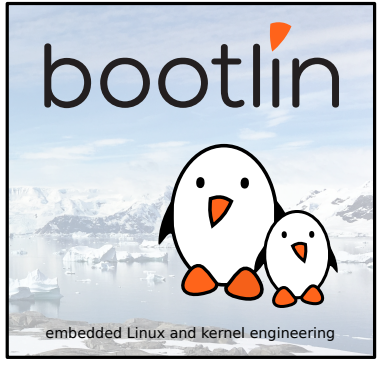
Linux kernel and driver development training



Latest update: February 27, 2024 (by Florian Vaussard).

Document updates and training details:
https://bootlin.com/training/kernel

Corrections, suggestions, contributions and translations are welcome!
Send them to feedback@bootlin.com

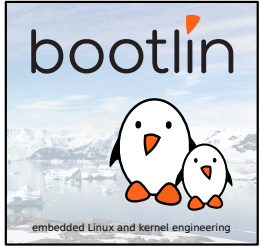
ReDS
Reconfigurable & embedded Digital Systems

Linux kernel and driver development training

- These slides are the training materials for Bootlin's *Linux kernel and driver development* training course.
- If you are interested in following this course with an experienced Bootlin trainer, we offer:
  - **Public online sessions**, opened to individual registration. Dates announced on our site, registration directly online.
  - **Dedicated online sessions**, organized for a team of engineers from the same company at a date/time chosen by our customer.
  - **Dedicated on-site sessions**, organized for a team of engineers from the same company, we send a Bootlin trainer on-site to deliver the training.
- Details and registrations: https://bootlin.com/training/kernel
- Contact: training@bootlin.com

Icon by Eucalyp, Flaticon

Linux Kernel Introduction



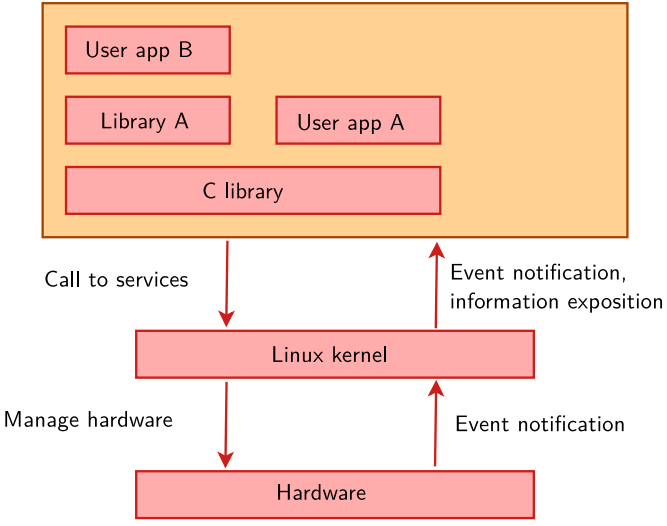
RoDS
Reconfigurable & embedded Digital Systems

Origin

- 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.
- As of today, about 2,000+ people contribute to each kernel release, individuals or companies big and small.

Linus Torvalds in 2014
Image credits (Wikipedia):
https://bit.ly/2UIa1TD

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 for "multiplexing" the hardware resource.

System calls

- The main interface between the kernel and user space is the set of system calls
- About 400 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 make a system call directly but rather use the corresponding C library function

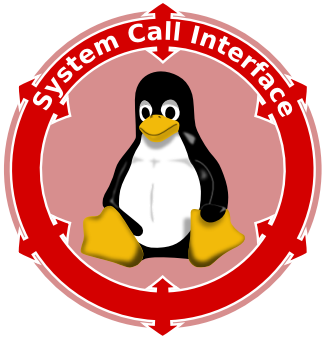


Image credits (Wikipedia): https://bit.ly/2U2rdGB

Pseudo filesystems

- Linux makes system and kernel information available in user space through **pseudo filesystems**, sometimes also called **virtual filesystems**
- Pseudo filesystems allow applications to see directories and files that do not exist on any real storage: they are created and updated on the fly by the kernel
- The two most important pseudo filesystems are
  - `proc`, usually mounted on `/proc`:
    Operating system related information (processes, memory management parameters...)
  - `sysfs`, usually mounted on `/sys`:
    Representation of the system as a tree of devices connected by buses. Information gathered by the kernel frameworks managing these devices.

