Raspberry Pi 5 Driver Development Setup Guide

Complete A-Z Setup for Linux Kernel 6.6 Development

Author: Based on successful setup by Earnest
Date: February 10, 2026
Target Hardware: Raspberry Pi 5
Host OS: Ubuntu 22.04/24.04 LTS
Kernel Version: Linux 6.6.78 (LTS)
Architecture: 64-bit ARM (aarch64)

Table of Contents

Overview

This guide walks you through setting up a complete embedded Linux driver development environment for Raspberry Pi 5. You'll build a custom Linux kernel 6.6.78, cross-compile kernel modules on your PC, and deploy them to the Pi 5 for testing.

What You'll Achieve

- Cross-compilation environment on your PC
- Custom Linux kernel 6.6.78 for Pi 5
- Ability to build and test kernel modules
- Ready to follow Linux driver development tutorials

Hardware Requirements

Essential

- **Raspberry Pi 5** (any RAM variant)

- **MicroSD card** (16GB minimum, 32GB+ recommended, Class 10 or better)
- **USB-C power supply** (5V/5A recommended for Pi 5)
- **Ethernet cable** (for initial setup and development)
- **Development PC** running Ubuntu 22.04 or 24.04 LTS

Optional

- HDMI monitor + USB keyboard (useful for troubleshooting)
- SD card reader/adapter

Part 1: Host Machine Setup (Your Development PC)

Step 1: Update Your System

```
sudo apt update && sudo apt upgrade -y
```

Time estimate: 5-15 minutes depending on updates

Step 2: Install Core Build Dependencies

```
sudo apt install -y \
  git \
  bc \
  bison \
  flex \
  libssl-dev \
  make \
  libc6-dev \
  libncurses5-dev \
  build-essential
```

Why these packages: - `git` - Clone kernel source - `bc`, `bison`, `flex` - Build tools - `libssl-dev` - Crypto support - `libncurses5-dev` - For menuconfig UI

Step 3: Install 64-bit Cross-Compiler Toolchain

```
sudo apt install -y crossbuild-essential-arm64
```

Verify installation:

```
aarch64-linux-gnu-gcc --version
```

Expected output:

```
aarch64-linux-gnu-gcc (Ubuntu ...) 11.x.x or later
```

CRITICAL: Pi 5 requires **arm64** (64-bit), NOT armhf (32-bit)

Step 4: Create Working Directory

```
mkdir -p ~/rpi-driver-dev
cd ~/rpi-driver-dev
```

This keeps all your development files organized in one place.

Step 5: Clone Kernel Source for Pi 5

```
cd ~/rpi-driver-dev
git clone --depth=1 -b rpi-6.6.y https://github.com/raspberrypi/linux
cd linux
```

Time estimate: 5-10 minutes (downloads ~200MB)

Why rpi-6.6.y branch: - Long Term Support (LTS) until December 2026 - Stable and well-tested for Pi 5 - Better Pi 5 hardware support than older kernels

---

Step 6: Configure Kernel for Pi 5

```
export KERNEL=kernel_2712
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- bcm2711_defconfig
```

Expected output:

```
#
# configuration written to .config
#
```

Note: We use `bcm2711_defconfig` which works for Pi 3, 4, and 5 in 64-bit mode. The Pi 5's BCM2712 SoC is supported through device tree files.

Verify config was created:

```
ls -la .config
```

You should see a `.config` file with a recent timestamp.

Step 7: Customize Kernel Configuration (menuconfig)

```
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- menuconfig
```

A text-based menu will appear.

