List of Real time Projects:

1. Boot time Optimization
2. Add Splash screen in Human Machine Interface
3. Self-Diagnostic Tool
4. Data logger
5. Smart Weather Monitoring System
6. Vibration Sentry
7. Network Router
8. Data Acquisition System

Project Execution:

Step 1: Setup Embedded Linux Development Environment

Step 2: Prepare Project Abstract

Step 3: Implementation phase @ Bootloader

Step 4: Implementation phase @ Kernel – Device drivers

Step 5: Implementation phase @ User

Real-Time Projects Guide

1. Boot Time Optimization

Goal: Reduce system boot time (e.g., Beagle Bone, Raspberry Pi, custom board).

- **Applications:** Automotive dashboards, Drones, Medical devices where startup time must be < 3s.
- **Execution:**
    - Bootloader: Optimize U-Boot (disable delays, unnecessary drivers).
    - Kernel: Use initramfs, disable unused kernel configs.
    - User space: Parallel service startup, systemd optimization.

---

2. Add Splash Screen in Human Machine Interface (HMI)

Goal: Show a company logo/animation during boot.

- **Applications:** Automotive infotainment, Industrial HMI panels, ATMs, Smart appliances.
- **Execution:**
    - Bootloader: Add BMP logo support in U-Boot.
    - Kernel: Use fbsplash / plymouth.
    - User: Integrate with GUI framework (Qt/GTK).

---

3. Self-Diagnostic Tool

Goal: Tool that checks CPU load, memory, I/O devices, network, sensors.

- **Applications:** Defense electronics, aerospace, medical electronics (must run pre-checks before usage).
- **Execution:**
    - Bootloader: Basic hardware init check.
    - Kernel: Write diagnostic drivers (e.g., EEPROM read, sensor probe).
    - User: CLI/GUI to display results, generate logs.

---

4. Data Logger

Goal: Collect sensor/telemetry data & store with timestamps.

- **Applications:** Automotive black box, Industrial monitoring, Weather stations.
- **Execution:**
    - Bootloader: RTC setup for timestamps.
    - Kernel: I2C/SPI/UART drivers for sensors.
    - User: Log to SQLite/CSV, support USB export.

---

5. Smart Weather Monitoring System

Goal: Collect Temp, Humidity, Pressure, Wind data + Cloud logging.

- **Applications:** Smart cities, Agriculture, Defense border posts.
- **Execution:**
    - Bootloader: Board bring-up with minimal boot.

- Kernel: Drivers for I2C sensors (BME280, DHT11).
- User: GUI + MQTT/HTTP publish to cloud.

---

6. Vibration Sentry

Goal: Detect abnormal vibrations (FFT, threshold) to prevent damage.

- **Applications:** Aircraft health monitoring, Industrial machines, Bridges/turbines.
- **Execution:**
  - Bootloader: Enable high-speed ADC init.
  - Kernel: SPI/ADC driver for accelerometer.
  - User: Signal processing (FFT in C/Python), alerts (buzzer/email).

---

7. Network Router

Goal: Build a custom embedded router (NAT, Firewall, DHCP).

- **Applications:** Defense communication, IoT gateways, Industrial routers.
- **Execution:**
  - Bootloader: Optimize boot for networking.
  - Kernel: Enable netfilter, iptables, routing modules.
  - User: Provide Web UI / CLI to configure router.

---

8. Data Acquisition System (DAS)

Goal: Collect, process & visualize multi-sensor data in real-time.

- **Applications:** Flight test systems, Automotive ECU test benches, Industrial automation.
- **Execution:**
    - Bootloader: RTC, fast init.
    - Kernel: Drivers for UART/SPI/I2C/Ethernet.
    - User: Store in DB + live graph visualization (Qt/Python matplotlib).

Project Execution Flow (same for all projects)

Step 1: Setup Embedded Linux Development Environment

- Cross compiler: arm-linux-gnueabihf-gcc
- U-Boot source & Kernel source
- RootFS (BusyBox, Buildroot, Yocto)
- Hardware: BBB / RPi

Step 2: Prepare Project Abstract

- Problem Statement
- Hardware used
- Software stack (Bootloader, Kernel, User space apps)

- Expected output + demo plan

Step 3: Implementation Phase @ Bootloader

- Configure U-Boot: board init, splash/logo, RTC init, fast boot.
- Optimize environment variables.

Step 4: Implementation Phase @ Kernel – Device drivers

- Write/modify drivers for UART, SPI, I2C, ADC, Display.
- Optimize kernel configs (remove unused features).

