CE Department
Sharif University of Technology

Internet of Things Lab

Fault Detection on I²C Communication Buses Using Raspberry Pi + Yocto

Project Full Report

Ali Shahheidar: 400105777
Alireza Rezaeimoghadam: 400109938

Spring 03-04

Fault Detection on I²C Communication Buses Using Raspberry Pi + Yocto

Application Pi (Master) + Guardian Pi (Monitor / BusGuard)

Abstract

We build custom Linux images with Yocto for Raspberry Pi and realize a two-board topology:

Application Pi acts as I²C master, communicating with a real sensor/module.

Guardian Pi acts as a passive bus monitor, sampling SDA/SCL, reconstructing I²C frames, and flagging anomalies (unauthorized addresses, NACK/no-response, timing violations and clock stretching, out-of-range values, unexpected register writes, stale data, late responses).

This final report documents every step: build environments (WSL2, Ubuntu 24, final on Ubuntu 22), all Yocto commands and layer configuration, image building and flashing, serial console bring-up, wiring, and lessons learned. It also outlines the BusGuard software architecture and testing strategy.

System Overview

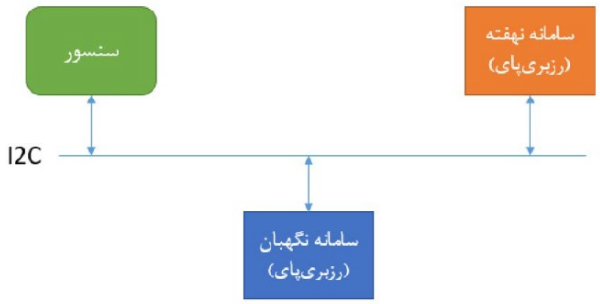
Bus: I²C (shared SDA/SCL).

Application Pi: Raspberry Pi master talking to sensor.

Guardian Pi: Raspberry Pi GPIO inputs only, samples clock/data transitions to reconstruct frames and evaluate rules.

Yocto Image: Built with poky (kirkstone) + meta-raspberrypi + meta-openembedded (+ our custom layer meta-busguard for the monitor).

Deployment: Two SD cards, one per Pi; optional TTL-USB serial console for first-boot & debugging.



Tools & Materials

2× Raspberry Pi (Pi 3/3B+ or Pi 4)

I²C sensor/module (e.g., temperature, IMU, ADC)

MicroSD cards (≥8 GB) + reliable USB SD reader

USB-to-TTL serial, jumper wires; common GND

Build host: Linux (final: Ubuntu 22), Yocto prerequisites, bmaptool

Runtime tools: i2c-tools, python3-smbus, libgpiod, pigpio

BusGuard Architecture

buswatchd – edge capture via pigpio DMA or libgpiod interrupts; frame decode (START/STOP, 7-bit address + R/W, data bytes, ACK/NACK); timestamping; event output (journald/stdout/Unix socket).

buswatch-cli – pretty-prints events; **CSV/JSON** export; filters by rule-file; tail/follow mode.

master-demo.py – minimal I²C traffic generator (read/write a couple of registers periodically).

What we detect (configurable):
Unauthorized address, addressing errors, **NO_ACK**, **bus busy too long**, **tLOW/tHIGH** violations, **clock stretching** beyond limits, **data out-of-range**, **unexpected writes**, **stale data**, **late responses**.

Implementation Details

System Model

Bus: I²C (shared SDA/SCL)

Application Pi (Master): Talks to sensor

Guardian Pi (Monitor): Samples SDA/SCL (GPIO inputs), decodes frames, runs rules, logs & exports events

How Detection Works

Sampling: microsecond-level edges on SDA/SCL

Decoding: transitions → START/ADDR/ACK/byte/STOP tokens

Validation: structure + timing + semantic checks against rules

Reporting: journald/CLI + CSV/JSON export

Yocto Integration (summary)

poky (base distro)

meta-raspberrypi (BSP)

meta-openembedded (meta-oe, meta-python, meta-networking)

meta-busguard (our layer: image + daemon + CLI + systemd)