Navigation: - **Arrow keys:** Navigate menus - **Enter:** Select/enter submenu - **Space:** Toggle option (`< >` → `<M>` → `<*>`) - **ESC ESC:** Go back/exit - **/** (slash):** Search for options

---

Enable Required Features for Driver Development: 1. SPI Support:

```
Device Drivers --->
    [*] SPI support --->
        <M> User mode SPI device driver support
```

2. UIO (Userspace I/O):

```
Device Drivers --->
    <M> Userspace I/O drivers --->
        <M> Userspace I/O platform driver with generic IRQ handling
```

3. Industrial I/O (IIO):

```
Device Drivers --->
    <M> Industrial I/O support --->
        -*- Enable buffer support within IIO
        <M>   Industrial I/O buffering based on kfifo
```

4. Input Subsystem:

```
Device Drivers --->
    Input device support --->
        [*] Miscellaneous devices --->
            <M> User level driver support
```

5. LED Support:

```
Device Drivers --->
    [*] LED Support --->
        <M>   LED Support for GPIO connected LEDs
```

6. I2C (verify it's enabled):

```
Device Drivers --->
    -*- I2C support --->
        <*>   I2C device interface
```

Symbol meanings: - `< >` = Not selected - `<M>` = Module (loadable .ko file) - **Use this for most features** - `<*>` = Built-in (compiled into kernel) - `[*]` = Feature enabled - `-*-` = Required dependency (auto-selected)

When done: 1. Press **ESC ESC** 2. Select **Yes** to save 3. Exit to terminal

---

Step 8: Build the Kernel

```
make -j$(nproc) \
  ARCH=arm64 \
  CROSS_COMPILE=aarch64-linux-gnu- \
  Image modules dtbs
```

What this does: - `make -j$(nproc)` - Use all CPU cores for faster compilation - `Image` - Build kernel image - `modules` - Build loadable kernel modules - `dtbs` - Build device tree binaries (hardware descriptions)

Time estimate: - 4-core CPU: ~60-90 minutes - 8-core CPU: ~30-45 minutes - 16+ core CPU: ~15-25 minutes

Check CPU cores:

```
nproc
```

Expected output: Lots of scrolling compilation messages. When complete, you'll return to the command prompt with no errors.

Step 9: Verify Build Success

Check 1: Kernel image exists

```
ls -lh arch/arm64/boot/Image
```

Expected: File size ~25-35 MB

Check 2: Device tree files exist

```
ls arch/arm64/boot/dts/broadcom/bcm2712*.dtb
```

Expected output:

```
arch/arm64/boot/dts/broadcom/bcm2712-rpi-5-b.dtb
arch/arm64/boot/dts/broadcom/bcm2712-rpi-cm5-cm4io.dtb
arch/arm64/boot/dts/broadcom/bcm2712-rpi-cm5-cm5io.dtb
```

Check 3: Modules were built

```
find . -name "*.ko" | wc -l
```

Expected: 1000-2000+ (number of kernel modules)

Check 4: Note your kernel version

```
make kernelrelease
```

Expected output:

```
6.6.78-v8-16k+
```

IMPORTANT: Write this down! You'll need it for module compatibility checks.

Part 2: Prepare SD Card

Step 10: Flash Raspberry Pi OS

Install Raspberry Pi Imager:

```
# Via snap (recommended - latest version):
sudo snap install rpi-imager

# OR via apt:
sudo apt install rpi-imager
```

Launch Imager:

```
rpi-imager
```

In the Imager GUI:

1. **Choose Device:** Select **"Raspberry Pi 5"**
2. **Choose OS:** Select **"Raspberry Pi OS (64-bit)"**
   - Use the full desktop version OR
   - Use "Raspberry Pi OS Lite (64-bit)" for headless/minimal setup
3. **Choose Storage:** Select your SD card
   - **DOUBLE CHECK** - wrong selection = data loss!
4. **Click "Next"**
5. **Click "EDIT SETTINGS"** (very important!)

