A COMPLETE, BEGINNER-FRIENDLY WALKTHROUGH

Building Your Own Linux Kernel

The Gentoo Way — configuration, compilation & installation from source

From sources → .config → bzImage → a bootable system

01

SECTION 01 · FOUNDATIONS

Custom Kernels Are Universal

Before Gentoo specifics: the kernel is just a program you can build yourself — on any distribution.

The kernel is just a program

One source tree. Every distro.

The Linux kernel is a single open codebase published at kernel.org. Arch, Debian, Fedora, Ubuntu and Gentoo all ship builds of that same source. Nothing about compiling it is Gentoo-only — what differs between distributions is only how each one packages, installs and boots the result. Once you understand the build, you can run a custom kernel anywhere.

You can run a custom kernel on any distro

The compile step is identical everywhere. Only the install conventions change.

Arch

Drop the image in /boot, regenerate the initramfs with mkinitcpio, update the bootloader.

Debian / Ubuntu

make deb-pkg builds .deb packages, or copy manually and run update-grub.

Fedora / RHEL

make rpm-pkg, or copy the image and run grubby / grub2-mkconfig.

Gentoo

Build in /usr/src/linux, install modules, copy bzImage, update GRUB. This deck's focus.

Why build your own kernel?

There are real benefits — and an honest counterpoint.

Reasons to do it

- Smaller image: include only drivers your hardware needs
- Faster boot and lower memory footprint
- Enable features a generic kernel omits (specific schedulers, debugging, hardening)
- Deep understanding of how your system actually works
- Full control: you decide what is built in, modular, or absent

The honest counterpoint

- Distribution kernels exist and work well for most people
- They are tested, signed, and auto-updated
- A custom kernel is maintenance you now own
- Recommendation: build to learn and optimize — but a distro kernel is a fine default
- You can always keep a distro kernel as a fallback boot entry

Gentoo gives you three paths

All are legitimate. This presentation follows the manual path — where the real learning happens.

gentoo-sources (manual)

Raw kernel source. You configure and compile by hand. Maximum control and understanding. THIS DECK.

gentoo-kernel (distro)

Built and managed by emerge using a sane default config. Compiles on your machine, updates like any package.

gentoo-kernel-bin (binary)

Pre-compiled. No build wait at all. Closest to other distros. Great fallback or for slow machines.

genkernel (helper)

A tool that automates config + build + initramfs from source. A middle ground between manual and distro.

Install the sources & set the symlink

gentoo-sources unpacks into /usr/src. The /usr/src/linux symlink tells every build tool which tree is 'current'.

```
# 1. Install the kernel source tree
emerge --ask sys-kernel/gentoo-sources

# 2. List installed kernel sources (the * marks the active symlink)
eselect kernel list
#   [1]   linux-6.12.x-gentoo
#   [2]   linux-6.16.x-gentoo *

# 3. Point /usr/src/linux at the version you want to build
eselect kernel set 2

# 4. This is now where all build commands run:
cd /usr/src/linux
```

Why: *Why the symlink matters: tools like 'make', module builds and external drivers (e.g. NVIDIA) all look at /usr/src/linux. Repoint it whenever you switch kernel versions.*

02

SECTION 02 · START SMART

make localmodconfig

Don't configure 12,000 options from zero. Start from what your machine actually uses.

A distro .config enables everything

Copying an ArchISO / generic .config gives you a working but enormous starting point.

What you inherit

- Drivers for thousands of devices you will never own
- Every filesystem, every network protocol, every obscure NIC
- A build that can take 30–90+ minutes
- A huge kernel image and hundreds of modules

What you actually want

- Only the drivers your real hardware loads
- A build measured in minutes, not hours
- A lean image that is easy to reason about
- A starting point you can trim further by hand

lsmod already knows your hardware

Keep only what is loaded right now.

When Linux boots, udev loads a driver for each device it detects (matched by PCI/USB IDs). The output of lsmod is therefore a precise list of what THIS machine needs. 'make localmodconfig' reads that list and switches off every module that is not currently loaded — turning a giant config into a tailored one automatically.

