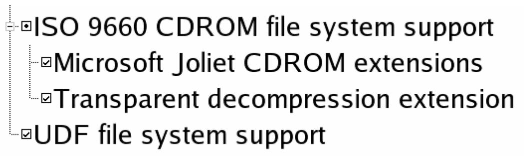
CONFIG_MTD_DOCPROBE_ADDRESS:

By default, the probe for DiskOnChip devices will look for a
DiskOnChip at every multiple of 0x2000 between 0xC8000 and 0xEE000.
This option allows you to specify a single address at which to probe
for the device, which is useful if you have other devices in that
range which get upset when they are probed.

Kernel configuration options

Compiled as a module (separate file)
`CONFIG_ISO9660_FS=m`

Driver options
`CONFIG_JOLIET=y`
`CONFIG_ZISOFS=y`



⊡ ISO 9660 CDROM file system support
- ☑ Microsoft Joliet CDROM extensions
- ☑ Transparent decompression extension

☑ UDF file system support

Compiled statically into the kernel
`CONFIG_UDF_FS=y`

Witekio
EMBEDDING SUCCESS

Corresponding .config file excerpt

Options are grouped by sections and are prefixed with `CONFIG_`.

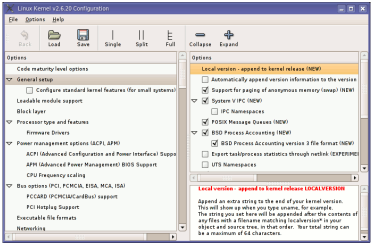
```
#
# CD-ROM/DVD Filesystems
#
CONFIG_ISO9660_FS=m
CONFIG_JOLIET=y
CONFIG_ZISOFS=y
CONFIG_UDF_FS=y
CONFIG_UDF_NLS=y

#
# DOS/FAT/NT Filesystems
#
# CONFIG_MSDOS_FS is not set
# CONFIG_VFAT_FS is not set
CONFIG_NTFS_FS=m
# CONFIG_NTFS_DEBUG is not set
CONFIG_NTFS_RW=y
```

Witekio
EMBEDDING SUCCESS

make gconfig

`make gconfig`

- *GTK* based graphical configuration interface. Functionality similar to that of make `xconfig`.
- Just lacking a search functionality.
- Required Debian packages: `libglade2-dev`


Linux Kernel v2.6.20 Configuration
File Options Help
Back Load Save Single Split Full Collapse Expand

Options
- Code maturity level options
- General setup
  - ☐ Configure standard kernel features (for small systems)
- Loadable module support
- Block layer
- Processor type and features
  - Firmware Drivers
- Power management options (ACPI, APM)
  - ACPI (Advanced Configuration and Power Interface) Suppo
  - APM (Advanced Power Management) BIOS Support
  - CPU Frequency scaling
- Bus options (PCI, PCMCIA, EISA, MCA, ISA)
  - PCCARD (PCMCIA/CardBus) support
  - PCI Hotplug Support
- Executable file formats
- Networking

Options
- Local version - append to kernel release (NEW)
- ☐ Automatically append version information to the version
- ✓ Support for paging of anonymous memory (swap) (NEW)
- ✓ System V IPC (NEW)
  - ☐ IPC Namespaces
- ✓ POSIX Message Queues (NEW)
- ✓ BSD Process Accounting (NEW)
  - ✓ BSD Process Accounting version 3 file format (NEW)
- ☐ Export task/process statistics through netlink (EXPERIME
- ☐ UTS Namespaces

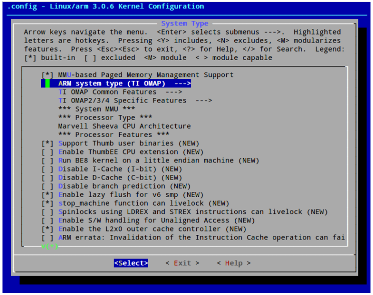
Local version - append to kernel release LOCALVERSION