Linux kernel sources

Location of the official kernel sources

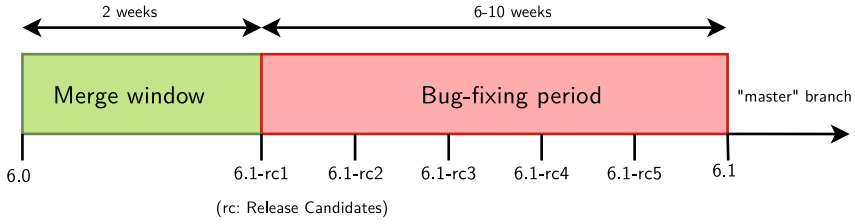
- The mainline versions of the Linux kernel, as released by Torvalds
  - These versions follow the development model of the kernel (`master` branch)
  - They may not contain the latest developments from a specific area yet
  - A good pick for products development phase
  - https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

Linux versioning scheme

- Until 2003, there was a new "stabilized" release branch of Linux every 2 or 3 years (2.0, 2.2, 2.4). Development branches took 2-3 years to be merged (too slow!).
- Since 2003, there is a new official release of Linux about every 10 weeks:
  - Versions `2.6` (Dec. 2003) to `2.6.39` (May 2011)
  - Versions `3.0` (Jul. 2011) to `3.19` (Feb. 2015)
  - Versions `4.0` (Apr. 2015) to `4.20` (Dec. 2018)
  - Versions `5.0` (Mar. 2019) to `5.19` (July 2022)
  - Version `6.0` was released in Oct. 2022.
- Features are added to the kernel in a progressive way. Since 2003, kernel developers have managed to do so without having to introduce a massively incompatible development branch.

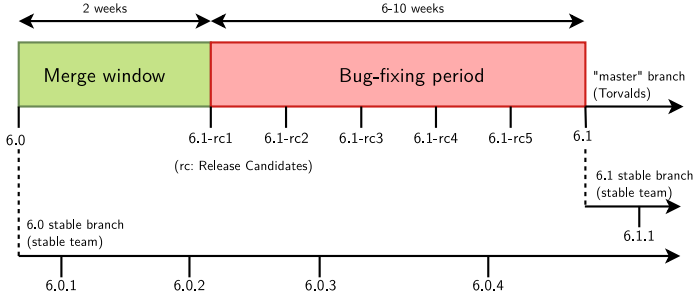
Linux development model

- Each new release starts with a two-week merge window for new features
- Follow about 8 release candidates (one week each)
- Until adoption of a new official release.



Need to further stabilize the official kernels

- Issue: bug and security fixes only merged into the master branch, need to update to the latest kernel to benefit from them.
- Solution: a stable maintainers team goes through all the patches merged into Torvald's tree and backports the relevant ones into their stable branches for at least a few months.



Location of the stable kernel sources

- The stable versions of the Linux kernel, as maintained by a maintainers group
  - These versions do not bring new features compared to Linus' tree
  - Only bug fixes and security fixes are pulled there
  - Each version is stabilized during the development period of the next mainline kernel
  - A good pick for products commercialization phase
  - https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git
  - Certain versions will be maintained much longer

Need for long term support

- Issue: bug and security fixes only released for most recent kernel versions.
- Solution: the last release of each year is made an LTS *(Long Term Support)* release, and is supposed to be supported (and receive bug and security fixes) for up to 6 years. But recent versions might be supported for only 2 years.

Longterm release kernels

Version	Maintainer	Released	Projected EOL
6.6	Greg Kroah-Hartman & Sasha Levin	2023-10-29	Dec, 2026
6.1	Greg Kroah-Hartman & Sasha Levin	2022-12-11	Dec, 2026
5.15	Greg Kroah-Hartman & Sasha Levin	2021-10-31	Dec, 2026
5.10	Greg Kroah-Hartman & Sasha Levin	2020-12-13	Dec, 2026
5.4	Greg Kroah-Hartman & Sasha Levin	2019-11-24	Dec, 2025
4.19	Greg Kroah-Hartman & Sasha Levin	2018-10-22	Dec, 2024
4.14	Greg Kroah-Hartman & Sasha Levin	2017-11-12	Jan, 2024

Captured on `https://kernel.org` in Nov. 2023, following the *Releases* link.

- Example at Google: starting from *Android O (2017)*, all new Android devices have to run such an LTS kernel.

Need for even longer term support

- You could also get long term support from a commercial embedded Linux provider.
  - Wind River Linux can be supported for up to 15 years.
  - Ubuntu Core can be supported for up to 10 years.
- *"If you are not using a supported distribution kernel, or a stable / longterm kernel, you have an insecure kernel"* - Greg KH, 2019
  Some vulnerabilities are fixed in stable without ever getting a CVE.
- The *Civil Infrastructure Platform* project is an industry / Linux Foundation effort to support much longer (at least 10 years) selected LTS versions (currently 4.4, 4.19, 5.10 and 6.1) on selected architectures. See https://wiki.linuxfoundation.org/civilinfrastructureplatform/start.