Running localmodconfig

Run it inside /usr/src/linux. It reads lsmod from the running system (or a saved file) and rewrites .config.

```
# Simplest form: trim based on THIS running system's loaded modules
cd /usr/src/linux
make localmodconfig

# It uses an existing .config as the base. Seed one first if needed:
zcat /proc/config.gz > .config       # config of the currently running kernel
# (requires CONFIG_IKCONFIG_PROC=y in the running kernel)

# Cross-machine: capture lsmod on machine A, trim on machine B
lsmod > /tmp/mylsmod                  # on the target machine
make LSMOD=/tmp/mylsmod localmodconfig # on the build machine
```

Why: *localmodconfig never invents a config from nothing — it always trims an existing .config down. Always start from a base that already boots.*

It only sees hardware active right now

This is the one trap. Anything not loaded at scan time gets disabled — and then its driver is missing.

⚠ Plug in everything first

- Insert every USB device you use (drives, dongles, audio)
- Connect the network interface(s) you rely on
- Mount the filesystems you need so their modules load
- Then — and only then — run localmodconfig

localmodconfig vs localyesconfig

- localmodconfig: keep needed drivers as MODULES (M)
- localyesconfig: build those same drivers INTO the kernel (Y)
- Yes-config can reduce the need for an initramfs (more later)
- Mod-config keeps the image smaller and more flexible

Why localmodconfig prompts you

After trimming, the tool may still ask you about individual options. Here is why — it bridges directly into the next section.

- Your base .config and the kernel you are building are rarely the same version.
- A newer kernel introduces NEW config symbols that your old .config never answered.
- When a symbol has no recorded answer and no safe default, the tool must ask you.
- So you will see a stream of [Y/n/m/?] prompts — these are not errors, they are genuine choices.
- The next section decodes the prompts you are most likely to meet.

03

SECTION 03 · THE QUESTIONS

Decoding the Prompts

Every [Y/n/m/?] question explained — and a strategy for the ones we don't cover.

Never panic — the '?' always helps

[Y/n/m/?]

Each letter is a possible answer. The UPPERCASE letter is the default (just press Enter to accept it). Y = build into the kernel, n = leave it out, m = build as a loadable module, ? = print the help text for this exact option. When you don't recognize a symbol, press ? first — it explains what the option does before you commit.

Built-in (Y) vs Module (M) vs Off (N)

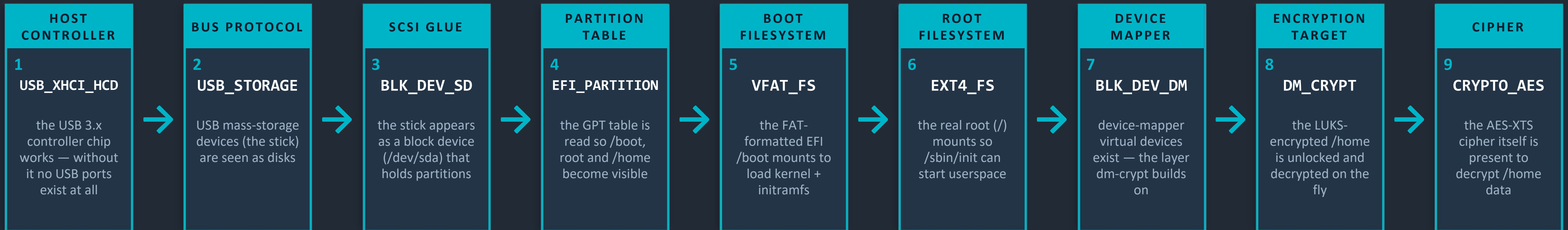
This single choice, repeated for every driver, is the heart of kernel configuration.

Choice	What it does	Trade-off	Good for
Y (built-in)	Compiled directly into the kernel image (bzImage)	Always present, loads earliest; grows the image; needs rebuild to remove	Anything required to reach root: disk, filesystem, crypto
M (module)	Compiled as a separate .ko, loaded on demand	Flexible, smaller image; not available until rootfs is mounted	Optional / hot-plug hardware: USB gadgets, extra NICs
N (off)	Not compiled at all	Smallest kernel; the feature simply doesn't exist	Hardware you don't have, features you won't use

