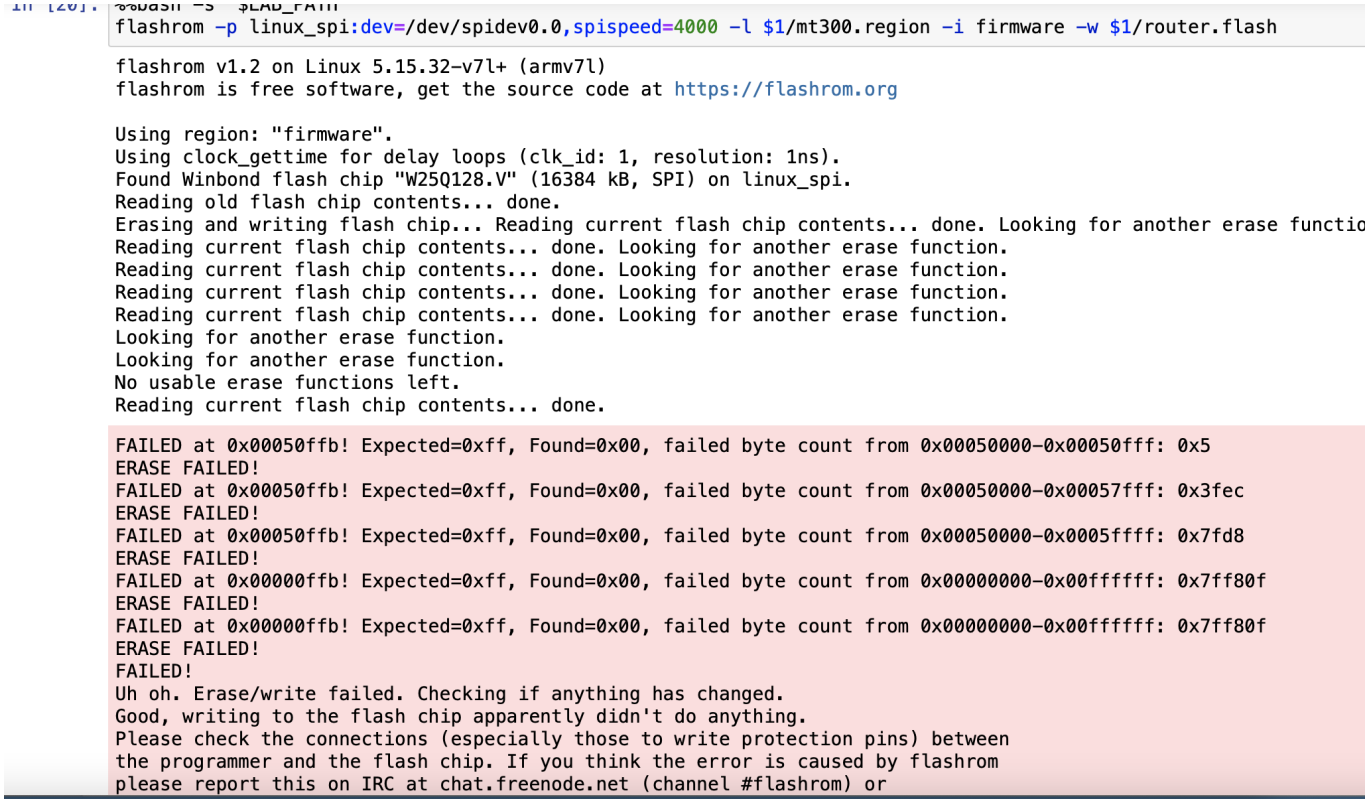
# Router Reflashing Issues / Notes

Notes and Information from a June training Offering

## Problem Statement / Technical Details

- Router PCB Rev: V1.2
- Flash Chip: Winbond 25Q128JVSQ/2142
    - Datasheet
- Problem: When attempting to reflash via flashrom, a student is seeing the following error



```
flashrom -p linux_spi:dev=/dev/spidev0.0,spispeed=4000 -l $1/mt300.region -i firmware -w $1/router.flash

flashrom v1.2 on Linux 5.15.32-v7l+ (armv7l)
flashrom is free software, get the source code at https://flashrom.org

Using region: "firmware".
Using clock_gettime for delay loops (clk_id: 1, resolution: 1ns).
Found Winbond flash chip "W25Q128.V" (16384 kB, SPI) on linux_spi.
Reading old flash chip contents... done.
Erasing and writing flash chip... Reading current flash chip contents... done. Looking for another erase functio
Reading current flash chip contents... done. Looking for another erase function.
Reading current flash chip contents... done. Looking for another erase function.
Reading current flash chip contents... done. Looking for another erase function.
Reading current flash chip contents... done. Looking for another erase function.
Looking for another erase function.
Looking for another erase function.
No usable erase functions left.
Reading current flash chip contents... done.

FAILED at 0x00050ffb! Expected=0xff, Found=0x00, failed byte count from 0x00050000-0x00050fff: 0x5
ERASE FAILED!
FAILED at 0x00050ffb! Expected=0xff, Found=0x00, failed byte count from 0x00050000-0x00057fff: 0x3fec
ERASE FAILED!
FAILED at 0x00050ffb! Expected=0xff, Found=0x00, failed byte count from 0x00050000-0x0005ffff: 0x7fd8