Append an extra string to the end of your kernel version.
This will show up when you type uname, for example.
The string you set here will be appended after the contents of
any files with a filename matching localversion* in your
object and source tree, in that order. Your total string can
be a maximum of 64 characters.

make menuconfig

`make menuconfig`

- Useful when no graphics are available. Pretty convenient too!
- Same interface found in other tools: BusyBox, Buildroot...
- Required Debian packages: `libncurses-dev`

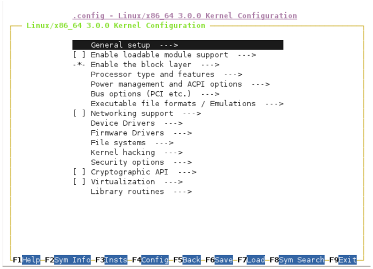


```
.config - Linux/arm 3.0.6 Kernel Configuration
                              System Type
  Arrow keys navigate the menu.  <Enter> selects submenus --->.  Highlighted
  letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes
  features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend:
  [*] built-in  [ ] excluded  <M> module  < > module capable

      [*] MMU-based Paged Memory Management Support
          ARM system type (TI OMAP)  --->
          TI OMAP Common Features  --->
          TI OMAP2/3/4 Specific Features  --->
          *** System MMU ***
          *** Processor Type ***
          Marvell Sheeva CPU Architecture
          *** Processor Features ***
      [*] Support Thumb user binaries (NEW)
      [ ] Enable ThumbEE CPU extension (NEW)
      [ ] Run BE8 kernel on a little endian machine (NEW)
      [ ] Disable I-Cache (I-bit) (NEW)
      [ ] Disable D-Cache (C-bit) (NEW)
      [ ] Disable branch prediction (NEW)
      [*] Enable lazy flush for v6 smp (NEW)
      [*] stop_machine function can livelock (NEW)
      [ ] Spinlocks using LDREX and STREX instructions can livelock (NEW)
      [ ] Enable S/W handling for Unaligned Access (NEW)
      [*] Enable the L2x0 outer cache controller (NEW)
      [ ] ARM errata: Invalidation of the Instruction Cache operation can fai
      v(+)

                    <Select>    < Exit >    < Help >
```

make nconfig

`make nconfig`

- A newer, similar text interface
- More user friendly (for example, easier to access help information).
- Required Debian packages: `libncurses-dev`



```
.config - Linux/x86_64 3.0.0 Kernel Configuration
 Linux/x86_64 3.0.0 Kernel Configuration

        General setup  --->
    [ ] Enable loadable module support  --->
    -*- Enable the block layer  --->
        Processor type and features  --->
        Power management and ACPI options  --->
        Bus options (PCI etc.)  --->
        Executable file formats / Emulations  --->
    [ ] Networking support  --->
        Device Drivers  --->
        Firmware Drivers  --->
        File systems  --->
        Kernel hacking  --->
        Security options  --->
    [ ] Cryptographic API  --->
    [ ] Virtualization  --->
        Library routines  --->

F1Help F2Sym Info F3Insts F4Config F5Back F6Save F7Load F8Sym Search F9Exit
```

make oldconfig

`make oldconfig`

- Needed very often!
- Useful to upgrade a `.config` file from an earlier kernel release
- Issues warnings for configuration parameters that no longer exist in the new kernel.
- Asks for values for new parameters (while `xconfig` and `menuconfig` silently set default values for new parameters).

If you edit a `.config` file by hand, it's strongly recommended to run `make oldconfig` afterwards!

Undoing configuration changes

A frequent problem:

- After changing several kernel configuration settings, your kernel no longer works.
- If you don't remember all the changes you made, you can get back to your previous configuration:
  ```
  $ cp .config.old .config
  ```
- All the configuration interfaces of the kernel (`xconfig`, `menuconfig`, `oldconfig`...) keep this `.config.old` backup copy.