Location of non-official kernel sources

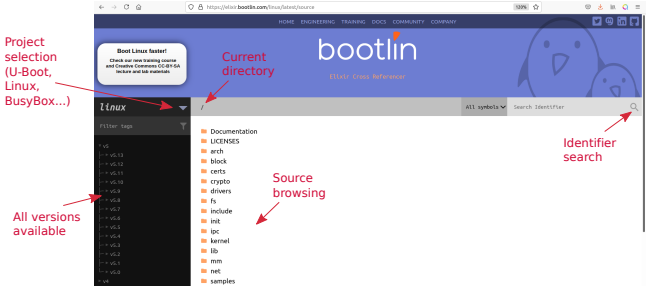
- Many chip vendors supply their own kernel sources
  - Focusing on hardware support first
  - Can have a very important delta with mainline Linux
  - Sometimes they break support for other platforms/devices without caring
  - Useful in early phases only when mainline hasn't caught up yet (many vendors invest in the mainline kernel at the same time)
  - Suitable for PoC, not suitable for products on the long term as usually no updates are provided to these kernels
  - Getting stuck with a deprecated system with broken software that cannot be updated has a real cost in the end
- Many kernel sub-communities maintain their own kernel, with usually newer but fewer stable features, only for cutting-edge development
  - Architecture communities (ARM, MIPS, PowerPC, etc)
  - Device drivers communities (I2C, SPI, USB, PCI, network, etc)
  - Other communities (filesystems, memory-management, scheduling, etc)
  - Not suitable to be used in products

Getting Linux sources

- The kernel sources are available from https://kernel.org/pub/linux/kernel as **full tarballs** (complete kernel sources) and **patches** (differences between two kernel versions).
- But today the entire open source community has settled in favor of Git
  - Fast, efficient with huge code bases, reliable, open source
  - Incidentally written by Torvalds

Going through Linux sources

- Development tools:
  - Any text editor will work
  - Vim and Emacs support ctags and cscope and therefore can help with symbol lookup and auto-completion.
  - It's also possible to use more elaborate IDEs to develop kernel code, like Visual Studio Code.
- Powerful web browsing: Elixir
  - Generic source indexing tool and code browser for C and C++.
  - Very easy to find symbols declaration/implementation/usage
  - Try out https://elixir.bootlin.com!



Linux kernel size and structure

- Linux v5.18 sources: close to 80k files, 35M lines, 1.3GiB
- But a compressed Linux kernel just sizes a few megabytes.
- So, why are these sources so big?
  Because they include numerous device drivers, network protocols, architectures, filesystems... The core is pretty small!
- As of kernel version v5.18 (in percentage of total number of lines):

- drivers/: 61.1%
- arch/: 11.6%
- fs/: 4.4%
- sound/: 4.1%
- tools/: 3.9%
- net/: 3.7%
- include/: 3.5%
- Documentation/: 3.4%
- kernel/: 1.3%
- lib/: 0.7%
- usr/: 0.6%
- mm/: 0.5%
- scripts/, security/, crypto/, block/, samples/, ipc/, virt/, init/, certs/: <0.5%
- Build system files: Kbuild, Kconfig, Makefile
- Other files: COPYING, CREDITS, MAINTAINERS, README

Linux kernel source code

Programming language

- Implemented in C like all UNIX systems
- A little Assembly is used too:
  - CPU and machine initialization, exceptions
  - Critical library routines.
- No C++ used, see http://vger.kernel.org/lkml/#s15-3
- All the code compiled with gcc
  - Many gcc specific extensions used in the kernel code, any ANSI C compiler will not compile the kernel
  - See https://gcc.gnu.org/onlinedocs/gcc-10.2.0/gcc/C-Extensions.html
- A subset of the supported architectures can be built with the LLVM C compiler (Clang) too: https://clangbuiltlinux.github.io/
- Rust support was recently added in Linux v6.1 and is under active development. See the Rust-related articles on LWN for the latest news.

No C library

- The kernel has to be standalone and can't use user space code.
- Architectural reason: user space is implemented on top of kernel services, not the opposite.
- Technical reason: the kernel is on its own during the boot up phase, before it has accessed a root filesystem.
- Hence, 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()`, ...

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
  - I/O memory access
  - Memory barriers to provide ordering guarantees if needed
  - DMA API to flush and invalidate caches if needed
- Never use floating point numbers in kernel code. Your code may need to run on a low-end processor without a floating point unit.

Linux kernel to user API/ABI stability

Linux kernel to userspace API is stable

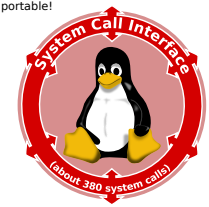
- Source code for userspace applications will not have to be updated when compiling for a more recent kernel
  - System calls, `/proc` and `/sys` content cannot be removed or changed. Only new entries can be added.

Linux kernel to userspace ABI is stable

- Binaries are portable and can be executed on a more recent kernel
  - The way memory is accessed, the size of the variables in memory, how structures are organized, the calling convention, etc, are all stable over time.

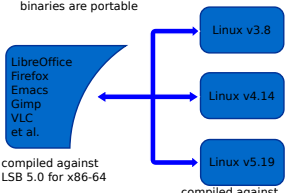
Linux kernel to user API

☑ API stability **is** guaranteed, source code is portable!