Minimal conf/local.conf (development):

```
MACHINE ??= "raspberrypi3"
IMAGE_FSTYPES += "rpi-sdimg wic.bz2"

DISTRO_FEATURES:append = " systemd"
VIRTUAL-RUNTIME_init_manager = "systemd"

IMAGE_FEATURES += "ssh-server-dropbear"
EXTRA_IMAGE_FEATURES += "debug-tweaks"

IMAGE_INSTALL:append = " i2c-tools python3 python3-smbus libgpiod pigpio
buswatchd buswatch-cli"

# Enable I2C in firmware for Pi (boot/config.txt)
RPI_EXTRA_CONFIG = "\n# BusGuard\n dtparam=i2c_arm=on\n dtoverlay=i2c1\n
enable_uart=1\n"
```

Rules example `/etc/busguard/rules.yaml`:

```
bus: 1
address_whitelist: [0x48, 0x50]
register_map:
  0x48:
    readable: [0x00, 0x01]
    writable: [0x02]
value_ranges:
  "0x48:0x00": {min: 0, max: 100}
timing_limits:
  max_bus_busy_ms: 50
  max_clock_stretch_us: 3000
stale_window: 20
```

Reproducible Build, final, working path on Ubuntu 22:

What to do now?

prereqs, workspace, source (kirkstone), init build, add layers, local setting, build (examples).

```
sudo apt update && sudo apt upgrade -y
sudo apt install -y gawk wget git-core diffstat unzip texinfo gcc build-essential
\
  chrpath socat cpio python3 python3-pip python3-pexpect xz-utils debianutils \
  iputils-ping python3-git python3-jinja2 libegl1-mesa libsdl1.2-dev \
  xterm curl lzop zstd bmap-tools

mkdir -p ~/yocto && cd ~/yocto

git clone -b kirkstone git://git.yoctoproject.org/poky
cd poky
git clone -b kirkstone https://github.com/agherzan/meta-raspberrypi.git
git clone -b kirkstone https://github.com/openembedded/meta-openembedded.git

source oe-init-build-env build-rpi

bitbake-layers add-layer ../meta-raspberrypi
bitbake-layers add-layer ../meta-openembedded/meta-oe
bitbake-layers add-layer ../meta-openembedded/meta-python
bitbake-layers add-layer ../meta-openembedded/meta-networking
bitbake-layers show-layers

cat >> conf/local.conf <<'EOF'
MACHINE ??= "raspberrypi4"
CONF_VERSION = "2"
ENABLE_UART = "1"
IMAGE_FEATURES += "ssh-server-dropbear"
EXTRA_IMAGE_FEATURES += "debug-tweaks"
LICENSE_FLAGS_ACCEPTED = "synaptics-killswitch"
EOF

bitbake core-image-base
```

Artifacts live under:

```
~/yocto/poky/build-rpi/tmp/deploy/images/raspberrypi3/
```

Look for: `*.wic.bz2` (+ matching `.wic.bmap`).

Flashing SD (with bmaptool)

Unmount partitions first:

```
# Identify the SD card device
lsblk

sudo umount /dev/sdb1
sudo umount /dev/sdb2
```

Copy image with bmaptool

```
sudo bmaptool copy \
  ~/poky/build-rpi/tmp/deploy/images/raspberrypi3/core-image-base-
raspberrypi3.rootfs-20250421093029.wic.bz2 \
  /dev/sdb
```

Pi 3, full cmd line:

```
ls tmp/deploy/images/raspberrypi3/ | grep -i wic
sudo bmaptool copy tmp/deploy/images/raspberrypi3/core-image-full-cmdline-
raspberrypi3.rootfs-20250428154835.wic.bz2 /dev/sdb
```

Useful checks:

```
dmesg | tail -n 20          # device attach/detach messages
sudo dmesg | tail -n 20
lsblk                       # verify /dev/sdb, /dev/sdb1, /dev/sdb2
```

Serial Console & UART

Wiring (CP210x USB-TTL ↔ Raspberry Pi header)