Configuration per architecture

- The set of configuration options is architecture dependent
  - Some configuration options are very architecture-specific
  - Most of the configuration options (global kernel options, network subsystem, filesystems, most of the device drivers) are visible in all architectures.
- By default, the kernel build system assumes that the kernel is being built for the host architecture, i.e. native compilation
- The architecture is not defined inside the configuration, but at a higher level
- We will see later how to override this behaviour, to allow the configuration of kernels for a different architecture

Kernel Source Code

Kernel Native Compilation

Kernel compilation

- `make`
  - in the main kernel source directory
  - Remember to run multiple jobs in parallel if you have multiple CPU cores. Example: `make -j 4`
  - No need to run as root!
- Generates
  - `vmlinux`, the raw uncompressed kernel image, in ELF format, useful for debugging purposes, but cannot be booted
  - `arch/<arch>/boot/*Image`, the final, usually compressed, kernel image that can be booted
    - `bzImage` for x86, `zImage` for ARM, `vmImage.gz` for Blackfin, etc.
  - `arch/<arch>/boot/dts/*.dtb`, compiled Device Tree files (on some architectures)
  - All kernel modules, spread over the kernel source tree, as `.ko` files.

Kernel installation

- `make install`
  - Does the installation for the host system by default, so needs to be run as root. Generally not used when compiling for an embedded system, as it installs files on the development workstation.
- Installs
  - `/boot/vmlinuz-<version>`
    Compressed kernel image. Same as the one in `arch/<arch>/boot`
  - `/boot/System.map-<version>`
    Stores kernel symbol addresses
  - `/boot/config-<version>`
    Kernel configuration for this version
- Typically re-runs the bootloader configuration utility to take the new kernel into account.

Module installation

- `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`
    Module aliases for module loading utilities. Example line:
    `alias sound-service-?-0 snd_mixer_oss`
  - `modules.dep`
    Module dependencies
  - `modules.symbols`
    Tells which module a given symbol belongs to.

Kernel cleanup targets

- Clean-up generated files (to force re-compilation):
  `make clean`
- 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`

Kernel Source Code

Kernel Cross Compilation

Cross-compiling the kernel

When you compile a Linux kernel for another CPU architecture

- Much faster than compiling natively, when the target system is much slower than your GNU/Linux workstation.
- Much easier as development tools for your GNU/Linux workstation are much easier to find.
- 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-`

Specifying cross-compilation (1)

The CPU architecture and cross-compiler prefix are defined through the `ARCH` and `CROSS_COMPILE` variables in the toplevel Makefile.

- `ARCH` is the name of the architecture. It is defined by the name of the subdirectory in `arch/` in the kernel sources
  - Example: `arm` if you want to compile a kernel for the `arm` architecture.
- `CROSS_COMPILE` is the prefix of the cross compilation tools
  - Example: `arm-linux-` if your compiler is `arm-linux-gcc`

Witekio
EMBEDDING SUCCESS

Specifying cross-compilation (2)

Two solutions to define `ARCH` and `CROSS_COMPILE`:

- Pass `ARCH` and `CROSS_COMPILE` on the `make` command line:
  `make ARCH=arm CROSS_COMPILE=arm-linux- ...`
  Drawback: it is easy to forget to pass these variables when you run any `make` command, causing your build and configuration to be screwed up.
- Define `ARCH` and `CROSS_COMPILE` as environment variables:
  ```
  export ARCH=arm
  export CROSS_COMPILE=arm-linux-
  ```
  Drawback: it only works inside the current shell or terminal. You could put these settings in a file that you source every time you start working on the project. If you only work on a single architecture with always the same toolchain, you could even put these settings in your `~/.bashrc` file to make them permanent and visible from any terminal.