Drivers Form a Chain — Each Link Needs the One Before It

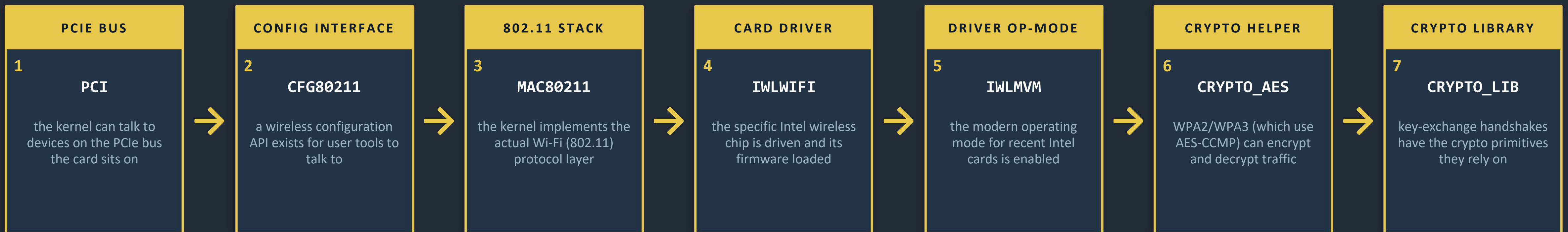
To reach a target (mount root, unlock /home) the kernel needs EVERY link below — built in (Y) for the early links before root is mounted, or supplied by the initramfs. Miss one early link and the whole chain fails; the system can't boot.

Chain A — Boot a USB stick with /boot, root (/), and encrypted /home *goal: power on → fully running encrypted system*



✓ **RESULT:** USB powers up → partitions seen → /boot loads kernel → root mounts → /home decrypts. Drop ANY link and boot stops there.

Chain B — Get Wi-Fi working on an Intel laptop *goal: associate with an encrypted Wi-Fi network*



✓ **RESULT:** PCIe sees the card → stack + driver bring it up → crypto secures the link → you associate with the network.

When you NEED an initramfs

An initramfs (initrd) is a tiny temporary root that loads modules needed to reach the real root.

The problem

- To mount the real root, the kernel needs: the disk controller driver, the filesystem driver, and (if encrypted) dm-crypt/LUKS — plus LVM or RAID if used
- If any of those is a MODULE, it lives ON the root filesystem...
- ...which can't be mounted yet because the module that mounts it isn't loaded.
- That is the chicken-and-egg deadlock.

The rule of thumb

- Build root-critical drivers IN (Y) → you may not need an initramfs at all
- Keep them as MODULES (M) → you DO need an initramfs (e.g. dracut) to load them early
- Example machine: NVMe + XFS + LUKS root → those three must be reachable before mount
- Safest beginner path: built-in the root essentials, use an initramfs for everything else

Kernel compression type

CONFIG_KERNEL_* — how the kernel image (and often the initramfs) is compressed. The kernel self-decompresses at boot.

Option	Size	Decompress speed	When to pick
GZIP	Medium	Medium	Maximum compatibility, safe default
LZ4	Larger	Fastest	Fastest boot, when storage is plentiful
ZSTD	Small	Fast	Modern sweet spot — recommended for most
XZ	Smallest	Slow	Tiny /boot partitions, boot time less critical

Note: *Recommendation for newcomers: ZSTD. Great compression with fast decompression. If unsure, GZIP is always safe.*

RCU_BOOST

RCU = Read-Copy-Update, a lock-free mechanism that lets readers run without blocking writers. RCU_BOOST temporarily raises the priority of tasks holding up RCU on preemptible kernels.

What it solves

On low-latency / real-time (PREEMPT_RT) kernels, a low-priority task can delay RCU cleanup. Boosting prevents that stall.

Do you need it?

Only relevant when you run a preemptible / real-time kernel for audio, robotics or similar latency-critical work.

Typical desktop / server

Leave it OFF (N). The default non-RT kernel doesn't need it and it adds overhead.

Beginner answer

N — unless you are deliberately building a real-time kernel.

SHA-3 (CONFIG_CRYPTO_SHA3)

A cryptographic hash algorithm exposed through the kernel's crypto API — alongside SHA-256, SHA-512 and others.

What uses it

Integrity subsystems (IMA), some signing/verification paths, and any module that explicitly requests SHA-3. Not used by typical LUKS setups, which favor SHA-256/512.

Cost of including it

Tiny. As a module it sits unused until something asks for it.

Decision

If unsure, M (module). It is then available on demand without bloating the built-in image.

Beginner answer

M — safe middle ground for crypto algorithms you might need.

SLUB_CPU_PARTIAL

A tuning option for SLUB, the kernel's default memory (slab) allocator. It keeps per-CPU lists of partially-used memory slabs.

What it does

Lets each CPU keep partial slabs locally, so allocations avoid contending on shared locks.

Trade-off

Faster allocation and better throughput, at the cost of slightly higher memory use.

Who turns it off

Real-time kernels disable it to cut latency spikes; throughput-oriented systems keep it on.