ERASE FAILED!
FAILED at 0x00000ffb! Expected=0xff, Found=0x00, failed byte count from 0x00000000-0x00ffffff: 0x7ff80f
ERASE FAILED!
FAILED at 0x00000ffb! Expected=0xff, Found=0x00, failed byte count from 0x00000000-0x00ffffff: 0x7ff80f
ERASE FAILED!
FAILED!
Uh oh. Erase/write failed. Checking if anything has changed.
Good, writing to the flash chip apparently didn't do anything.
Please check the connections (especially those to write protection pins) between
the programmer and the flash chip. If you think the error is caused by flashrom
please report this on IRC at chat.freenode.net (channel #flashrom) or
```

This results in the following router behavior

- No LED activity
- No UART
- Complete failure to boot/reflash via flashrom

The following table shows the voltages of the EEPROM pins after this error occurs and the router is powered on via USB:

| Pin | Voltage | Usage |
|---|---|---|
| 1 | 53mV | CS |
| 2 | 3.26V | DO |
| 3 | 3.3V | WP |
| 4 | 0V | GND |
| 5 | 0V | DI |
| 6 | 1.5V | CLK |
| 7 | 3.3V | HOLD |
| 8 | 3.3V | VCC |

Based on the state of these pins, we can assume that the main CPU is trying to communicate with the EEPROM (CS pulled low, CLK at 1.5V).

This is a problem for us if we want to reflash the chip due to CS/CLK contention; we also have to investigate the issue of the erase operation initially failing.

## Relevant GH/Mailing List Issues

Other folks have seen this issue when using flashrom with this family of chips; solutions range from using shorter wires to adding a capacitor - it is ultimately dismissed as user error.

- Similar Erase Problem
- Open GH Issue (also using Pi)
- Folks asking for chip support