Witekio
EMBEDDING SUCCESS

Predefined configuration files

- Default configuration files available, per board or per-CPU family
  - They are stored in `arch/<arch>/configs/`, and are just minimal `.config` files
  - This is the most common way of configuring a kernel for embedded platforms
- Run `make help` to find if one is available for your platform
- To load a default configuration file, just run `make acme_defconfig`
  - This will overwrite your existing `.config` file!
- To create your own default configuration file
  - `make savedefconfig`, to create a minimal configuration file
  - `mv defconfig arch/<arch>/configs/myown_defconfig`

Configuring the kernel

- After loading a default configuration file, you can adjust the configuration to your needs with the normal `xconfig`, `gconfig` or `menuconfig` interfaces
- You can also start the configuration from scratch without loading a default configuration file
- As the architecture is different from your host architecture
  - Some options will be different from the native configuration (processor and architecture specific options, specific drivers, etc.)
  - Many options will be identical (filesystems, network protocols, architecture-independent drivers, etc.)
- Make sure you have the support for the right CPU, the right board and the right device drivers.

Device Tree

- Many embedded architectures have a lot of non-discoverable hardware.
- Depending on the architecture, such hardware is either described using C code directly within the kernel, or using a special hardware description language in a *Device Tree*.
- ARM, PowerPC, OpenRISC, ARC, Microblaze are examples of architectures using the Device Tree.
- A *Device Tree Source*, written by kernel developers, is compiled into a binary *Device Tree Blob*, passed at boot time to the kernel.
  - There is one different Device Tree for each board/platform supported by the kernel, available in `arch/arm/boot/dts/<board>.dtb`.
- The bootloader must load both the kernel image and the Device Tree Blob in memory before starting the kernel.

Witekio
EMBEDDING SUCCESS

Building and installing the kernel

- Run `make`
- Copy the final kernel image to the target storage
  - can be `uImage`, `zImage`, `vmlinux`, `bzImage` in `arch/<arch>/boot`
  - copying the Device Tree Blob might be necessary as well, they are available in `arch/<arch>/boot/dts`
- `make install` is rarely used in embedded development, as the kernel image is a single file, easy to handle
  - It is however possible to customize the make install behaviour in `arch/<arch>/boot/install.sh`
- `make modules_install` is used even in embedded development, as it installs many modules and description files
  - `make INSTALL_MOD_PATH=<dir>/ modules_install`
  - The `INSTALL_MOD_PATH` variable is needed to install the modules in the target root filesystem instead of your host root filesystem.

Witekio
EMBEDDING SUCCESS

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` or `bootm X - Y`
     The `-` in the middle indicates no *initramfs*

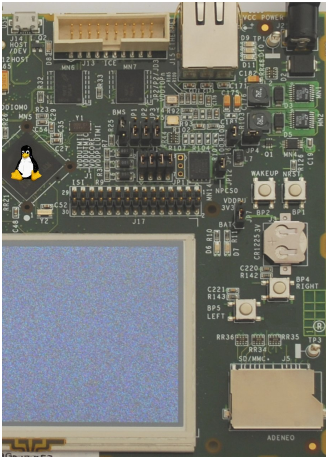
Witekio EMBEDDING SUCCESS

Kernel command line

- In addition to the compile time configuration, the kernel behaviour can be adjusted with no recompilation using the **kernel command line**
- The kernel command line is a string that defines various arguments to the kernel
  - It is very important for system configuration
  - `root=` for the root filesystem (covered later)
  - `console=` for the destination of kernel messages
  - Many more exist. The most important ones are documented in `Documentation/kernel-parameters.txt` in kernel sources.
- This kernel command line is either
  - Passed by the bootloader. In U-Boot, the contents of the `bootargs` environment variable is automatically passed to the kernel
  - Built into the kernel, using the `CONFIG_CMDLINE` option.

Basic Driver Development

Free Electrons, Witekio



Kernel Modules

Kernel 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.

Module dependencies

- Some kernel modules can depend on other modules, which need to be loaded first.
- Example: the `usb-storage` module depends on the `scsi_mod`, `libusual` and `usbcore` modules.
- Dependencies are described in
  `/lib/modules/<kernel-version>/modules.dep`
  This file is generated when you run `make modules_install`.