Beginner answer

Y — keep the default for a normal desktop or server.

DNET (CONFIG_DNET)

A driver for the 'Dave DNET' Ethernet controller — an obscure chip found in a few embedded boards.

What it is

A single, specific NIC driver. Has nothing to do with your PC's normal network card.

Do you have it?

Almost certainly not. It appears only on niche embedded hardware.

Why you still see it

The configurator lists every driver in the tree, however rare, so you must say yes or no to each.

Beginner answer

N — disable it. This is the right answer for the vast majority of machines.

NETERION — and the vendor-menu pattern

CONFIG_NET_VENDOR_NETERION gates the Neterion / Exar 10-Gigabit server NIC drivers (s2io, vxge).

What it is

Enterprise 10GbE cards from a specific vendor. Found in some servers, never in typical desktops/laptops.

The pattern to learn

Networking groups drivers under 'NET_VENDOR_<name>' switches. Turn a vendor OFF and ALL its driver prompts vanish at once.

Why this is powerful

Disabling vendors you don't own is the fastest way to skip dozens of irrelevant prompts and shrink the kernel.

Beginner answer

N — unless you specifically own a Neterion/Exar card.

I2C (CONFIG_I2C)

I^2C is a low-speed bus connecting small on-board chips: sensors, EEPROMs, HDMI/DisplayPort DDC, and some touchpads.

What rides on it

Temperature/voltage sensors, monitor detection (EDID over DDC), battery controllers, many laptop touchpads (i2c-hid).

Why your lsmod showed i2c_*

GPUs use I^2C to read monitor capabilities; sensor and HID drivers depend on it. Real hardware needs it.

Built-in or module?

Often safest as Y or M. Many GPU and sensor features silently depend on I^2C being present.

Beginner answer

Y or M — keep it. It is genuinely used on most real systems.

How to answer ANY unknown prompt

You will meet options this deck never named. This four-step decision handles all of them.

1. **Do I have this hardware / need this feature?**
   If clearly not (an exotic NIC, a foreign architecture, a filesystem you'll never use) → N.

2. **Is it required to reach my root filesystem?**
   Disk controller, root filesystem, encryption/LVM/RAID → build it IN (Y), or be sure your initramfs provides it.

3. **Press ? to read the help.**
   Every option documents itself. The help text states what depends on it and what it's for. Read before deciding.

4. **When still unsure, choose M.**
   A module costs almost nothing if unused, and is there if something needs it. The safe middle ground.

Other common surprise prompts

A quick reference so unfamiliar questions don't stop you. Press ? on any of these for detail.

Prompt area	What it is	Sensible default
Preemption model	How aggressively the scheduler can interrupt tasks	Desktop: 'Low-Latency Desktop'. Server: 'No Forced Preemption'
KASAN / debug	Heavy memory-error debugging for kernel developers	N — big slowdown, only for debugging
Transparent Hugepages	Automatic large memory pages for performance	'madvise' or 'always' — default is fine
EXPERT symbols	Unlocks rarely-needed low-level options	Leave hidden unless you know you need it
Default CPU governor	Frequency scaling policy at boot	'schedutil' on modern kernels

04

SECTION 04 · THE FULL TOUR

make menuconfig

A guided walk through the main menu — every top-level entry, what's inside, and why it matters.

Driving the menuconfig interface

A curses (text) UI over the same .config. Learn six keys and you can move anywhere instantly.

```
cd /usr/src/linux
make menuconfig

#  ↑ ↓          move between options
#  Enter       enter a submenu  (entries ending in --->)
#  Esc Esc     go back up one level
#  /           SEARCH for a symbol  <- your superpower
#  ?           help for the highlighted option
#
#  Toggling an option:
#    [ ] / [*]    bool    : off / built-in            (space cycles)
#    < > / <M>    tristate: off / module / built-in   (y, m, n)
#    { }          a module forced by something selecting it
```

Why: *The '/' search is the fastest tool here: type a symbol name (e.g. NVME), and it shows the exact menu path, dependencies, and what selects it.*

Why your menu may look different

Don't worry if your screen doesn't match a tutorial exactly. The tree is conditional.

- Architecture changes the menu: x86-64, ARM and others expose different entries.
- Enabling EXPERT (in General setup) reveals extra low-level options that are normally hidden.
- Dependencies hide things: an option only appears once whatever it requires is enabled.
- Kernel version matters: new releases add, rename and reorganize symbols — this deck targets modern kernels (6.x and the upcoming 7.x line).
- Use '/' to find anything by name rather than hunting visually.