Step 5: Implementation Phase @ User

- Build application in C/Qt/Python.
- Add GUI, storage (DB/CSV), and cloud (MQTT/HTTP).
- Logs, alerts, visualization.

💯 Perfect ra baa 🚀
You said it correctly – all these projects follow **the same execution backbone** → only hardware/drivers & user-apps will differ.
So let's prepare a **ground-level execution flow** once → then you can **apply it to any project (Data Logger, DAS, Router, etc.)**.

Ground-Level Project Execution Flow (for ALL Embedded Linux Projects)

Step 1: Setup Embedded Linux Development Environment

This is your **foundation**.

1. **Host Setup (Ubuntu PC / VM)**
2. sudo apt-get update
3. sudo apt-get install gcc-arm-linux-gnueabihf build-essential git make libncurses5-dev bison flex u-boot-tools
4. **Cross Compiler**
5. arm-linux-gnueabihf-gcc --version

(Confirm toolchain is installed)

6. **Source Code Tree**
7. ~/KM_GITHUB/
8. ├── u-boot/ # U-Boot source
9. ├── kernel/ # Linux source
10. ├── rootfs/ # BusyBox / Yocto / Buildroot
11. └── apps/ # Your custom C/Qt/Python apps
12. **Hardware Target**
    - Start with **BeagleBone Black** or **Raspberry Pi**.
    - Use SD card boot.

Step 2: Prepare Project Abstract

- **Objective**: e.g., “Data Logger collects sensor values over I2C and stores in CSV.”
- **Hardware**: e.g., BBB + BME280 sensor + 16x2 LCD + SD card.
- **Software Stack**: U-Boot → Kernel → Drivers → User app.
- **Output**: Data file + Graph + Logs.

Step 3: Implementation @ Bootloader

Work in U-Boot stage

1. **Get U-Boot Source**
2. git clone https://source.denx.de/u-boot/u-boot.git
3. cd u-boot
4. make CROSS_COMPILE=arm-linux-gnueabihf- am335x_boneblack_defconfig
5. make CROSS_COMPILE=arm-linux-gnueabihf- -j4
6. **Customization Points**
    - **Boot Time Optimization** → remove delays (bootdelay=0 in include/configs/board.h).
    - **Splash Screen** → enable BMP support & embed logo.
    - **Diagnostics Init** → add minimal pre-boot checks.
7. **Deploy to SD card**

8. sudo dd if=MLO of=/dev/mmcblk0 seek=1
9. sudo dd if=u-boot.img of=/dev/mmcblk0 seek=1

Step 4: Implementation @ Kernel – Device Drivers

Main work for drivers

1. **Get Kernel Source**
2. git clone https://github.com/beagleboard/linux.git -b 5.10
3. cd linux
4. make CROSS_COMPILE=arm-linux-gnueabihf- bb.org_defconfig
5. make CROSS_COMPILE=arm-linux-gnueabihf- zImage dtbs -j4
6. **Write / Modify Drivers**
    - **Char driver** (for sensors, ADC, EEPROM).
    - **Platform driver** (for I2C/SPI).
    - **Net driver configs** (for router project).

Example skeleton driver:

```
static int my_probe(struct platform_device *pdev) {
   pr_info("Device probed\n");
   return 0;
}
```

```
static int my_remove(struct platform_device *pdev) {
    pr_info("Device removed\n");
    return 0;
}
static struct platform_driver my_driver = {
    .probe = my_probe,
    .remove = my_remove,
    .driver = {
        .name = "my_device",
    },
};
module_platform_driver(my_driver);
```

7. **Deploy Kernel & Modules**
8. make CROSS_COMPILE=arm-linux-gnueabihf- modules_install INSTALL_MOD_PATH=~/rootfs

---

Step 5: Implementation @ User Space

Here comes your application logic.

1. **RootFS Build**
    - **BusyBox** for minimal apps.

- **Yocto / Buildroot** if you want full Linux distro.

2. git clone git://busybox.net/busybox.git
3. cd busybox
4. make defconfig
5. make CROSS_COMPILE=arm-linux-gnueabihf- install
6. **Custom User Apps**
    - **C app** → read /dev/my_driver → log to file.
    - **GUI (Qt/GTK)** → for HMI projects.
    - **Python/MQTT** → for cloud apps (weather system).

Example Data Logger App (C):

```
#include <stdio.h>
int main() {
   FILE *fp = fopen("/dev/sensor", "r");
   int val;
   while(1) {
      fscanf(fp, "%d", &val);
      printf("Sensor Value: %d\n", val);
      FILE *log = fopen("/data/log.csv", "a");
      fprintf(log, "%d\n", val);
```

```
        fclose(log);
        sleep(1);
    }
    return 0;
}
```

7. **Testing**
    - Boot board, check /dev for your drivers.
    - Run your app & verify logs/UI.