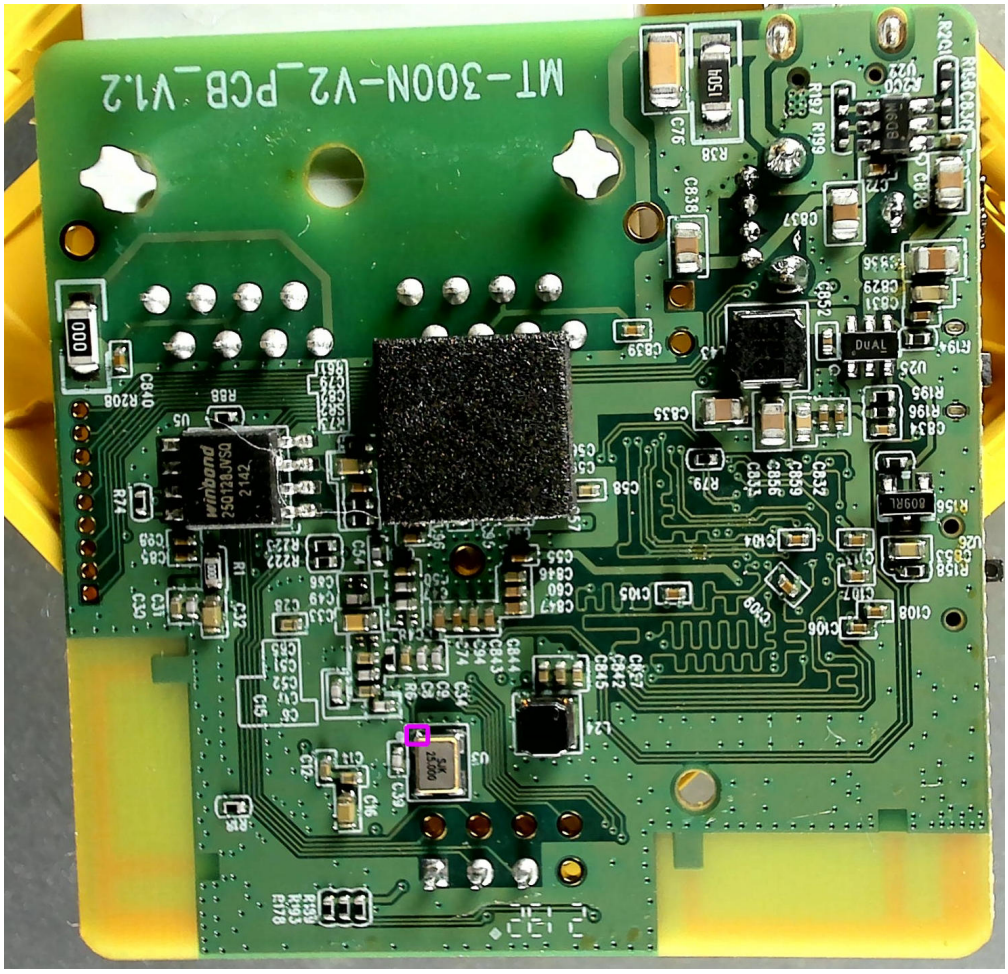
## Holding the Router in Reset

Before we can debug *why* erases aren't working, we need to find a way to stop the CPU from accessing the EEPROM while in this error state; typically, this is done with a CPU reset line.

According to the datasheet, pin 138 is `PORST_N - Power-On Reset`. It is unclear without desoldering the CPU to see where the tiny trace goes. For now, we can stop the CPU from booting via the clock grounding method. By grounding the oscillator on startup, we prevent the CPU from being able to boot correctly and subsequently communicate with / manipulate the EEPROM.

Steps to reproduce:

1. Connect ground from the UART connector to ground on the raspberry pi
2. Power on the router while pulling the following pad to GND using a jumper wire; in testing, the pad was held low for approximately ten seconds before removing it.




**Note:** Make sure you have a common ground between your router and the Pi before doing this; an easy one to use is the one on the UART header.

The state of the SPI flash pins in this "reset" mode can be seen below:

- **Note:** These did vary across resets; the core thing here is that the CLK line is *not* at 1.5V, most successful attempts occurred when the clock was idling at 3.3V.

| Pin | Voltage | Usage |
|---|---|---|
| 1 | 30mV | CS |
| 2 | .9V | DO |
| 3 | 3.3V | WP |
| 4 | 0V | GND |
| 5 | 2.1mV | DI |

| Pin | Voltage | Usage |
|---|---|---|
| 6 | 3.3V | CLK |
| 7 | 3.25V | HOLD |
| 8 | 3.24V | VCC |

# Flashrom in Reset Mode

Flashrom is producing *very* inconsistent results with this flash chip; we are getting inconsistent reads and writes. This raises multiple questions, namely:

1. Why is the erase failing?
2. Why is it erasing data outside of the specified range?
3. What is causing these reads to fail?

One of the first troubleshooting steps with SPI flash reads/writes in circuit is to shorten wires and decrease clock speed. Unfortunately, these same inconsistent results were seen regardless of the `spispeed` variable; even going as low as 100 caused issues.

In practice, it looks like sector/block erases were not working correctly. Therefore, as a sanity check, we will implement our program that will attempt to erase/read/write the chip.

# Python Notes / Pi Documentation

## Prerequisites

1. Stock GPIO settings
2. Chip in a reset state via holding oscillator
3. Use a clock speed of 4000 Hz (this is hardcoded in the script)