General setup

Broad kernel-wide behavior and identity. The first menu, and one of the most important.

Inside

Local version string, compression for the image, init system support (initramfs, systemd vs OpenRC deps), cgroups, namespaces, kernel .config embedding (IKCONFIG).

Why it matters

cgroups + namespaces enable containers; IKCONFIG_PROC lets you read /proc/config.gz later; the compression choice from Section 3 lives here.

Newcomer focus

Enable IKCONFIG and IKCONFIG_PROC, pick your compressor, and make sure cgroup/namespace support matches your container plans.

Processor type and features

Tell the kernel exactly what CPU it will run on — a major source of real-world speedups.

Inside

Processor family, SMP (multi-core) support, number of CPUs, vendor features (AMD/Intel), virtualization hooks, mitigations for CPU vulnerabilities, NUMA.

Why it matters

Choosing your actual CPU family lets the compiler use instructions your chip supports. SMP is essential on any multi-core machine.

Newcomer focus

Set the processor family to match your CPU (or generic x86-64), keep SMP on, leave security mitigations at defaults.

Bus options (PCI etc.)

Support for the physical buses that connect devices to the CPU.

Inside

PCI / PCI Express, PCI hotplug, message-signalled interrupts (MSI), and legacy bus types.

Why it matters

Practically every modern device — GPU, NVMe, NIC, sound — hangs off PCIe. Without PCI support, most hardware is invisible.

Newcomer focus

Keep PCI and PCIe enabled (they are by default). You rarely change anything here.

Binary Emulations + Firmware Drivers

Two smaller menus, grouped: running other-format binaries, and talking to platform firmware.

Binary Emulations

On x86-64, support for running 32-bit (IA32) binaries. Keep on if you run any 32-bit software (some games, legacy apps).

Firmware Drivers — EFI

Interfaces to UEFI/BIOS firmware: EFI variables, EFI stub (lets the kernel be booted directly by UEFI), DMI/SMBIOS info.

Newcomer focus

On a UEFI system keep EFI support and the EFI stub enabled. Keep IA32 emulation unless you are sure you need only 64-bit.

Power management and ACPI options

Sleep states, thermal control, CPU frequency scaling, and battery handling.

Inside

ACPI (the platform power/config standard), suspend-to-RAM and hibernate, CPUFreq governors, thermal and fan control.

Why it matters

Essential on laptops (battery, sleep, thermals) and useful on desktops/servers for power efficiency and frequency scaling.

Newcomer focus

Keep ACPI on, enable CPUFreq with the 'schedutil' governor, and enable suspend/hibernate if you use them.

General architecture-dependent options

Cross-cutting low-level knobs that depend on your CPU architecture.

Inside

Kernel hardening features, seccomp (syscall filtering, used by sandboxes/containers), stack protector, kprobes, and module-signing toggles.

Why it matters

seccomp is required by many container and browser sandboxes; stack protector and hardening improve security at minimal cost.

Newcomer focus

Keep seccomp enabled, keep the default stack protector, leave the rest at defaults unless hardening deliberately.

Enable loadable module support

The master switch that makes 'M' possible at all.

Inside

Module loading/unloading, forced unload, module versioning, and module signature verification.

Why it matters

Without this, EVERY driver must be built-in (Y). Out-of-tree drivers like NVIDIA also require module support to load.

Newcomer focus

Keep it ON. Enable module signature support if you care about verifying module integrity; otherwise defaults are fine.

Enable the block layer

The subsystem that handles block devices — anything you store data on in blocks.

Inside

Block device support, I/O schedulers (mq-deadline, BFQ, Kyber), partition table formats (GPT, MBR), and large-device support.

Why it matters

Disks, SSDs and NVMe are all block devices. Without the block layer there is no storage — and no root filesystem.

Newcomer focus

Keep it ON. Enable GPT partition support (modern UEFI systems) and a scheduler such as BFQ (desktop) or mq-deadline.

Memory Management options

How the kernel manages RAM, paging, and swap behavior.

Inside

The slab allocator (SLUB — see SLUB_CPU_PARTIAL from Section 3), transparent hugepages, swap, KSM (page de-duplication), memory compaction.

Why it matters

Defaults are well tuned. A few options (hugepages, KSM) matter for databases or virtualization hosts.

Newcomer focus

Accept defaults. Enable transparent hugepages on 'madvise' and leave the allocator as SLUB.