Linux kernel to user ABI

☑ compatible ABI **can be** guaranteed, binaries are portable



Modified Image from Wikipedia:
https://bit.ly/2U2rdGB

Linux internal API/ABI instability

Linux internal API is not stable

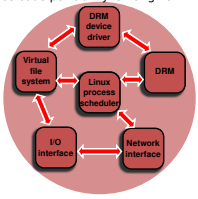
- The source code of a driver is not portable across versions
  - In-tree drivers are updated by the developer proposing the API change: works great for mainline code
  - An out-of-tree driver compiled for a given version may no longer compile or work on a more recent one
  - See `process/stable-api-nonsense` for reasons why

Linux internal ABI is not stable

- A binary module compiled for a given kernel version cannot be used with another version
  - The module loading utilities will perform this check prior to the insertion

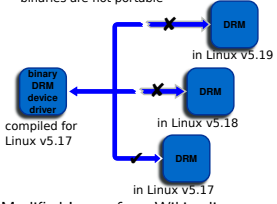
Linux internal API

✘ API stability **is not** guaranteed, source code portability is not given



Linux internal ABI

✘ **no** stable ABI over Linux kernel releases, binaries are not portable



Modified Image from Wikipedia: https://bit.ly/2U2rdGB

Kernel memory constraints

- No memory protection
- The kernel doesn't try to recover from attemps to access illegal memory locations. It just dumps *oops* messages on the system console.
- Fixed size stack (8 or 4 KB). Unlike in user space, no mechanism was implemented to make it grow. Don't use recursion!
- Swapping is not implemented for kernel memory either (except *tmpfs* which lives completely in the page cache and on swap)

Linux kernel licensing constraints

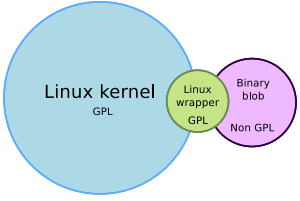
- The Linux kernel is licensed under the GNU General Public License version 2
  - This license gives you the right to use, study, modify and share the software freely
- However, when the software is redistributed, either modified or unmodified, the GPL requires that you redistribute the software under 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 GPL license has been successfully enforced in courts:
  https://en.wikipedia.org/wiki/Gpl-violations.org#Notable_victories
- 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

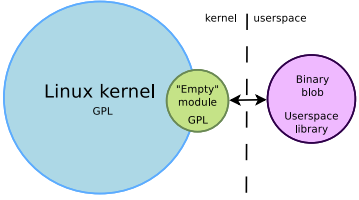
- It is illegal to distribute a binary kernel that includes statically compiled proprietary drivers
- The kernel modules are a gray area: unclear if they are legal or not
  - The general opinion of the kernel community is that proprietary modules are bad: process/kernel-driver-statement
  - From a legal point of view, each driver is probably a different case:
    - Are they derived works of the kernel?
    - Are they designed to be used with another operating system?
- Is it really useful to keep drivers secret anyway?

Abusing the kernel licensing constraints

- There are some examples of proprietary drivers
  - Nvidia uses a wrapper between their drivers and the kernel
  - They claim the drivers could be used with a different OS with another wrapper
  - Unclear whether it makes it legal or not



- The current trend is to hide the logic in the firmware or in userspace. The GPL kernel driver is almost empty and either:
  - Blindly writes an incoming flow of bytes in the hardware
  - Exposes a huge MMIO region to userspace through `mmap`



Advantages of GPL drivers

- You don't have to write your driver from scratch. You can reuse code from similar free software drivers.
- Your drivers can be freely and easily shipped by others (for example by Linux distributions or embedded Linux build systems).
- Legal certainty, you are sure that a GPL driver is fine from a legal point of view.

Advantages of mainlining your kernel drivers

- The community, reviewers and maintainers will review your code before accepting it, offering you the opportunity to enhance it and understand better the internal APIs.
- Once accepted, you will get cost-free bug and security fixes, support for new features, and general improvements.
- Your work will automatically follow the API changes.
- Accessing your code will be much easier for users.
- Your code will remain valid no matter the kernel version.

This will for sure reduce your maintenance and support work

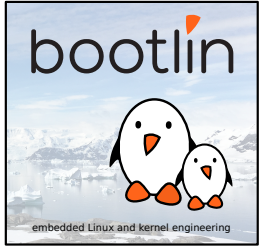
User space device drivers 1/2

- The kernel provides certain mechanisms to access hardware directly from userspace:
  - USB devices with *libusb*, https://libusb.info/
  - SPI devices with *spidev*, spi/spidev
  - I2C devices with *i2cdev*, i2c/dev-interface
  - Memory-mapped devices with *UIO*, including interrupt handling, driver-api/uio-howto
- These solutions can only be used if:
  - 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.
- Certain classes of devices like printers and scanners do not have any kernel support, they have always been handled in user space for historical reasons.
- Otherwise this is *not* how the system should be architectured. Kernel drivers should always be preferred!

User space device drivers 2/2

- Advantages
  - No need for kernel coding skills.
  - 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 use floating-point computation.
  - Potentially higher performance, especially for memory-mapped devices, thanks to the avoidance of system calls.
- Drawbacks
  - The kernel has no longer access to the device.
  - None of the standard applications will be able to use it.
  - Cannot use any hardware abstraction or software helpers from the kernel
  - Need to adapt applications when changing the hardware.
  - Less straightforward to handle interrupts: increased latency.