How This Flow Applies to Each Project

- **Boot Time Optimization** → focus on Step 3 (U-Boot + Kernel).
- **Splash Screen** → Step 3 (U-Boot) + Step 4 (Framebuffer driver).
- **Self-Diagnostic** → Step 4 (drivers) + Step 5 (test apps).
- **Data Logger** → Step 4 (sensor driver) + Step 5 (logging app).
- **Weather System** → Step 4 (I2C drivers) + Step 5 (cloud publish).
- **Vibration Sentry** → Step 4 (ADC driver) + Step 5 (FFT app).
- **Router** → Step 4 (netfilter configs) + Step 5 (web/CLI).
- **DAS** → Step 4 (multi-driver integration) + Step 5 (visualization).

End-to-end flow: Bootloader → Kernel → Driver → User-space app → Demo.

Project: Data Logger on Beagle Bone Black

Objective

- Collect sensor values (e.g., Temp/ADC input)
- Log them with timestamps into a file (/data/log.csv)
- Optional: send to cloud via MQTT/HTTP

Hardware Setup

- **Board**: Beagle Bone Black (BBB)
- **Sensor**: LM35 (Analog Temp sensor → ADC input)
- **Storage**: eMMC/SD card

Step 1: Bootloader (U-Boot)

Minimal work, but prepare system for fast boot.

1. **Get U-Boot**
2. git clone https://source.denx.de/u-boot/u-boot.git
3. cd u-boot
4. make CROSS_COMPILE=arm-linux-gnueabihf- am335x_boneblack_defconfig

5. make CROSS_COMPILE=arm-linux-gnueabihf- -j4
6. **Optimize Boot**
    - Edit include/configs/am335x_evm.h
    - #define CONFIG_BOOTDELAY 0
    - Set console env var to only one UART.
7. **Flash to SD card**
8. sudo dd if=MLO of=/dev/mmcblk0 seek=1
9. sudo dd if=u-boot.img of=/dev/mmcblk0 seek=1

Now boot is faster and ready.

Step 2: Kernel (Driver Level)

We'll write a **simple character driver** to expose sensor (ADC) readings at /dev/logger.

1. **Skeleton Driver (logger.c)**
2. #include <linux/module.h>
3. #include <linux/fs.h>
4. #include <linux/uaccess.h>
5. #include <linux/init.h>
6.
7. #define DEVICE_NAME "logger"

```
8.  static int major;
9.
10. // Dummy read function (replace with ADC read later)
11. static ssize_t logger_read(struct file *f, char __user *buf, size_t len, loff_t *off) {
12.     int sensor_val = 1234; // mock data, replace with actual ADC
13.     char temp[20];
14.     int n = sprintf(temp, "%d\n", sensor_val);
15.     if (copy_to_user(buf, temp, n))
16.         return -EFAULT;
17.     return n;
18. }
19.
20. static struct file_operations fops = {
21.     .owner = THIS_MODULE,
22.     .read = logger_read,
23. };
24.
25. static int __init logger_init(void) {
26.     major = register_chrdev(0, DEVICE_NAME, &fops);
```

```
    printk(KERN_INFO "Logger driver loaded, major=%d\n", major);
    return 0;
}
static void __exit logger_exit(void) {
    unregister_chrdev(major, DEVICE_NAME);
    printk(KERN_INFO "Logger driver unloaded\n");
}

module_init(logger_init);
module_exit(logger_exit);
MODULE_LICENSE("GPL");
```

Makefile

```
obj-m += logger.o
all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Build & Insert

```
make CROSS_COMPILE=arm-linux-gnueabihf-
```

```
scp logger.ko root@192.168.7.2:/root/
ssh root@192.168.7.2
insmod logger.ko
dmesg | tail
ls -l /dev/logger
```

Now /dev/logger gives dummy sensor data.

Step 3: User-Space Application

1. **Logger App (logger_app.c)**

```
#include <stdio.h>
#include <unistd.h>
#include <time.h>

int main() {
    FILE *dev, *log;
    int val;
    char line[100];
    while (1) {
        dev = fopen("/dev/logger", "r");
```