Configure OS Customization:

General Tab:

```
Set hostname: raspberrypi5
Set username and password:
 Username: pi
 Password: [your choice - remember this!]
Configure wireless LAN: (optional if using Ethernet)
 SSID: [your WiFi name]
 Password: [your WiFi password]
 Country: [your country code]
Set locale settings:
```

```
Time zone: [your timezone]
Keyboard layout: [your layout]
```

Services Tab:

```
Enable SSH
  Use password authentication
```

Click "SAVE"

---

6. **Click "YES"** to apply OS customization
7. **Click "YES"** to confirm erase
8. **Wait 5-15 minutes** for write and verification

When complete: Click "CONTINUE" and safely eject the SD card

Step 11: Enable SSH (Manual Method - If Imager Customization Fails)

If SSH doesn't work after flashing, you can enable it manually:

Re-insert the SD card into your PC after flashing.

Wait for partitions to mount, then:

```
# Create SSH enable file:
sudo touch /media/$USER/bootfs/ssh

# Verify:
ls -la /media/$USER/bootfs/ssh
```

Expected:

```
-rw-r--r-- 1 user user 0 Feb 10 XX:XX /media/$USER/bootfs/ssh
```

Step 12: Create User Account (Manual Method - If Login Fails)

If you can't login with the password you set, create the user manually:

Generate password hash:

```
echo 'your_password' | openssl passwd -6 -stdin
```

Copy the output hash (long string starting with `$6$...`)

Create userconf.txt:

```
echo 'pi:PASTE_HASH_HERE' | sudo tee /media/$USER/bootfs/userconf.txt
```

Replace `PASTE_HASH_HERE` with the actual hash from the previous command.

Example (password "raspberry"):

```
echo 'raspberry' | openssl passwd -6 -stdin
# Copy output, then:
echo 'pi:$6$x2ZoVGuao0yPk548$oxHFyvofhGJ0Fzb8CjqLqRtg4S7QA4/l8cl/AgxWwDHRlsVYe1xVcdcJw2wLSmc
```

Step 13: Identify SD Card Partitions

Before copying your custom kernel, identify where the SD card is mounted:

```
lsblk
```

Look for your SD card:

Example output:

```
NAME        MAJ:MIN RM   SIZE TYPE MOUNTPOINTS
sda           8:0    1  14.6G disk
 sda1         8:1    1   512M part /media/earnest/bootfs
 sda2         8:2    1   5.5G part /media/earnest/rootfs
```

Note your mount points: - Boot partition: `/media/YOUR_USER/bootfs` (or `/boot` or `/bootfs`) - Root partition: `/media/YOUR_USER/rootfs` (or `/root` or `/rootfs`)

Your device and mount points may be different! Adjust commands accordingly.

For this guide, we'll use: - Boot: `/media/earnest/bootfs` - Root: `/media/earnest/rootfs`

Replace with your actual paths.

Step 14: Install Kernel Modules to SD Card

```
cd ~/rpi-driver-dev/linux

sudo env PATH=$PATH make \
  ARCH=arm64 \
  CROSS_COMPILE=aarch64-linux-gnu- \
  INSTALL_MOD_PATH=/media/earnest/rootfs \
  modules_install
```

Replace `/media/earnest/rootfs` with YOUR root partition path

Time estimate: 2-5 minutes

Expected output: Lots of lines showing module installation:

```
  INSTALL /media/earnest/rootfs/lib/modules/6.6.78-v8-16k+/kernel/...
  DEPMOD  /media/earnest/rootfs/lib/modules/6.6.78-v8-16k+
```

Step 15: Backup Original Kernel & Install Custom Kernel

Backup the stock kernel:

```
sudo cp /media/earnest/bootfs/kernel_2712.img \
       /media/earnest/bootfs/kernel_2712.img.backup
```

Copy your custom kernel:

```
sudo cp arch/arm64/boot/Image \
       /media/earnest/bootfs/kernel_2712.img
```

Copy device tree files:

```
sudo cp arch/arm64/boot/dts/broadcom/bcm2712*.dtb \
       /media/earnest/bootfs/
```

Copy device tree overlays:

```
sudo cp arch/arm64/boot/dts/overlays/*.dtb* \
       /media/earnest/bootfs/overlays/

sudo cp arch/arm64/boot/dts/overlays/README \
       /media/earnest/bootfs/overlays/
```

Again, replace `/media/earnest/bootfs` with YOUR boot partition path

Step 16: Save Kernel Version String

```
make kernelrelease > ~/kernel_version.txt
cat ~/kernel_version.txt
```

Expected output:

```
6.6.78-v8-16k+
```

Save this - you'll need it for verifying module compatibility.