Linux Kernel Usage



RoDS
Reconfigurable & embedded Digital Systems

Kernel configuration

Kernel configuration

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

Kernel configuration and build system

- The kernel configuration and build system is based on multiple Makefiles
- One only interacts with the main `Makefile`, present at the **top directory** of the kernel source tree
- Interaction takes place
  - using the `make` tool, which parses the Makefile
  - through various **targets**, defining which action should be done (configuration, compilation, installation, etc.).
  - Run `make help` to see all available targets.
- Example
  - `cd linux/`
  - `make <target>`

Specifying the target architecture

First, specify the architecture for the kernel to build

- Set `ARCH` to the name of a directory under `arch/`:
  `ARCH=arm` or `ARCH=arm64` or `ARCH=riscv`, etc
- By default, the kernel build system assumes that the kernel is configured and built for the host architecture (`x86` in our case, native kernel compiling)
- The kernel build system will use this setting to:
  - Use the configuration options for the target architecture.
  - Compile the kernel with source code and headers for the target architecture.

Choosing a compiler

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

- Specifying the compiler is already needed at configuration time, as some kernel configuration options depend on the capabilities of the compiler.
- When compiling natively
  - Leave `CROSS_COMPILE` undefined and the kernel will be natively compiled for the host architecture using `gcc`.
- When using a cross-compiler
  - Specify the prefix of your cross-compiler executable, for example for `arm-linux-gnueabi-gcc`: `CROSS_COMPILE=arm-linux-gnueabi-`

Set `LLVM` to `1` to compile your kernel with Clang.
See our LLVM tools for the Linux kernel presentation.

Specifying ARCH and CROSS_COMPILE

There are actually two ways of defining `ARCH` and `CROSS_COMPILE`:

- Pass `ARCH` and `CROSS_COMPILE` on the `make` command line:
  `make ARCH=arm CROSS_COMPILE=arm-linux- ...`
  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, see also the https://direnv.net/ project.

Initial configuration

Difficult to find which kernel configuration will work with your hardware and root filesystem. Start with one that works!

- Desktop or server case:
  - Advisable to start with the configuration of your running kernel: `cp /boot/config-`uname -r` .config`
- Embedded platform case:
  - Default configurations stored in-tree as minimal configuration files (only listing settings that are different with the defaults) in `arch/<arch>/configs/`
  - `make help` will list the available configurations for your platform
  - To load a default configuration file, just run `make foo_defconfig` (will erase your current `.config`!)
    - On ARM 32-bit, there is usually one default configuration per CPU family
    - On ARM 64-bit, there is only one big default configuration to customize

Create your own default configuration

- Use a tool such as `make menuconfig` to make changes to the configuration
- Saving your changes will overwrite your `.config` (not tracked by Git)
- When happy with it, create your own default configuration file:
  - Create a minimal configuration (non-default settings) file:
    `make savedefconfig`
  - Save this default configuration in the right directory:
    `mv defconfig arch/<arch>/configs/myown_defconfig`
- This way, you can share a reference configuration inside the kernel sources and other developers can now get the same `.config` as you by running
  `make myown_defconfig`

Built-in 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 built-in 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

Kconfig

Kernel configuration options are defined in `Kconfig` files:

- There are approximately 1600 Kconfig files throughout the tree (as of v6.1)
- `Kconfig` uses a dedicated language, which is documented in the kernel's Kconfig Language.

Example (from `drivers/tty/serial/8250/Kconfig`):

```
config SERIAL_8250
        tristate "8250/16550 and compatible serial support"
        depends on !S390
        select SERIAL_CORE
        select SERIAL_MCTRL_GPIO if GPIOLIB
        help
          This selects whether you want to include the driver for the standard
          serial ports.  The standard answer is Y.  People who might say N
```

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
  Example: `CONFIG_PAGE_OFFSET=0xC0000000`
- `string` options, to specify string values
  Example: `CONFIG_LOCALVERSION=-no-network`
  Useful to distinguish between two kernels built from different options

Kernel option dependencies

Enabling a network driver requires the network stack to be enabled, therefore configuration symbols have two ways to express dependencies:

- `depends on` dependency:

```
config B
    depends on A
```

  - B is not visible until A is enabled
  - Works well for dependency chains

- `select` dependency:

```
config A
    select B
```

  - When A is enabled, B is enabled too (and cannot be disabled manually)
  - Should preferably not select symbols with `depends on` dependencies
  - Used to declare hardware features or select libraries

```
config SPI_ATH79
        tristate "Atheros AR71XX/AR724X/AR913X SPI controller driver"
        depends on ATH79 || COMPILE_TEST
        select SPI_BITBANG
        help
          This enables support for the SPI controller present on the
          Atheros AR71XX/AR724X/AR913X SoCs.
```

Kernel configuration details

- The configuration is stored in the `.config` file at the root of kernel sources
  - Simple text file, `CONFIG_PARAM=value`
  - Options are grouped by sections and are prefixed with `CONFIG_`
  - Included by the top-level kernel Makefile
  - Typically not edited by hand because of the dependencies