```
    if (!dev) {
        perror("open");
        return -1;
    }
    fgets(line, sizeof(line), dev);
    fclose(dev);

    val = atoi(line);

    time_t t = time(NULL);
    struct tm tm = *localtime(&t);

    log = fopen("/data/log.csv", "a");
    fprintf(log, "%04d-%02d-%02d %02d:%02d:%02d, %d\n",
            tm.tm_year+1900, tm.tm_mon+1, tm.tm_mday,
            tm.tm_hour, tm.tm_min, tm.tm_sec, val);
    fclose(log);

    printf("Logged: %d\n", val);
```

```
        sleep(1);
    }
    return 0;
}
```

35. **Compile**

```
arm-linux-gnueabihf-gcc logger_app.c -o logger_app
scp logger_app root@192.168.7.2:/root/
```

38. **Run on Target**

```
./logger_app
tail -f /data/log.csv
```

You will see sensor values getting logged every second with timestamp.

Step 4: Extension Ideas

- Replace dummy driver code with **ADC read** (ti_am335x_adc.c in kernel).
- Add **USB/SD export** → copy logs to USB.
- Add **MQTT publish** → send data to cloud (IoT).
- Add **Qt GUI** → real-time graph on touchscreen.

Applications

- Automotive Black Box (ECU logs)
- Industrial Machine Monitoring
- Defense: Flight test recorders
- Weather Stations
- Power plant data recorders

Project: Smart Weather Monitoring System

Objective

- Collect **temperature, humidity, pressure** from sensor
- Log locally (/data/weather.csv)
- Display on console/GUI (Qt/GTK)
- Push data to **Cloud (MQTT broker / HTTP server)**

Hardware Setup

- **Board**: BeagleBone Black (BBB)
- **Sensor**: BME280 (I^2C Temp + Pressure + Humidity sensor)
- **Optional**: 16x2 LCD or HDMI display for GUI
- **Connectivity**: Ethernet/Wi-Fi (for cloud push)

Step 1: Bootloader (U-Boot)

👉 Same as Data Logger – just ensure **fast boot + correct device tree loading**.

bootdelay=0

setenv bootargs console=ttyO0,115200n8 root=/dev/mmcblk0p2 rw

saveenv

✅ No major changes except ensuring **I²C enabled in DT**.

Step 2: Kernel (Driver Level)

We'll expose **BME280 sensor** via a platform driver → create /dev/weather.

1. **Enable I²C in Kernel**
2. make menuconfig
3. Device Drivers → I2C Support → I2C Device Interface (enable)
4. **Minimal BME280 Driver (weather.c)**
5. #include <linux/module.h>
6. #include <linux/i2c.h>
7. #include <linux/fs.h>
8. #include <linux/uaccess.h>
9.

```
#define DEVICE_NAME "weather"
static int major;
static struct i2c_client *bme280_client;

static ssize_t weather_read(struct file *f, char __user *buf, size_t len, loff_t *off) {
    char data[8];
    int temp, hum, press;

    // Fake values for demo (replace with real I2C read later)
    temp = 27; hum = 60; press = 1013;
    int n = sprintf(data, "%d,%d,%d\n", temp, hum, press);

    if (copy_to_user(buf, data, n)) return -EFAULT;
    return n;
}

static struct file_operations fops = {
    .owner = THIS_MODULE,
    .read = weather_read,
```

```
29. };
30.
31. static int weather_probe(struct i2c_client *client, const struct i2c_device_id *id) {
32.     bme280_client = client;
33.     major = register_chrdev(0, DEVICE_NAME, &fops);
34.     pr_info("Weather device ready at /dev/%s major=%d\n", DEVICE_NAME, major);
35.     return 0;
36. }
37.
38. static int weather_remove(struct i2c_client *client) {
39.     unregister_chrdev(major, DEVICE_NAME);
40.     return 0;
41. }
42.
43. static const struct i2c_device_id weather_id[] = { { "bme280", 0 }, {} };
44. MODULE_DEVICE_TABLE(i2c, weather_id);
45.
46. static struct i2c_driver weather_driver = {
47.     .driver = { .name = "weather" },
```

```
    .probe = weather_probe,
    .remove = weather_remove,
    .id_table = weather_id,
};

module_i2c_driver(weather_driver);
MODULE_LICENSE("GPL");
```

Build & Deploy

```
make CROSS_COMPILE=arm-linux-gnueabihf-
scp weather.ko root@192.168.7.2:/root/
ssh root@192.168.7.2
insmod weather.ko
cat /dev/weather
```

✅ /dev/weather → gives “27,60,1013” (temp=27°C, humidity=60%, pressure=1013 hPa).

Step 3: User-Space Application

1. **Weather Logger App (weather_app.c)**