Step 17: Safely Unmount SD Card

```
sync
sync
sync
```

```
sudo umount /media/earnest/bootfs
sudo umount /media/earnest/rootfs
```

Wait for commands to complete, then physically remove the SD card.

Part 3: First Boot and Configuration

Step 18: Boot Raspberry Pi 5

1. **Insert SD card** into Raspberry Pi 5
2. **Connect Ethernet cable** from Pi to your router/switch
3. **Connect power** (USB-C, 5V/5A recommended)
4. **Wait 1-2 minutes** for first boot

LED indicators: - Red LED: Power - Green LED: Activity (should blink during boot)

Step 19: Find Pi's IP Address

Method 1: Check your router (Easiest)

1. Access router web interface (usually `192.168.1.1` or `192.168.0.1`)
2. Find "Connected Devices" or "DHCP Clients"
3. Look for:
   - Device named `raspberrypi5` (your hostname)
   - Device with MAC starting with `D8:3A:DD` or `DC:A6:32` (Raspberry Pi)
4. Note the IP address

Method 2: Use hostname (If mDNS works on your network)

```
ping raspberrypi5.local
```

Method 3: Network scan

```
# Install nmap:
sudo apt install nmap

# Scan (adjust IP range for your network):
nmap -sn 192.168.1.0/24 | grep -B 2 "Raspberry"
```

Example result:

```
Nmap scan report for raspberrypi5.local (192.168.1.121)
```

Step 20: SSH into Pi 5

```
ssh pi@192.168.1.121  # Use YOUR Pi's IP address
```

First connection:

```
The authenticity of host '192.168.1.121' can't be established.
ED25519 key fingerprint is SHA256:...
Are you sure you want to continue connecting (yes/no)?
```

Type: `yes` and press **Enter**

Enter your password when prompted.

If SSH refuses connection or password fails, see the Troubleshooting section.

Step 21: Verify Custom Kernel is Running

```
uname -r
```

Expected output:

```
6.6.78-v8-16k+
```

This should match the output from `make kernelrelease` earlier.

Also verify architecture:

```
uname -m
```

Expected output:

```
aarch64
```

Check full kernel info:

```
uname -a
```

Expected output:

```
Linux raspberrypi 6.6.78-v8-16k+ #1 SMP PREEMPT [your build date] aarch64 GNU/Linux
```

If you see `6.6.78-v8-16k+`, your custom kernel is running successfully!

If you see a different version (like `6.12.x`), the stock kernel is running. You'll need to re-copy the kernel files to the SD card.

Part 4: Test Your Setup

Step 22: Create a Test Kernel Module

On your development PC:

```
mkdir -p ~/rpi-driver-dev/test_module
cd ~/rpi-driver-dev/test_module
```

Create `hello.c`:

```
cat > hello.c << 'EOF'
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

static int __init hello_init(void) {
    pr_info("Hello from Pi 5 with custom kernel!\n");
    return 0;
}

static void __exit hello_exit(void) {
    pr_info("Goodbye from Pi 5!\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("Test module for Pi 5 driver development");
MODULE_VERSION("1.0");
EOF
```

Create `Makefile`:

```
cat > Makefile << 'EOF'
KERNEL_SRC := $(HOME)/rpi-driver-dev/linux

obj-m += hello.o

all:
    make ARCH=arm64 \
      CROSS_COMPILE=aarch64-linux-gnu- \
      -C $(KERNEL_SRC) \
      M=$(PWD) modules

clean:
    make ARCH=arm64 \
      CROSS_COMPILE=aarch64-linux-gnu- \
      -C $(KERNEL_SRC) \
```

```
	M=$(PWD) clean
EOF
```