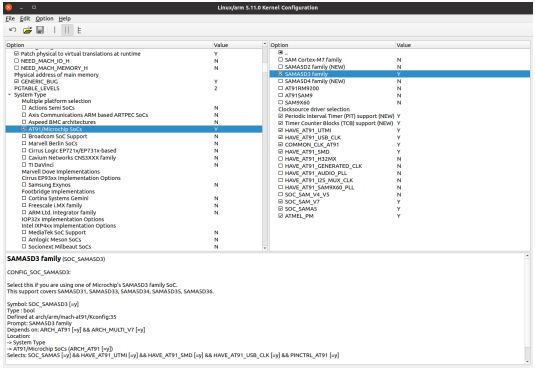
```
#
# CD-ROM/DVD Filesystems
#
CONFIG_ISO9660_FS=m
CONFIG_JOLIET=y
CONFIG_ZISOFS=y
CONFIG_UDF_FS=y
# end of CD-ROM/DVD Filesystems

#
# DOS/FAT/EXFAT/NT Filesystems
#
CONFIG_FAT_FS=y
CONFIG_MSDOS_FS=y
# CONFIG_VFAT_FS is not set
CONFIG_FAT_DEFAULT_CODEPAGE=437
# CONFIG_EXFAT_FS is not set
```

xconfig

`make xconfig`

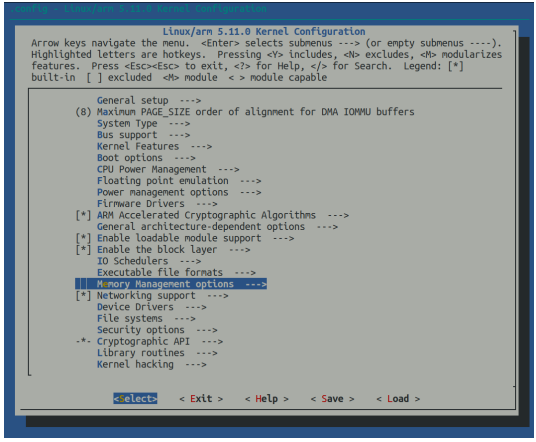
- A graphical interface to configure the kernel.
- File browser: easy to load configuration files
- Search interface to look for parameters (`[Ctrl] + [f]`)
- Required Debian/Ubuntu packages: `qtbase5-dev` on Ubuntu 22.04



menuconfig

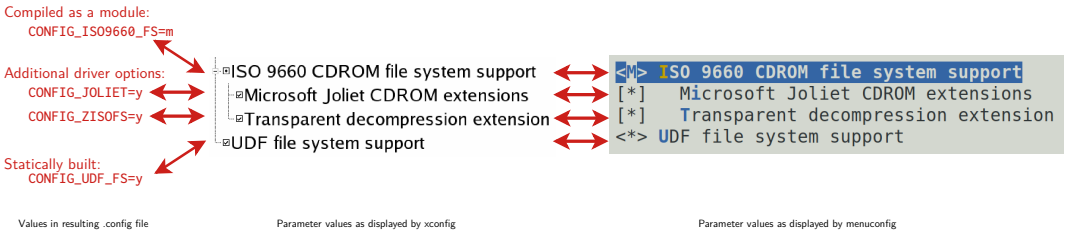
`make menuconfig`

- Useful when no graphics are available. Very efficient interface.
- Same interface found in other tools: BusyBox, Buildroot...
- Convenient number shortcuts to jump directly to search results.
- Required Debian/Ubuntu packages: `libncurses-dev`
- Alternative: `make nconfig` (now also has the number shortcuts)



Kernel configuration options

You can switch from one tool to another, they all load/save the same `.config` file, and show the same set of options



make oldconfig

`make oldconfig`

- Useful to upgrade a `.config` file from an earlier kernel release
- Asks for values for new parameters.
- ... unlike `make menuconfig` and `make xconfig` which silently set default values for new parameters.

If you edit a `.config` file by hand, it's useful to run `make oldconfig` afterwards, to set values to new parameters that could have appeared because of dependency changes.

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 tools keep this `.config.old` backup copy.

Compiling and installing the kernel

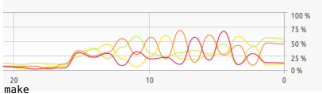
Kernel compilation

`make`

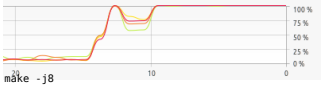
- Only works from the top kernel source directory
- Equivalent to `make all`
- Should not be performed as a priviledged user
- Run several **j**obs in parallel. Our advice: `ncpus * 2` to fully load the CPU and I/Os at all times.
  Example: `make -j 8`
- To **re**compile faster (7x according to some benchmarks), use the `ccache` compiler cache:
  `export CROSS_COMPILE="ccache arm-linux-"`

Benefits of parallel compile jobs (make -j<n>)