Executable file formats

Which program binary formats the kernel can load and run.

Inside

ELF (the universal Linux format), and binfmt_misc (lets the kernel dispatch other formats to interpreters — e.g. running foreign-arch binaries via QEMU).

Why it matters

ELF is mandatory — without it nothing runs. binfmt_misc enables neat tricks like transparently running ARM binaries on x86.

Newcomer focus

Keep ELF on (it is). Enable binfmt_misc if you use Docker multi-arch, Wine helpers, or cross-architecture emulation.

Networking support — core

The entire network stack lives here, starting with the protocols themselves.

Inside

TCP/IP (IPv4 & IPv6), sockets, the wireless stack (cfg80211/mac80211), Bluetooth, bridging, VLANs, traffic control/QoS.

Why it matters

This is networking itself, separate from the device drivers (which are under Device Drivers). WiFi needs cfg80211/mac80211 here.

Newcomer focus

Keep TCP/IP on. Enable the wireless stack only if you use WiFi; bridging/VLAN only if you do virtualization or advanced networking.

Networking support — Netfilter

The firewall and packet-manipulation framework — the kernel side of nftables/iptables.

Inside

nf_tables (modern), legacy iptables, connection tracking (nf_conntrack), NAT/masquerade, and the xt_/nft_ match & target modules.

Why it matters

These are exactly the nft_*, nf_nat, nf_conntrack modules seen in a typical lsmod. They power firewalls, NAT, Docker, and libvirt networking.

Newcomer focus

Enable nf_tables + connection tracking + NAT as MODULES. Your firewall tooling and container/VM networking will load them on demand.

Device Drivers

The biggest menu by far.

This is where the kernel meets your actual hardware — hundreds of submenus covering storage, graphics, networking devices, USB, input, sound and more. It dwarfs every other top-level entry combined. Because it is so large, we break it across the next four slides by category. This is also where localmodconfig saves you the most: it disables the thousands of device drivers you'll never touch.

Device Drivers — storage (root-critical)

The drivers that let the kernel see the disk your system lives on. Get these wrong and it will not boot.

NVMe

Under 'NVME Support'. Modern M.2 / PCIe SSDs. If your root is on NVMe, this MUST be reachable at boot (built-in, or in the initramfs).

SATA / AHCI

Under 'Serial ATA and Parallel ATA drivers'. Traditional SSDs and hard drives.

USB storage

usb-storage / uas, for external drives and bootable USB sticks. Useful to keep even if not your root device.

Rule

Whatever holds your root filesystem: build it IN, or guarantee your initramfs loads it. This is the #1 cause of unbootable custom kernels.

Device Drivers — Graphics (DRM)

Under 'Graphics support'. The DRM (Direct Rendering Manager) subsystem drives your display.

In-tree drivers

amdgpu (AMD), i915/xe (Intel), nouveau (open NVIDIA). Enable the one matching your GPU, plus framebuffer/console support.

Out-of-tree NVIDIA

The proprietary nvidia modules are NOT in this menu — they come from x11-drivers/nvidia-drivers via emerge and build against your kernel.

What the kernel still needs

Even for proprietary NVIDIA you must enable DRM and the right low-level options (e.g. CONFIG_DRM) so the external module can attach.

Newcomer focus

Enable DRM + your GPU's in-tree driver (or DRM-only if using proprietary NVIDIA). Don't enable nouveau and proprietary NVIDIA together.

Device Drivers — network, USB, HID, I²C

The everyday peripheral drivers — this is what localmodconfig trims most aggressively.

Network device drivers

Organized by vendor (the NET_VENDOR_* pattern from Section 3). Enable only your NIC's vendor; disable the rest to skip dozens of prompts.

USB support

xHCI/EHCI host controllers (needed for USB ports at all), plus per-device drivers. Keep host controller support on.

HID (input)

Keyboards, mice, touchpads, game controllers. i2c-hid covers many laptop touchpads; generic USB HID covers most peripherals.

I²C

The bus from Section 3 — sensors, EEPROMs, monitor DDC. Enable it; GPU and sensor features depend on it.

File systems

Which filesystems the kernel can mount — including, critically, the one your root lives on.

Common choices

ext4 (default everywhere), XFS (high-performance, scalable), Btrfs (snapshots, checksums), F2FS (flash-optimized).

The EFI partition

Enable VFAT/FAT — the UEFI System Partition is FAT-formatted. Without it, /boot/EFI can't be mounted.