```
#include <stdio.h>
#include <unistd.h>
```

```
4.  #include <time.h>
5.
6.  int main() {
7.      FILE *dev, *log;
8.      char line[100];
9.      int temp, hum, press;
10.
11.     while (1) {
12.         dev = fopen("/dev/weather", "r");
13.         if (!dev) { perror("open"); return -1; }
14.         fgets(line, sizeof(line), dev);
15.         fclose(dev);
16.
17.         sscanf(line, "%d,%d,%d", &temp, &hum, &press);
18.
19.         time_t t = time(NULL);
20.         struct tm tm = *localtime(&t);
21.
22.         log = fopen("/data/weather.csv", "a");
```

```
        fprintf(log, "%04d-%02d-%02d %02d:%02d:%02d, %d, %d, %d\n",
                tm.tm_year+1900, tm.tm_mon+1, tm.tm_mday,
                tm.tm_hour, tm.tm_min, tm.tm_sec,
                temp, hum, press);
        fclose(log);

        printf("Logged: Temp=%dC, Hum=%d%%, Press=%dhPa\n", temp, hum, press);
        sleep(5);
    }
    return 0;
}
```

Compile

```
arm-linux-gnueabihf-gcc weather_app.c -o weather_app
scp weather_app root@192.168.7.2:/root/
```

Run on Target