Tests on Linux 5.11 on arm
`make allnoconfig` configuration
`gnome-system-monitor` showing the load on 4 threads / 2 CPUs



`make`
Command: make
Total time: 129 s



`make -j8`
Command: make -j8
Total time: 67 s

Kernel compilation results

- `arch/<arch>/boot/Image`, uncompressed kernel image that can be booted
- `arch/<arch>/boot/*Image*`, compressed kernel images that can also be booted
  - `bzImage` for x86, `zImage` for ARM, `Image.gz` for RISC-V, `vmlinux.bin.gz` for ARC, etc.
- `arch/<arch>/boot/dts/*.dtb`, compiled Device Tree Blobs
- All kernel modules, spread over the kernel source tree, as `.ko` (*Kernel Object*) files.
- `vmlinux`, a raw uncompressed kernel image in the ELF format, useful for debugging purposes but generally not used for booting purposes

Kernel compilation: make targets

Instead of building everything, you can build a specific component:

- `make Image`: uncompressed kernel image
- `make zImage`: compressed ARM kernel
- `make dtbs`: Device Tree Blobs for enabled boards
- `make target.dtb`: a specific Device Tree Blob located in `arch/<arch>/boot/dts/target.dtb`
- `make dir/file.ko`: a specific module

Refer to `make help` for more options.

Kernel installation: native case

- `sudo make install`
  - Does the installation for the host system by default
- Installs
  - `/boot/vmlinuz-<version>`
    Compressed kernel image. Same as the one in `arch/<arch>/boot`
  - `/boot/System.map-<version>`
    Stores kernel symbol addresses for debugging purposes (obsolete: such information is usually stored in the kernel itself)
  - `/boot/config-<version>`
    Kernel configuration for this version
- In GNU/Linux distributions, typically re-runs the bootloader configuration utility to make the new kernel available at the next boot.

Kernel installation: embedded case

- `make install` is rarely used in embedded development, as the kernel image is a single file, easy to handle.
- Another reason is that there is no standard way to deploy and use the kernel image.
- Therefore making the kernel image (`zImage` + `target.dtb`) available to the target is usually manual or done through scripts in build systems.
- It is however possible to customize the `make install` behavior in `arch/<arch>/boot/install.sh`

Module installation: native case

- `sudo make modules_install`
  - Does the installation for the host system by default, so needs to be run as root
- Installs all modules in `/lib/modules/<version>/`
  - `kernel/`
    Module `.ko` (Kernel Object) files, in the same directory structure as in the sources.
  - `modules.alias`, `modules.alias.bin`
    Aliases for module loading utilities
  - `modules.dep`, `modules.dep.bin`
    Module dependencies. Kernel modules can depend on other modules, based on the symbols (functions and data structures) they use.
  - `modules.symbols`, `modules.symbols.bin`
    Tells which module a given symbol belongs to (related to module dependencies).
  - `modules.builtin`
    List of built-in modules of the kernel.

Module installation: embedded case

- In embedded development, you can't directly use `make modules_install` as it would install target modules in `/lib/modules` on the host!
- The `INSTALL_MOD_PATH` variable is needed to generate the module related files and install the modules in the target root filesystem instead of your host root filesystem (no need to be root):
  `make INSTALL_MOD_PATH=<dir>/ modules_install`

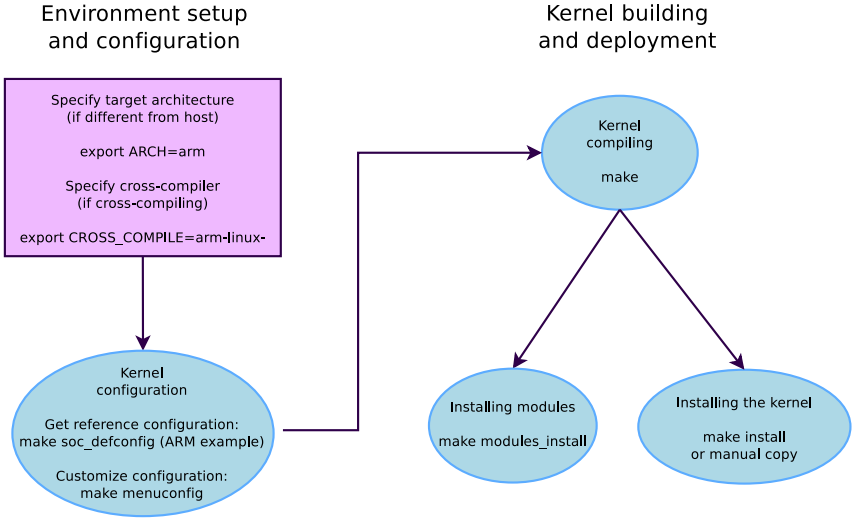
Kernel cleanup targets

- From `make help`:

```
Cleaning targets:
  clean           - Remove most generated files but keep the config and
                    enough build support to build external modules
  mrproper        - Remove all generated files + config + various backup files
  distclean       - mrproper + remove editor backup and patch files
```

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

Kernel building overview



Booting the kernel

Hardware description

- Many embedded architectures have a lot of non-discoverable hardware (serial, Ethernet, I2C, Nand flash, USB controllers...)
- This hardware needs to be described and passed to the Linux kernel.
- Using C code directly within the kernel is legacy, nowadays the bootloader/firmware is expected to provide this description when starting the kernel:
  - On x86: using ACPI tables
  - On most embedded devices: using an OpenFirmware Device Tree (DT)
- This way, a kernel supporting different SoCs knows which SoC and device initialization hooks to run on the current board.