Pseudo filesystems

Keep /proc, /sys (sysfs), tmpfs and devtmpfs — userspace and the boot process depend on them.

Root rule

Build your ROOT filesystem driver IN (e.g. XFS=Y) so the kernel can mount root even without an initramfs.

Security options

Access-control frameworks and integrity features layered on top of standard Unix permissions.

Inside

Linux Security Modules (LSM): SELinux, AppArmor, Smack, TOMOYO; plus IMA/EVM integrity measurement and the kernel keyring.

Why it matters

Gentoo hardened profiles use SELinux; many distros use AppArmor. The choice affects mandatory access control policy.

Newcomer focus

Defaults are safe. If you don't run a specific MAC policy, you can leave SELinux/AppArmor disabled — but keep the keyring and basic LSM support.

Cryptographic API

The kernel's library of ciphers and hashes — used by disk encryption, module signing, and the network stack.

Inside

Ciphers (AES, ChaCha20), hashes (SHA-256/512, and SHA-3 from Section 3), and hardware acceleration (AES-NI on x86, ARM crypto extensions).

Why it matters

dm-crypt/LUKS depends on having the right cipher + hash here. AES-NI makes encrypted disks fast — enable it on modern CPUs.

Newcomer focus

Enable AES, SHA-256/512 and AES-NI acceleration. These cover LUKS and most everyday cryptography.

The remaining menus

Smaller top-level entries, grouped. Most newcomers leave these at defaults.

Library routines

Shared helper code (CRC algorithms, compression libs) auto-selected by drivers. You rarely set these by hand.

Kernel hacking

Developer debugging: KGDB, dynamic debug, KASAN, lockdep, ftrace. Powerful but heavy — keep mostly off for production.

Virtualization

KVM (hardware-assisted VMs) for AMD/Intel. Enable if you run QEMU/libvirt; these are the kvm/kvm_amd modules from lsmod.

Sound / Device tree / others

ALSA sound (sound cards, HDMI audio), plus arch-specific entries. Enable sound as modules if you want audio.

What MUST be right or it won't boot

Storage · Filesystem · Crypto · initramfs

Across the whole menu, four things decide whether your kernel boots. (1) The driver for the disk holding root (e.g. NVMe). (2) The root filesystem driver (e.g. XFS). (3) If encrypted, the crypto + dm-crypt support (LUKS). (4) An initramfs if any of those is a module instead of built-in. Get these four right and the rest is optimization.

05

SECTION 05 · BUILD & INSTALL

Compiling & Installing

From a finished .config to a working boot entry.

make — building the kernel

Compilation is parallelizable. The -j flag sets how many jobs run at once.

```
cd /usr/src/linux

# Build using one job per CPU thread (nproc reports the count)
make -j$(nproc)

#  -j N   run N compile jobs in parallel
#  nproc  prints your number of CPU threads, e.g. 16
#
# Expect anywhere from a couple of minutes (lean config, fast CPU)
# to much longer (a fat distro-style config). This is exactly why
# we trimmed with localmodconfig first.
```

Why: *If the build fails, the error usually names a missing tool or an unmet config dependency — read the last few lines; they are specific.*

make modules_install

Installs every module you built (the M choices) into the standard location, where the kernel finds them at runtime.

```
make modules_install

# Installs to:
#   /lib/modules/<kernel-version>/
#
# This must run BEFORE you rely on any modular driver, and the
# version directory must match the kernel image you boot. An
# initramfs (built next) pulls the early-boot modules from here.
```

Why: *Order matters: install modules before generating an initramfs, so the initramfs can copy the modules it needs from /lib/modules.*

Copying the kernel image to /boot

The compiled kernel for x86 is called bzImage. You can let 'make install' place it, or copy it manually to understand the layout.

```
# Make sure /boot is mounted first (often a separate partition!)
mount /boot      # if it isn't already

# The freshly built image lives here on x86/x86-64:
#   arch/x86/boot/bzImage

# Manual copy with a versioned name (recommended for clarity):
cp arch/x86/boot/bzImage  /boot/vmlinuz-6.16.0-gentoo
cp System.map             /boot/System.map-6.16.0-gentoo
cp .config                /boot/config-6.16.0-gentoo

# ...or let the kernel's installer do it for you:
make install
```

Why: *Versioned names let multiple kernels coexist in /boot, so you always keep a known-good fallback to boot if the new one misbehaves.*

Generating an initramfs (when needed)