```
./weather_app
tail -f /data/weather.csv
```

✅ Live weather data gets logged every 5s.

Step 4: Cloud Push (Optional but Killer 🔥)

1. **Install Mosquitto (MQTT Broker on PC/Cloud)**
2. sudo apt install mosquitto mosquitto-clients
3. mosquitto -v
4. **Modify User App**
5. char mqtt_cmd[200];
6. sprintf(mqtt_cmd, "mosquitto_pub -h 192.168.1.100 -t weather -m \"Temp=%d,Hum=%d,Press=%d\"",
7. temp, hum, press);
8. system(mqtt_cmd);
9. **Receive on Subscriber**
10. mosquitto_sub -h 192.168.1.100 -t weather

✅ Now your BBB sends **live sensor data to cloud**.

Step 5: Extensions

- Qt/GTK GUI with graphs 📊
- Store to **SQLite** for advanced querying
- Integrate with **Grafana/InfluxDB dashboard**
- SMS/email alerts when thresholds exceeded

Applications

- Agriculture: Greenhouse monitoring 🌱
- Smart Cities: Distributed environment monitoring 🌆
- Defense: Border weather monitoring 🪖
- Aviation: Runway weather check 🛬
- Industrial HVAC monitoring 🏭

Full ground-level roadmap for **Vibration Sentry**: hardware, boot/kernel/user-space pieces, exact commands, sample driver skeleton (IIO-based ADC consumer), a self-contained **C FFT** for user-space processing, alert methods, testing & tuning tips. Apply the same U-Boot / Kernel / User flow we used earlier.

Vibration Sentry — Goal

Detect abnormal vibration patterns in real time (FFT / threshold), raise alerts (buzzer, log, MQTT/email), and record events. Useable for motors, turbines, vehicles, bridges.

---

1) Hardware & system choices

- Board: **BeagleBone Black** (BBB) or any ARM board with IIO ADC or external ADC (SPI/I²C).
- Sensor: **3-axis accelerometer** (e.g., ADXL345 over SPI/I²C) or analog accelerometer (e.g., ADXL335) → requires ADC.
- Optional: external high-speed ADC (ADS1115/ADS1298) for higher precision.
- Output: Buzzer (GPIO), LED, Ethernet/Wi-Fi (for MQTT), SD card for logs.

Recommend starting with: BBB built-in ADC (for proof-of-concept) or SPI-accelerometer (ADXL345) for better sampling.

---

2) High-level design

- **Bootloader**: Ensure fast boot, device tree has I²C/SPI and ADC enabled.
- **Kernel**: Enable IIO (Industrial I/O) and driver for accelerometer / ADC. Write small kernel module that exposes a character device /dev/vibsentry or uses IIO buffer for high-rate sampling.
- **User-space**:
    - Reader process: read samples, run windowed FFT, compute spectral metrics (RMS, dominant freq, band energy).
    - Detector: apply thresholds and pattern logic (e.g., sudden spike, persistent vibration band).
    - Logger & Alert: write to CSV, send MQTT/email, toggle GPIO buzzer.
    - Optional GUI: Qt/GTK plot of time-domain & frequency-domain.

---

3) Bootloader (U-Boot) notes

- Ensure device tree has needed nodes (i2c/spi, adc, gpio) and fast boot.
- Minimal changes: bootdelay=0, correct bootargs for rootfs networking.

---

4) Kernel — enable required subsystems

Kernel config (menuconfig):

- Device Drivers → Industrial I/O support (IIO)

    - IIO support (y)
    - IIO buffer support (y)
    - IIO trigger support (y)
    - ADC/Accelerometer drivers (enable relevant drivers or generic spi/i2c client)
- Enable I2C/SPI support as needed.
- Consider PREEMPT_RT patch or CONFIG_PREEMPT if you need better determinism (for high-rate sampling).

Commands to build kernel (example):

cd linux

make CROSS_COMPILE=arm-linux-gnueabihf- bb.org_defconfig

make CROSS_COMPILE=arm-linux-gnueabihf- zImage dtbs -j4

make CROSS_COMPILE=arm-linux-gnueabihf- modules -j4

make CROSS_COMPILE=arm-linux-gnueabihf- modules_install INSTALL_MOD_PATH=~/rootfs

---

5) Kernel module skeleton — IIO consumer + char device

This skeleton gets sensor channel via IIO consumer API, starts a kernel thread sampling at given rate, and exposes latest samples via /dev/vibsentry. (Adapt to your board's channel names.)

vibsentry.c (simplified):

// Compile as kernel module. This is a simplified skeleton — adapt to your DT and channel names.

#include <linux/module.h>

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/kthread.h>
#include <linux/delay.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/iio/consumer.h>

#define DEVICE_NAME "vibsentry"
static int major;
static struct task_struct *thread;
static struct iio_channel *chan_x, *chan_y, *chan_z;
static atomic_t running = ATOMIC_INIT(0);
static int sample_rate_hz = 1000; // default sampling rate

static int vibs_thread(void *data) {
    int ms = 1000 / sample_rate_hz;
    while (!kthread_should_stop()) {
        int valx = 0, valy = 0, valz = 0;
```

```
        // Read processed values (micro-units depending on channel)
        iio_read_channel_processed(chan_x, &valx);
        iio_read_channel_processed(chan_y, &valy);
        iio_read_channel_processed(chan_z, &valz);
        // store to ring buffer or last-sample variables (omitted for brevity)
        // For demo, we just printk occasionally
        printk(KERN_DEBUG "VIB: x=%d y=%d z=%d\n", valx, valy, valz);
        msleep(ms);
    }
    return 0;
}

static ssize_t vibs_read(struct file *f, char __user *buf, size_t len, loff_t *off) {
    // Return last sample(s) in CSV form. (Implement proper buffering when needed)
    char tmp[64];
    int n = snprintf(tmp, sizeof(tmp), "0,0,0\n"); // replace with actual data
    if (copy_to_user(buf, tmp, n)) return -EFAULT;
    return n;
}
```

```
static struct file_operations fops = {
    .owner = THIS_MODULE,
    .read = vibs_read,
};

static int __init vibs_init(void) {
    major = register_chrdev(0, DEVICE_NAME, &fops);
    // get channels by name (replace "accel_x" etc. with your channel names)
    chan_x = iio_channel_get(NULL, "accel_x");
    chan_y = iio_channel_get(NULL, "accel_y");
    chan_z = iio_channel_get(NULL, "accel_z");
    if (IS_ERR(chan_x) || IS_ERR(chan_y) || IS_ERR(chan_z)) {
        pr_err("Failed to get iio channels\n");
        return -ENODEV;
    }
    atomic_set(&running, 1);
    thread = kthread_run(vibs_thread, NULL, "vibs_thread");
    pr_info("vibsentry loaded, major=%d\n", major);
```

```
    return 0;
}

static void __exit vibs_exit(void) {
    if (thread) kthread_stop(thread);
    iio_channel_release(chan_x);
    iio_channel_release(chan_y);
    iio_channel_release(chan_z);
    unregister_chrdev(major, DEVICE_NAME);
    pr_info("vibsentry unloaded\n");
}

module_init(vibs_init);
module_exit(vibs_exit);
MODULE_LICENSE("GPL");
```

Notes:

- Replace channel names per DT / board (use iio_info on target to inspect).
- For production, use an efficient ring buffer (kernels or user-space mmap) and proper locking.