Pi GND → USB-TTL GND

Pi GPIO14 (TXD0), **header pin** 8 → USB-TTL RX

Pi GPIO15 (RXD0), **header pin** 10 → USB-TTL TX

Host setup (Linux)

```
# Identify serial adapter
dmesg | tail -n 50

# Add user to the dialout group (then re-login)
sudo usermod -aG dialout $USER
```

Terminal programs

picocom (your transcript):

```
sudo picocom -b 115200 /dev/ttyUSB0
```

Then PuTTY

Enable UART on the Pi firmware

On **Raspberry Pi** with meta-raspberrypi, config.txt lives on the boot partition. Ensure:

```
enable_uart=1
```

On Yocto, **login user** is **not pi**. For development, add EXTRA_IMAGE_FEATURES += "debug-tweaks" in local.conf and log in as **root** (empty password) on the serial console.

If we want a pi user, create it at build time:

```
# Create a 'pi' user with password 'raspberry' (SHA-512 hash)
inherit extrausers
EXTRA_USERS_PARAMS = "\
  useradd -p '\$6\$CwZ8XvT2\$xX...yourhash.../.' -G
wheel,video,audio,dialout,gpio -s /bin/sh pi; \
"
IMAGE_FEATURES += "ssh-server-dropbear"
```

Generate a hash safely on Linux:

```
openssl passwd -6 -salt CwZ8XvT2 raspberry
```

The **Raspberry Pi Imager "OS Customisation"** (username pi, enabling SSH) **does not apply** to Yocto images; it only customizes Raspberry Pi OS. That's why login attempts with pi/1234 failed on your Yocto build.

Now we summarize all the 7 weeks reports:

Week 1 — WSL2 Attempt

Goal: build Yocto image for Raspberry Pi under **WSL2** on Windows.

Actions: installed WSL2, updated Ubuntu, installed Yocto prerequisites, fetched **poky**, **meta-raspberrypi**, **meta-openembedded**, initialized the build env, edited local.conf and bblayers.conf, ran bitbake core-image-minimal.

Issue: **space** (Yocto needs ~80 GB+) and general friction in WSL2 → repeated failures.

Decision: move to a VM/lab host.

Commands used:

```
wsl --install
sudo apt update && sudo apt upgrade -y
sudo apt install gawk wget git-core diffstat unzip texinfo gcc build-essential \
chrpath socat cpio python3 python3-pip python3-pexpect xz-utils debianutils \
iputils-ping python3-git python3-jinja2 libegl1-mesa libsdl1.2-dev xterm curl
lzop zstd
mkdir ~/yocto && cd ~/yocto
git clone -b kirkstone git://git.yoctoproject.org/poky
cd poky
git clone -b kirkstone https://github.com/agherzan/meta-raspberrypi.git
git clone -b kirkstone https://github.com/openembedded/meta-openembedded.git
source oe-init-build-env
bitbake core-image-minimal
```

Week 2 — Ubuntu 24 VM; AppArmor / user-namespaces error

Moved to **Ubuntu 24** VM (~200 GB).

Hit:

```
ERROR: User namespaces are not usable by BitBake, possibly due to AppArmor
```

After research, chose the pragmatic path: **Ubuntu 22 LTS** (widely used for Yocto builds) to avoid wrestling with policy changes.

Week 3 — Ubuntu 22 LTS VM (Successful build)

Added layers, verified priorities, edited local.conf, built core-image-base, prepared to flash SD.

```
bitbake-layers add-layer ~/poky/meta-raspberrypi/
bitbake-layers add-layer ~/poky/meta-openembedded/meta-openembedded
bitbake-layers show-layers

# local.conf (key lines)
MACHINE ??= "raspberrypi4"
CONF_VERSION = "2"
ENABLE_UART = "1"
LICENSE_FLAGS_ACCEPTED = "synaptics-killswitch"

bitbake core-image-base
cd ~/poky/build-rpi/tmp/deploy/images/raspberrypi4
ls | grep -i wic
```

Week 4 — Flashing SD & Serial Console bring-up

Identified SD as /dev/sdb; unmounted partitions; flashed with bmaptool.