Witekio
EMBEDDING SUCCESS

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 (**d**iagnostic **mes**sa**g**e)
- Kernel log messages are also displayed in the system console (console messages can be filtered by level using the `loglevel` kernel parameter, or completely disabled with the `quiet` parameter).
- Note that you can write to the kernel log from user space too:
  `echo "<n>Debug info" > /dev/kmsg`

Module utilities (1)

- `modinfo <module_name>`
  `modinfo <module_path>.ko`
  Gets information about a module: parameters, license, description and dependencies.
  Very useful before deciding to load a module or not.
- `sudo insmod <module_path>.ko`
  Tries to load the given module. The full path to the module object file must be given.

Witekio
EMBEDDING SUCCESS

Understanding module loading issues

- When loading a module fails, `insmod` often doesn't give you enough details!
- Details are often available in the kernel log.
- Example:

```
$ sudo insmod ./intr_monitor.ko
insmod: error inserting './intr_monitor.ko': -1 Device or resource busy
$ dmesg
[17549774.552000] Failed to register handler for irq channel 2
```

Module utilities (2)

- `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`!

Module utilities (3)

- `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)

Witekio
EMBEDDING SUCCESS

Passing parameters to modules

- Find available parameters:
  `modinfo snd-intel8x0m`
- Through `insmod`:
  `sudo insmod ./snd-intel8x0m.ko index=-2`
- Through `modprobe`:
  Set parameters in `/etc/modprobe.conf` or in any file in `/etc/modprobe.d/`:
  `options snd-intel8x0m index=-2`
- Through the kernel command line, when the driver is built statically into the kernel:
  `snd-intel8x0m.index=-2`
  - `snd-intel8x0m` is the *driver name*
  - `index` is the *driver parameter name*
  - `-2` is the *driver parameter value*

Witekio
EMBEDDING SUCCESS

Check module parameter values

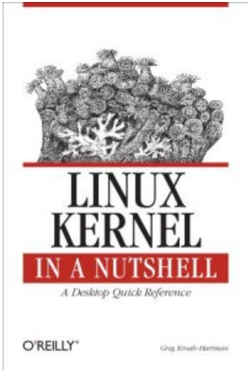
How to find the current values for the parameters of a loaded module?

- Check `/sys/module/<name>/parameters`.
- There is one file per parameter, containing the parameter value.

Useful reading

Linux Kernel in a Nutshell, Dec 2006

- By Greg Kroah-Hartman, O'Reilly
  `http://www.kroah.com/lkn/`
- A good reference book and guide on configuring, compiling and managing the Linux kernel sources.
- Freely available on-line!
  Great companion to the printed book for easy electronic searches!
  Available as single PDF file on
  `http://www.kroah.com/lkn/`



Witekio
EMBEDDING SUCCESS

Basic Driver Development

Kernel Module Development

Hello Module 1/2

```c
/* hello.c */
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

static int __init hello_init(void)
{
  pr_alert("Good morrow to this fair assembly.\n");
  return 0;
}

static void __exit hello_exit(void)
{
  pr_alert("Alas, poor world, what treasure hast thou lost!\n");
}

module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Greeting module");
MODULE_AUTHOR("William Shakespeare");
```

Witekio
EMBEDDING SUCCESS

Hello Module 2/2

- `__init`
  - removed after initialization (static kernel or module.)
- `__exit`
  - discarded when module compiled statically into the kernel, or when module unloading support is not enabled.