Build & deploy:

make CROSS_COMPILE=arm-linux-gnueabihf- -C /lib/modules/$(uname -r)/build M=$PWD modules

scp vibsentry.ko root@192.168.7.2:/root/

ssh root@192.168.7.2 'insmod /root/vibsentry.ko'

ls -l /dev/vibsentry

dmesg | tail

---

6) User-space: sampling, FFT & detection

We'll implement a simple windowed FFT (Cooley–Tukey) in C — small and self-contained. Use fixed window size (power of two). Example: 1024 samples window at 1 kHz => 1.024 s resolution.

fft_simple.c — contains a basic complex FFT and a demo of reading /dev/vibsentry and performing FFT on one axis.

```
/* Simplified single-file FFT + reader. Not optimized for speed. */

#include <math.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>


#define N 1024

typedef struct { double re, im; } complex_t;
```

```
void fft_recursive(complex_t *x, int n) {
    if (n <= 1) return;
    int k;
    complex_t *even = malloc(n/2 * sizeof(complex_t));
    complex_t *odd  = malloc(n/2 * sizeof(complex_t));
    for (k=0;k<n/2;k++) { even[k] = x[2*k]; odd[k] = x[2*k+1]; }
    fft_recursive(even, n/2); fft_recursive(odd, n/2);
    for (k=0;k<n/2;k++) {
        double t = -2*M_PI*k / n;
        complex_t wk = { cos(t), sin(t) };
        complex_t tmp = { wk.re*odd[k].re - wk.im*odd[k].im,
                          wk.re*odd[k].im + wk.im*odd[k].re };
        x[k].re = even[k].re + tmp.re; x[k].im = even[k].im + tmp.im;
        x[k+n/2].re = even[k].re - tmp.re; x[k+n/2].im = even[k].im - tmp.im;
    }
    free(even); free(odd);
}
```

```
int main() {
    FILE *dev = fopen("/dev/vibsentry", "r");
    if (!dev) { perror("open"); return -1; }
    double samples[N];
    while (1) {
        for (int i=0;i<N;i++) {
            char line[128];
            if (!fgets(line, sizeof(line), dev)) {
                // if read fails, reopen
                fclose(dev); dev = fopen("/dev/vibsentry","r"); i--; continue;
            }
            int x,y,z;
            if (sscanf(line,"%d,%d,%d",&x,&y,&z)!=3) { i--; continue; }
            samples[i] = (double)x; // use x-axis; scale/convert as needed
        }
        // prepare complex array
        complex_t *arr = malloc(N * sizeof(complex_t));
        for (int i=0;i<N;i++) { arr[i].re = samples[i]; arr[i].im = 0; }
        fft_recursive(arr, N);
```

```
    // compute magnitude & find peak bin
    int peak_idx = 0; double peak_val = 0;
    for (int k=0;k<N/2;k++) {
        double mag = sqrt(arr[k].re*arr[k].re + arr[k].im*arr[k].im);
        if (mag > peak_val) { peak_val = mag; peak_idx = k; }
    }
    double sampling_rate = 1000.0; // Hz (match kernel sampling rate)
    double freq = (double)peak_idx * sampling_rate / N;
    printf("Peak freq: %.2f Hz, magnitude %.2f\n", freq, peak_val);
    // Detection logic example:
    if (peak_val > 1e6 || freq > 200) {
        // trigger alert
        system("echo ALERT > /dev/vib_alert"); // implement /dev/vib_alert to toggle buzzer or use gpio sysfs
        // publish MQTT or log
        system("mosquitto_pub -h 192.168.1.100 -t vib/alerts -m \"vibration alert\"");
    }
    free(arr);
}
fclose(dev);
```

```
    return 0;
}
```

Compile for target:

arm-linux-gnueabihf-gcc fft_simple.c -lm -o fft_simple

scp fft_simple root@192.168.7.2:/root/

Notes:

- This FFT is recursive and simple — for production use use optimized libraries (KissFFT, FFTW, or fixed-point DSP libs). But manual FFT is great to learn and for small demos.
- Scale and convert ADC units to g (acceleration) if needed.

---

7) Alerting mechanisms

- **GPIO buzzer / LED**: Use sysfs or gpiod to toggle buzzer when detection occurs. Kernel can expose /dev/vib_alert or user-space can write to /sys/class/gpio/gpioX/value.
- **MQTT**: mosquitto_pub to publish alerts.
- **Email/SMS**: call REST API or sendmail script.
- **Hardware interlock**: via GPIO to a relay to shut down motor.

Example user-space buzzer toggle:

echo 60 > /sys/class/gpio/export

echo out > /sys/class/gpio/gpio60/direction

```
echo 1 > /sys/class/gpio/gpio60/value  # turn buzzer on
sleep 1
echo 0 > /sys/class/gpio/gpio60/value  # off
```

---

8) Data logging & visualization

- Log raw samples or downsampled metrics to CSV (/data/vib_log.csv).
- Use InfluxDB + Grafana on a host to visualize trends; push via MQTT→Telegraf→InfluxDB.
- For local GUI, write a Qt app that plots time-domain and frequency-domain every window.