Step 23: Build the Test Module

```
make
```

Expected output:

```
make ARCH=arm64 \
  CROSS_COMPILE=aarch64-linux-gnu- \
  -C /home/earnest/rpi-driver-dev/linux \
  M=/home/earnest/rpi-driver-dev/test_module modules
...
  CC [M]  .../test_module/hello.o
  MODPOST .../test_module/Module.symvers
  CC [M]  .../test_module/hello.mod.o
  LD [M]  .../test_module/hello.ko
```

Verify module was created:

```
ls -lh hello.ko
```

Expected: `hello.ko` file (~5-10 KB)

Step 24: Deploy to Raspberry Pi 5

Copy module to Pi:

```
scp hello.ko pi@192.168.1.121:~/
```

Enter your password when prompted.

Expected:

```
hello.ko                100%   6192   2.8MB/s   00:00
```

Step 25: Test the Module on Pi

SSH to the Pi:

```
ssh pi@192.168.1.121
```

Load the module:

```
sudo insmod hello.ko
```

Check kernel log:

```
dmesg | tail -5
```

Expected output:

```
[  XXX.XXXXXX] hello: loading out-of-tree module taints kernel.
[  XXX.XXXXXX] Hello from Pi 5 with custom kernel!
```

Note: "taints kernel" is normal for out-of-tree modules.

Unload the module:

```
sudo rmmod hello
```

Check log again:

```
dmesg | tail -5
```

Expected output:

```
[  XXX.XXXXXX] Hello from Pi 5 with custom kernel!
[  XXX.XXXXXX] Goodbye from Pi 5!
```

If you see both messages - SUCCESS!

Your development environment is fully operational and ready for driver development!

Development Workflow

Standard Workflow for Each Lab/Project

1. On Development PC - Create driver source:

```
cd ~/rpi-driver-dev
mkdir lab_X
cd lab_X
# Create your .c file and Makefile
```

2. Build the module:

```
make
```

3. Copy to Pi:

```
scp your_module.ko pi@192.168.1.121:~/
```

4. On Pi - Load and test:

```
ssh pi@192.168.1.121
sudo insmod your_module.ko
dmesg | tail -20
# Test functionality...
sudo rmmod your_module
```

5. Iterate: - Edit code on PC - Rebuild (`make`) - Copy to Pi - Test again

---

Useful Commands Reference

On Development PC:

```
# Clean build files:
make clean

# Rebuild just modules (after kernel config change):
cd ~/rpi-driver-dev/linux
make -j$(nproc) ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- modules

# Check module info before deploying:
modinfo your_module.ko
```

On Raspberry Pi:

```
# View recent kernel messages:
dmesg | tail -20

# View all messages from a module:
dmesg | grep your_module

# List loaded modules:
lsmod

# Get detailed module info:
modinfo module_name

# Force remove module (if stuck):
sudo rmmod -f module_name

# Check kernel version:
uname -r

# Monitor kernel log in real-time:
sudo dmesg -w
```

Troubleshooting

Issue: SSH Connection Refused

Symptoms:

```
ssh: connect to host 192.168.1.121 port 22: Connection refused
```

Solution 1: Enable SSH on SD card

1. Power off Pi, remove SD card
2. Insert into PC
3. Create SSH enable file:

   ```
   sudo touch /media/$USER/bootfs/ssh
   ```

4. Unmount, reboot Pi

Solution 2: Check if SSH service is running

If you have monitor access:

```
sudo systemctl status ssh
sudo systemctl enable ssh
sudo systemctl start ssh
```

Issue: SSH Permission Denied (Wrong Password)

Symptoms:

```
pi@192.168.1.121: Permission denied (publickey,password)
```

Solution: Create user manually on SD card

1. Power off Pi, remove SD card, insert into PC
2. Generate password hash:

   ```
   echo 'your_password' | openssl passwd -6 -stdin
   ```

3. Create userconf.txt:

   ```
   echo 'pi:HASH_FROM_STEP_2' | sudo tee /media/$USER/bootfs/userconf.txt
   ```