Verified attach/detach in dmesg; used **picocom** at 115200 on /dev/ttyUSB0.

```
lsblk
sudo umount /dev/sdb1
sudo umount /dev/sdb2

sudo bmaptool copy \
  ~/poky/build-rpi/tmp/deploy/images/raspberrypi4/core-image-base-
raspberrypi4.rootfs-20250421093029.wic.bz2 \
  /dev/sdb

dmesg | tail -n 20
sudo dmesg | tail -n 20

sudo picocom -b 115200 /dev/ttyUSB0
```

Week 5 — Boot issues & UART enablement

Reflashed images; Windows didn't recognize the Pi partitions initially; later confirmed enable_uart=1 in config.txt.

Confirmed Yocto's boot banner on serial:

```
Poky (Yocto Project Reference Distro) 5.0.8 raspberrypi3-64 /dev/ttyS0
```

Login attempts with pi failed (as expected on Yocto).

Fix: For development, use **root** on console by adding EXTRA_IMAGE_FEATURES += "debug-tweaks", or create a pi user at image build time.

Week 6 — Raspberry Pi Imager test & clarification

Used **Raspberry Pi Imager** and its "customization" (username pi, SSH) but it **does not affect Yocto images** → still failed to login with pi.

Verified CP210x settings in Windows **Device Manager** (COM3 @ 115200).

PuTTY connected; login remained "incorrect" until we switched to **root** (or baked a pi user via Yocto).

Week 7 — End-to-end image & next steps

Final flash of our own Yocto image (Pi 3 and Pi 4 variants) to SD;

Serial console working (COM3, 115200, 8N1);

Plan for next sprint: deploy **BusGuard** daemon & CLI via meta-busguard, wire the sensor + Guardian Pi, run functional/negative tests, export event logs (CSV/JSON).

Troubleshooting Notes

Disk space: allocate **≥200 GB** to the VM for stress-free Yocto builds.

Ubuntu 24: AppArmor/user-namespaces can block BitBake; **Ubuntu 22** avoids this.

SD reader: some readers silently fail under load; replace if in doubt.

Serial permissions: add user to dialout or use sudo picocom.

Yocto login: use **root** (with debug-tweaks) or bake a user in the image; **RPi Imager customisation does not modify Yocto images**.

UART conflicts (Pi 3): consider dtoverlay=miniuart-bt if BT grabs the PL011.

A tiny test (apps/master-demo.py):

```
import time, argparse
from smbus2 import SMBus

p = argparse.ArgumentParser()
p.add_argument("--bus", type=int, default=1)
p.add_argument("--addr", type=lambda x:int(x,0), default=0x48)
p.add_argument("--rate", type=float, default=5.0)
args = p.parse_args()
dt = 1.0 / args.rate

with SMBus(args.bus) as i2c:
    while True:
        try:
            i2c.write_byte(args.addr, 0x00)      # point to register 0x00
            v = i2c.read_byte(args.addr)         # read 1 byte
            i2c.write_byte_data(args.addr, 0x02, (v+1) & 0xFF)  # test write
        except Exception as e:
            print("master-demo:", e)
        time.sleep(dt)
```

Conclusions

- Yocto builds are now **repeatable** (Ubuntu 22), SD images are flashed reliably (bmaptool), and **serial console** works (115200, 8N1).
- Immediate next work items:
    1. Complete meta-busguard layer; build busguard-image.
    2. Wire Application Pi + sensor and Guardian Pi to the same I²C bus (shared SDA/SCL/GND).
    3. Validate normal traffic and execute negative tests; export CSV/JSON logs.
    4. Optional: add a web dashboard or remote syslog/MQTT streaming.

With these steps, the **Guardian Pi** will continuously supervise the I²C bus and provide actionable, time-stamped anomaly reports, making it suitable for embedded scenarios where silent communication errors are rare but **high-impact**.