---

9) Testing plan & demo steps

1. Boot board; ensure kernel modules loaded and /dev/vibsentry present.
2. Run cat /dev/vibsentry to verify streaming samples.
3. Start fft_simple on target; check printed peak freq lines.
4. Simulate vibration (tap sensor) — observe freq changes.
5. Validate alert: verify buzzer toggles and MQTT message received.
6. Long-run: run for hours, inspect /data/vib_log.csv for event records.

Demo script example:

```
insmod /root/vibsentry.ko
/root/fft_simple > /root/vib_out.log &
```

mosquitto_sub -h 192.168.1.100 -t vib/alerts &

simulate: gently tap sensor or connect to vibration shaker

---

10) Real-time & performance tuning

- **Sampling**: use kernel buffer (IIO buffer + trigger) for high sample rates and DMA if supported. Avoid msleep sampling in kernel for high rates.
- **Latency**: use PREEMPT_RT for deterministic behavior if you need low-latency detection.
- **Offload**: perform FFT in optimized C or use hardware DSP if available.
- **Windowing**: use overlap-add, window functions (Hann) for better spectral results.
- **Filtering**: apply anti-aliasing filter on analog path or digital low-pass.

---

11) Extensions & improvements

- Use **3-axis combined metrics** (vector magnitude) sqrt(x^2+y^2+z^2) for detection robustness.
- Use ML models (simple classifier) in user-space to detect fault signatures.
- Store events in SQLite with timestamps and GPS if mobile.
- Add OTA update for user-space logic.

---

12) Deliverables I can prepare next (choose one)

- Full **production-ready IIO buffered kernel driver** (with DT overlay & examples).

- Optimized **C FFT** using KissFFT with overlap/windowing and band-energy calculation.
- **Qt GUI** to show live waveform + spectrogram.
- **Test scripts** and sample data generator to simulate accelerometer.

Project	Bootloader Tasks	Kernel Tasks	User Space Tasks	Real-Time Critical Parts
Boot Time Optimization	Optimize bootcmd, skip unnecessary init	Remove unused modules, enable only required drivers	Disable unnecessary services	Faster boot to start critical applications
Splash Screen in HMI	Load image in framebuffer	Ensure display driver active	Optional GUI for animation	Display without blocking hardware init
Self-Diagnostic Tool	Optional early memory/storage checks	Drivers expose hardware status, handle interrupts	Poll drivers, alert faults, generate reports	Immediate fault detection
Data Logger	Standard boot	Drivers for sensors, DMA setup for efficient transfer	Poll sensors, log data, optional network upload	Precise timing for logging intervals
Smart Weather Monitoring System	Standard boot	Drivers for temp/humidity/wind/rain sensors, interrupt-based events	Collect/analyze data, remote monitoring, GUI display	Timely and accurate data collection
Vibration Sentry	Standard boot	Drivers for accelerometers, threshold interrupts	Analyze vibration patterns, log events, alert	Immediate alert on abnormal vibration

Project	Bootloader Tasks	Kernel Tasks	User Space Tasks	Real-Time Critical Parts
Network Router	Standard boot	Enable network stack, NIC drivers, firewall rules	DHCP/DNS server, routing management, web interface	Low-latency packet forwarding
Data Acquisition System	Standard boot	Drivers for ADC/DAC, SPI/I2C sensors, DMA for high-speed data	Process, visualize, store or transmit data, filtering	Precise and continuous sampling

"In embedded Linux projects:

- **Bootloader** handles **early hardware initialization**, configures memory, loads device tree, and prepares the system for kernel start.
- **Kernel** manages **device drivers**, handles **real-time interrupts**, configures DMA for fast data transfer, and ensures precise timing for hardware operations.
- **User Space** runs **application logic and visualization**, like GUIs, data logging, network services, or diagnostic tools.

Real-Time Critical Tasks occur mainly at **Kernel** (interrupt handling, DMA, sensor polling) and **User Space** (timing-sensitive data logging, analysis, or alerts), ensuring the system responds promptly and reliably."

"Bootloader initializes hardware early, Kernel handles drivers and real-time interrupts, and User Space runs application logic. Real-time critical tasks like DMA, interrupts, and precise data logging occur at Kernel and User Space."

Mnemonic: "BKU – Boot, Kernel, User"

- **B → Bootloader:** "**B**ring up hardware first" ✅
- **K → Kernel:** "**K**eep drivers & interrupts running" ✅
- **U → User Space:** "**U**se apps & visualization" ✅