4. Enable password authentication:

   ```
   sudo sed -i 's/#PasswordAuthentication yes/PasswordAuthentication yes/' /media/$USER/rc
   ```

5. Ensure SSH is enabled:

   ```
   sudo touch /media/$USER/bootfs/ssh
   ```

6. Fix home directory ownership:

   ```
   sudo chown -R 1000:1000 /media/$USER/rootfs/home/pi
   ```

7. Sync and unmount:

```
sync
sudo umount /media/$USER/bootfs
sudo umount /media/$USER/rootfs
```

8. Reboot Pi and try again

Issue: Stock Kernel Running Instead of Custom Kernel

Symptoms:

```
uname -r
# Shows: 6.12.x or 6.1.x instead of 6.6.78
```

Solution: Re-copy custom kernel

1. Power off Pi, remove SD card
2. Insert into PC, wait for mount
3. Re-copy kernel files:

```
cd ~/rpi-driver-dev/linux
sudo cp arch/arm64/boot/Image /media/$USER/bootfs/kernel_2712.img
sudo cp arch/arm64/boot/dts/broadcom/bcm2712*.dtb /media/$USER/bootfs/
sync
sudo umount /media/$USER/bootfs
sudo umount /media/$USER/rootfs
```

4. Boot Pi and verify: `uname -r`

Issue: Module Version Mismatch

Symptoms:

```
insmod: ERROR: could not insert module hello.ko: Invalid module format
dmesg shows: version magic mismatch
```

Solution:

Module was compiled for different kernel version.

Check module version:

```
modinfo hello.ko | grep vermagic
```

Check running kernel:

```
uname -r
```

If they don't match: 1. Ensure you're using the correct kernel source when building 2. Rebuild the module: `bash make clean make` 3. Verify `KERNEL_SRC` in Makefile points to `~/rpi-driver-dev/linux`

Issue: Cannot Find Pi on Network

Solutions:

1. **Wait longer** - First boot can take 2-3 minutes
2. **Check Ethernet cable** - Look for link lights on port
3. **Use different discovery method:**

```
# Try hostname:
ping raspberrypi5.local

# Scan network:
sudo nmap -sn 192.168.1.0/24

# Check ARP table:
arp -a | grep -i raspberry
```

4. **Connect monitor** to see if Pi booted correctly
5. **Check router's DHCP** client list

Issue: Kernel Build Fails

Common causes and solutions:

Missing dependencies:

```
sudo apt install -y libelf-dev libssl-dev
```

Out of disk space:

```
df -h ~
# Ensure at least 20GB free
```

Corrupted download:

```
cd ~/rpi-driver-dev
rm -rf linux
git clone --depth=1 -b rpi-6.6.y https://github.com/raspberrypi/linux
```

Key Differences from Pi 3/4

When following tutorials or books written for Pi 3/4:

Aspect	Pi 3/4	Pi 5 (This Guide)
SoC	BCM2837/BCM2711	**BCM2712**
CPU	Cortex-A53/A72	**Cortex-A76**
Architecture	32-bit or 64-bit	**64-bit only**
Cross-compiler	arm-linux-gnueabihf-	**aarch64-linux-gnu-**
Kernel config	bcm2709_defconfig (32-bit)	**bcm2711_defconfig (64-bit)**
Kernel image	kernel7.img or kernel8.img	**kernel_2712.img**
Device tree	bcm2710-rpi-3-b.dtb	**bcm2712-rpi-5-b.dtb**
Typical kernel	5.4.y, 5.10.y	**6.6.y (LTS)**
Boot partition	/boot	**/boot/firmware or /media/user/bootfs**

GPIO Differences

Pi 5 has the same 40-pin GPIO header layout, but: - Internal GPIO implementation changed - Some advanced features differ - Device tree overlays must specify BCM2712