Recall Section 3: if a root-critical driver is a module, you need an initramfs to load it early. dracut is the common tool.

```
# Install dracut once
emerge --ask sys-kernel/dracut

# Generate an initramfs for a specific kernel version
dracut --kver=6.16.0-gentoo

# This produces, e.g.:
#   /boot/initramfs-6.16.0-gentoo.img
#
# dracut auto-detects what's needed to reach root (NVMe, XFS,
# dm-crypt/LUKS, LVM...) and bundles those modules from
# /lib/modules. That's why modules_install runs first.
```

Why: *If you built every root-critical driver in (Y), you may be able to skip the initramfs entirely — but one is required for LUKS, LVM or RAID roots.*

Updating GRUB — the easy way

GRUB can scan /boot and write boot entries automatically, detecting your kernels and matching initramfs images.

```
# Regenerate the GRUB config (detects kernels + initramfs in /boot)
grub-mkconfig -o /boot/grub/grub.cfg

# Expected output mentions each kernel it found, e.g.:
#   Found linux image: /boot/vmlinuz-6.16.0-gentoo
#   Found initrd image: /boot/initramfs-6.16.0-gentoo.img
#
# Reboot and pick the new entry from the GRUB menu.
```

Why: *Always keep your previous working kernel as an entry. If the new kernel won't boot, pick the old one and fix the config — no reinstall needed.*

A manual GRUB entry, explained

Understanding the entry helps when auto-detection misses something. Each line has a clear job.

```
menuentry 'Gentoo 6.16.0 (custom)' {
    # which disk/partition GRUB reads the kernel from
    set root=(hd0,gpt2)

    # the kernel image + parameters passed to it
    linux /vmlinuz-6.16.0-gentoo root=/dev/nvme0n1p3 rootfstype=xfs rw

    # the initramfs to load alongside the kernel
    initrd /initramfs-6.16.0-gentoo.img
}

#  root=          which device holds the real root filesystem
#  rootfstype=    filesystem type of that root (helps early mount)
#  rw             mount root read-write at boot
```

Why: *For an encrypted root you'd add a parameter like rd.luks.uuid=... so the initramfs knows which LUKS device to unlock before mounting root.*

06

SECTION 06 · RECAP

Recap & Next Steps

The whole journey on one page — plus where to go from here.

The complete workflow

Every step from bare sources to a booted custom kernel.

1. **Get sources**
   emerge gentoo-sources; set /usr/src/linux with eselect kernel.
2. **Seed a config**
   Start from a working .config, then trim with make localmodconfig.
3. **Answer prompts**
   Decode [Y/n/m/?]; build root-critical drivers IN, use M when unsure.
4. **Refine**
   make menuconfig; verify storage, filesystem, crypto; disable unused vendors.
5. **Compile**
   make -j$(nproc).
6. **Install modules**
   make modules_install → /lib/modules/<ver>.
7. **Install image**
   Copy bzImage to /boot (or make install).
8. **initramfs + GRUB**
   dracut if needed; grub-mkconfig -o /boot/grub/grub.cfg; reboot.

Common boot failures & fixes

Almost every failed custom kernel comes down to one of these.

Symptom	Likely cause	Fix
Kernel panic: 'unable to mount root'	Disk or filesystem driver missing/modular with no initramfs	Build NVMe/SATA + root FS driver IN, or add them to the initramfs
'No init found'	Root mounted but wrong device, or rootfs lacks /sbin/init	Check root= in GRUB; verify the correct partition
Boots, but no encrypted root	dm-crypt/LUKS support missing from early boot	Ensure crypto + dm-crypt in initramfs; add rd.luks.uuid= parameter
Hangs early with modular drivers	Forgot the initramfs entirely	Generate one with dracut and reference it in the GRUB entry

Going further

Resources and gentler paths once you've done it the manual way.

Gentoo Wiki

wiki.gentoo.org — the Handbook and Kernel/Configuration pages are authoritative and kept current.

genkernel

Automates configuration, compilation and initramfs from source — a faster middle ground.

Kernel Seeds

kernelseeds.com — generates a lean, sensible starting .config you can refine.

Distribution kernels

gentoo-kernel / gentoo-kernel-bin — perfectly fine for daily use once you've learned the manual process.

THE END

You can build a kernel anywhere now

Sources → .config → bzImage → boot.

The manual Gentoo path teaches the real mechanics: trimming with localmodconfig, decoding each prompt, walking menuconfig, and installing the result. That knowledge transfers to every Linux distribution — because under all of them is the same kernel, waiting for you to build it.