Witekio
EMBEDDING SUCCESS

Hello Module Explanations

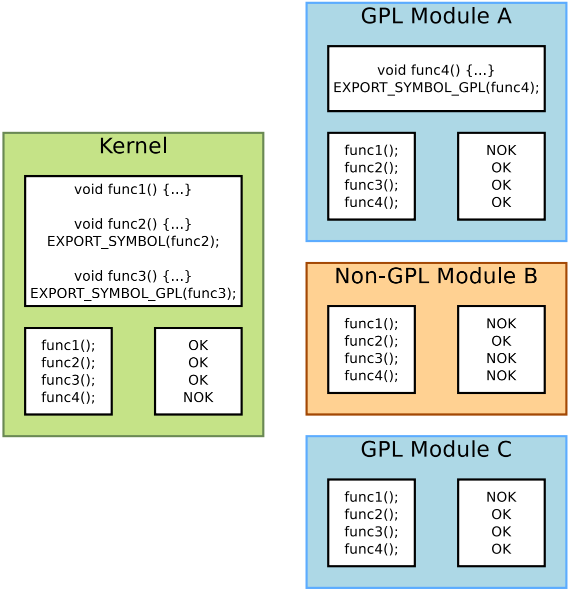
- Headers specific to the Linux kernel: `linux/xxx.h`
  - No access to the usual C library, we're doing kernel programming
- An initialization function
  - Called when the module is loaded, returns an error code (`0` on success, negative value on failure)
  - Declared by the `module_init()` macro: the name of the function doesn't matter, even though `<modulename>_init()` is a convention.
- A cleanup function
  - Called when the module is unloaded
  - Declared by the `module_exit()` macro.
- Metadata information declared using `MODULE_LICENSE()`, `MODULE_DESCRIPTION()` and `MODULE_AUTHOR()`

Symbols Exported to Modules 1/2

- From a kernel module, only a limited number of kernel functions can be called
- Functions and variables have to be explicitly exported by the kernel to be visible to a kernel module
- Two macros are used in the kernel to export functions and variables:
  - `EXPORT_SYMBOL(symbolname)`, which exports a function or variable to all modules
  - `EXPORT_SYMBOL_GPL(symbolname)`, which exports a function or variable only to GPL modules
- A normal driver should not need any non-exported function.

Witekio
EMBEDDING SUCCESS

Symbols exported to modules 2/2



Kernel
void func1() {...}
void func2() {...}
EXPORT_SYMBOL(func2);
void func3() {...}
EXPORT_SYMBOL_GPL(func3);
func1(); func2(); func3(); func4();
OK OK OK NOK
GPL Module A
void func4() {...}
EXPORT_SYMBOL_GPL(func4);
func1(); func2(); func3(); func4();
NOK OK OK OK
Non-GPL Module B
func1(); func2(); func3(); func4();
NOK OK NOK NOK
GPL Module C
func1(); func2(); func3(); func4();
NOK OK OK OK

Module License

- Several usages
  - Used to restrict the kernel functions that the module can use if it isn't a GPL licensed module
    - Difference between `EXPORT_SYMBOL()` and `EXPORT_SYMBOL_GPL()`
  - Used by kernel developers to identify issues coming from proprietary drivers, which they can't do anything about ("Tainted" kernel notice in kernel crashes and oopses).
  - Useful for users to check that their system is 100% free (check `/proc/sys/kernel/tainted`)
- Values
  - GPL compatible (see `include/linux/license.h`: `GPL`, `GPL v2`, `GPL and additional rights`, `Dual MIT/GPL`, `Dual BSD/GPL`, `Dual MPL/GPL`)
  - `Proprietary`

Witekio
EMBEDDING SUCCESS

Compiling a Module

- Two solutions
  - *Out of tree*
    - When the code is outside of the kernel source tree, in a different directory
    - Advantage: Might be easier to handle than modifications to the kernel itself
    - Drawbacks: Not integrated to the kernel configuration/compilation process, needs to be built separately, the driver cannot be built statically
  - Inside the kernel tree
    - Well integrated into the kernel configuration/compilation process
    - Driver can be built statically if needed

Witekio
EMBEDDING SUCCESS

Compiling an out-of-tree Module 1/2

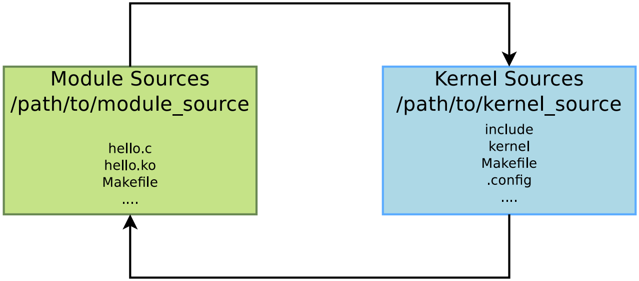
- The below `Makefile` should be reusable for any single-file out-of-tree Linux module
- The source file is `hello.c`
- Just run `make` to build the `hello.ko` file

```
ifneq ($(KERNELRELEASE),)
obj-m := hello.o
else
KDIR := /path/to/kernel/sources

all:
<tab>$(MAKE) -C $(KDIR) M=$$PWD
endif
```