When adapting code: - GPIO pin numbers (BCM numbering) remain the same - Physical pin layout is identical - Driver code may need minor adjustments for new GPIO chip

---

Device Tree Considerations

For Pi 3/4 tutorials: - Replace `bcm2710` or `bcm2711` with `bcm2712` in device tree paths - Pi 5 device tree: `arch/arm64/boot/dts/broadcom/bcm2712-rpi-5-b.dts` - Overlays location: `arch/arm64/boot/dts/overlays/`

Example overlay modification:

```
// Pi 4 overlay:
/dts-v1/;
/plugin/;
/ {
    compatible = "brcm,bcm2711";
    ...
```

```
};

// Pi 5 overlay:
/dts-v1/;
/plugin/;
/ {
    compatible = "brcm,bcm2712";  // Changed
    ...
};
```

Kernel API Differences (5.4 → 6.6)

Most APIs are compatible, but watch for:

GPIO subsystem: - Prefer `gpiod` (descriptor-based) API over legacy GPIO - Some deprecated functions removed

Timer API: - `timer_setup()` is standard (was new in 5.4) - Old `init_timer()` removed

I2C/SPI: - Mostly backward compatible - Some new helper functions available

Check kernel documentation when porting old code:

```
# In kernel source:
ls Documentation/driver-api/
```

Summary Checklist

Development PC Setup

- Ubuntu 22.04/24.04 LTS installed
- Cross-compiler toolchain installed
- Kernel source cloned (6.6.y branch)
- Kernel configured for Pi 5
- Kernel and modules built successfully
- Build verified (Image, dtbs, modules exist)

SD Card Preparation

- Raspberry Pi OS (64-bit) flashed
- SSH enabled
- User account created
- Custom kernel copied to boot partition
- Kernel modules installed to root partition
- Device tree files copied

Raspberry Pi 5

- First boot successful
- Network connectivity (Ethernet)
- SSH access working
- Custom kernel running (6.6.78-v8-16k+)
- Test module loads/unloads successfully

Development Workflow

- Can build modules on PC
- Can deploy modules to Pi via SCP
- Can load/unload modules on Pi
- Can view kernel logs (dmesg)

Next Steps

You're now ready to:

1. **Start learning Linux driver development**
   - Follow kernel module tutorials
   - Study device driver books (adapt for 6.6 and Pi 5)
   - Explore kernel documentation
2. **Experiment with hardware**
   - GPIO drivers
   - I2C/SPI device drivers
   - Interrupt handling
   - DMA operations
3. **Build real projects**
   - Custom device drivers
   - Hardware interfacing
   - Embedded Linux applications

Additional Resources

Official Documentation: - Raspberry Pi 5 Documentation: https://www.raspberrypi.com/documentation/ - BCM2712 Peripherals: https://datasheets.raspberrypi.com/bcm2712/ - Linux Kernel Documentation: https://www.kernel.org/doc/html/v6.6/

Kernel Development: - Linux Device Drivers (LDD3) - Classic book (adapt for modern kernel) - Bootlin's Embedded Linux training materials - Kernel source code: Best documentation available!

Community: - Raspberry Pi Forums: https://forums.raspberrypi.com/ - Linux Kernel Mailing List (LKML) - Stack Overflow (embedded-linux tag)

Credits

This guide was created based on a successful setup session with Earnest on February 10, 2026, adapting Linux kernel 5.4 development materials for Raspberry Pi 5 with kernel 6.6.78.

Special thanks to: - Raspberry Pi Foundation for excellent hardware and software - Linux kernel developers for their outstanding work - The open-source community for documentation and support

---

Version History

- **v1.0** (Feb 10, 2026) - Initial complete guide based on successful Pi 5 setup
  - Target: Raspberry Pi 5
  - Kernel: Linux 6.6.78-v8-16k+
  - Architecture: aarch64 (64-bit ARM)

License

Happy driver development!