Booting with U-Boot

- On ARM32, U-Boot can boot `zImage` (`bootz` command)
- On ARM64 or RISC-V, it boots the `Image` file (`booti` command)
- In addition to the kernel image, U-Boot should also pass a DTB to the kernel.
- The typical boot process is therefore:
  1. Load `zImage` at address X in memory
  2. Load `<board>.dtb` at address Y in memory
  3. Start the kernel with `boot[z|i] X - Y`
     The `-` in the middle indicates no *initramfs*

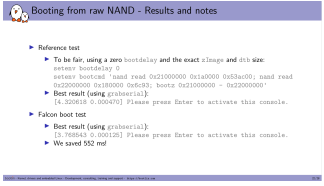
Kernel command line

- In addition to the compile time configuration, the kernel behavior 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
  - Example: `console=ttyS0 root=/dev/mmcblk0p2 rootwait`
  - Many more exist. The most important ones are documented in `admin-guide/kernel-parameters` in kernel documentation.

Passing the kernel command line

- U-Boot carries the Linux kernel command line string in its `bootargs` environment variable
- Right before starting the kernel, it will store the contents of `bootargs` in the `chosen` section of the Device Tree
- The kernel will behave differently depending on its configuration:
  - If `CONFIG_CMDLINE_FROM_BOOTLOADER` is set:
    The kernel will use only the string from the bootloader
  - If `CONFIG_CMDLINE_FORCE` is set:
    The kernel will only use the string received at configuration time in `CONFIG_CMDLINE`
  - If `CONFIG_CMDLINE_EXTEND` is set:
    The kernel will concatenate both strings

See the "Understanding U-Boot Falcon Mode" presentation from Michael Opdenacker, for details about how U-Boot boots Linux.



Slides: https://bootlin.com/pub/conferences/2021/lee/
Video: https://www.youtube.com/watch?v=LFe3x2QMhSo

Kernel log

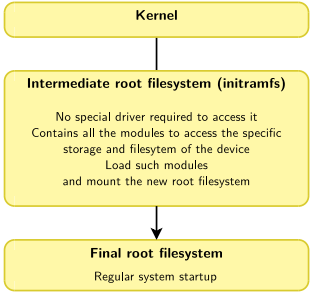
- The kernel keeps its messages in a circular buffer in memory
  - The size is configurable using `CONFIG_LOG_BUF_SHIFT`
- When a module is loaded, related information is available in the kernel log.
- Kernel log messages are available through the `dmesg` command (**d**iagnostic **mes**sa**g**e)
- Kernel log messages are also displayed on the console pointed by the `console=` kernel command line argument
  - Console messages can be filtered by level using the `loglevel` parameter:
    - `loglevel=` allows to filter messages displayed on the console based on priority
    - `ignore_loglevel` (same as `loglevel=8`) will lead to all messages being printed
    - `quiet` (same as `loglevel=0`) prevents any message from being displayed on the console
  - Example: `console=ttyS0 root=/dev/mmcblk0p2 loglevel=5`
- It is possible to write to the kernel log from user space:
  `echo "<n>Debug info" > /dev/kmsg`

Using kernel modules

Advantages of modules

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

Using kernel modules to support many different devices and setups



The modules in the initramfs are updated every time a kernel upgrade is available.

Module utilities: extracting information

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

- `modinfo <module_name>` (for modules in `/lib/modules`)
  `modinfo <module_path>.ko`
  Gets information about a module without loading it: parameters, license, description and dependencies.

Module utilities: loading

- `sudo insmod <module_path>.ko`
  Tries to load the given module. The full path to the module object file must be given.
- `sudo modprobe <top_module_name>`
  Most common usage of `modprobe`: tries to load all the dependencies of the given top module, 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`!

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: removals

- `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 <top_module_name>`
  Tries to remove the given top module and all its no longer needed dependencies

Passing parameters to modules

- Find available parameters:
  `modinfo usb-storage`
- Through `insmod`:
  `sudo insmod ./usb-storage.ko delay_use=0`
- Through `modprobe`:
  Set parameters in `/etc/modprobe.conf` or in any file in `/etc/modprobe.d/`:
  `options usb-storage delay_use=0`
- Through the kernel command line, when the module is built statically into the kernel:
  `usb-storage.delay_use=0`
  - `usb-storage` is the *module name*
  - `delay_use` is the *module parameter name*. It specifies a delay before accessing a USB storage device (useful for rotating devices).
  - `0` is the *module parameter value*

Check module parameter values

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

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