- For `KDIR`, you can either set:
  - full kernel source directory
    (configured + `make modules_prepare`)
  - or just kernel headers directory (`make headers_install`)

Witekio
EMBEDDING SUCCESS

Compiling an out-of-tree Module 2/2



- The module Makefile is interpreted with `KERNELRELEASE` undefined, so it calls the kernel Makefile, passing the module directory in the `M` variable
- The kernel Makefile knows how to compile a module, and thanks to the `M` variable, knows where the Makefile for our module is. The module Makefile is interpreted with `KERNELRELEASE` defined, so the kernel sees the `obj-m` definition.

Modules and Kernel Version

- To be compiled, a kernel module needs access to the kernel headers, containing the definitions of functions, types and constants.
- Two solutions
  - Full kernel sources
  - Only kernel headers (`linux-headers-*` packages in Debian/Ubuntu distributions)
- The sources or headers must be configured
  - Many macros or functions depend on the configuration
- A kernel module compiled against version X of kernel headers will not load in kernel version Y
  - `modprobe` / `insmod` will say `Invalid module format`

New Driver in Kernel Sources 1/2

- To add a new driver to the kernel sources:
  - Add your new source file to the appropriate source directory. Example: `drivers/usb/serial/navman.c`
  - Single file drivers in the common case, even if the file is several thousand lines of code big. Only really big drivers are split in several files or have their own directory.
  - Describe the configuration interface for your new driver by adding the following lines to the `Kconfig` file in this directory:

```
config USB_SERIAL_NAVMAN
        tristate "USB Navman GPS device"
        depends on USB_SERIAL
        help
          To compile this driver as a module, choose M
          here: the module will be called navman.
```

New Driver in Kernel Sources 2/2

- Add a line in the `Makefile` file based on the `Kconfig` setting:
  `obj-$(CONFIG_USB_SERIAL_NAVMAN) += navman.o`
- It tells the kernel build system to build `navman.c` when the `USB_SERIAL_NAVMAN` option is enabled. It works both if compiled statically or as a module.
  - Run `make xconfig` and see your new options!
  - Run `make` and your new files are compiled!
  - See `Documentation/kbuild/` for details and more elaborate examples like drivers with several source files, or drivers in their own subdirectory, etc.

Witekio
EMBEDDING SUCCESS

Hello Module with Parameters 1/2

```
/* hello_param.c */
#include <linux/init.h>
#include <linux/module.h>

MODULE_LICENSE("GPL");

/* A couple of parameters that can be passed in: how many
   times we say hello, and to whom */

static char *whom = "world";
module_param(whom, charp, 0);

static int howmany = 1;
module_param(howmany, int, 0);
```

Hello Module with Parameters 2/2

```
static int __init hello_init(void)
{
  int i;
  for (i = 0; i < howmany; i++)
    pr_alert("(%d) Hello, %s\n", i, whom);
  return 0;
}

static void __exit hello_exit(void)
{
  pr_alert("Goodbye, cruel %s\n", whom);
}

module_init(hello_init);
module_exit(hello_exit);
```

Thanks to Jonathan Corbet for the example.

Declaring a module parameter

```
module_param(
    name, /* name of an already defined variable */
    type, /* either byte, short, ushort, int, uint, long, ulong,
             charp, bool or invbool. (checked at run time!) */
    perm  /* for /sys/module/<module_name>/parameters/<param>,
             0: no such module parameter value file */
);

/* Example */
static int irq=5;
module_param(irq, int, S_IRUGO);
```

Modules parameter arrays are also possible with `module_param_array()`.

Witekio
EMBEDDING SUCCESS

Last slide

Thank you!

And may the Source be with you