## Replicating the Fix

### Wiring Modification

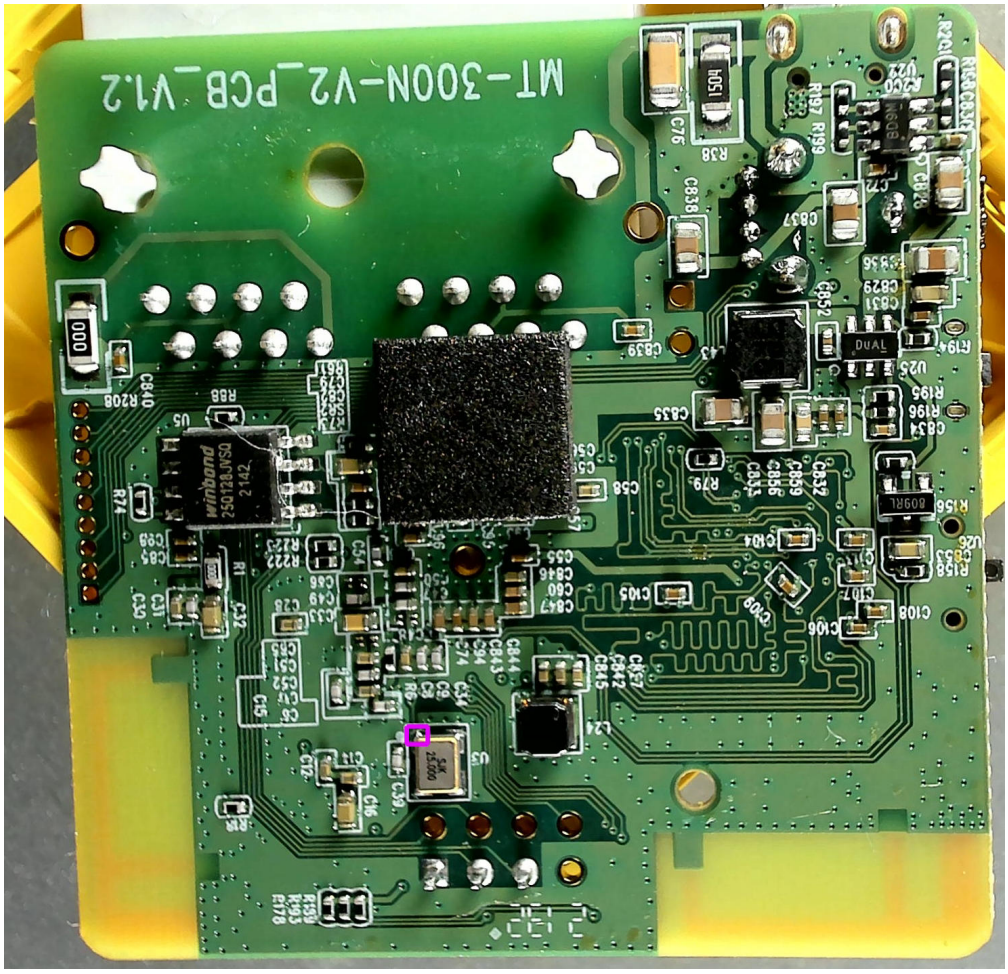
1. Remove the 100 OHM pulldown resistor from the CS line.
2. Add a 3.3V jumper to pin 8 (VCC) of the clip
    1. **Note:** This may not be entirely necessary, but I wanted to make sure that the voltage did not drop below a functional range
3. Connect the ground from the UART connector (Pin 1) to the ground on your Pi

1. We need common ground established to short the oscillator with our other ground wire

## Reset Mode

To properly run our script, we need to make sure that the router is no longer accessing the SPI flash; we can do this as shown below:

1. Power on the router while holding the following pad to GND using a jumper wire
    1. **Note:** In testing, I held the oscillator low for 10-15 seconds before attaching the clip and running the python script.




By holding this line low, we keep the processor from fully booting. This lets us easily access the SPI flash as the CPU is in a faulty state.

## Running the Script

We can rule out signal integrity being a potential issue by checking the ID via the `0x9F` command. Once this was tested, I tried read commands with our known working eeprom.

After that, I tested each erase instruction on the router flash and read the data back out of the chip. Methods involving sectors/blocks erase very inconsistently; the chip erase command, however, seemed to work. After that, all that was needed was to implement a page program function, which appears to be enough to write a UBoot image back to the chip.

The python script does the following:

1. Ping the chip for its chip ID (command `0x9F`)
2. Erase the *entire* chip (command `0xC6`)
3. Reflash the UBoot partition using the page program command

Using this, if we issue the Chip Erase command (0xC6) manually, it seems like we can then reprogram the UBoot partition and break into it. From here, we can update the firmware via the TFTP method.

The python script has been developed to read a binary called `uboot.bin` from the local directory and flash those contents to the router; you can use the one from your original extraction or the one located in `/home/pi/labs/firmware/backup-images/v2`.

The script can is run as shown below:

```
python3 spi.py
```

You should initially see the chip ID get returned as well as the size of your `uboot.bin` binary:

```
pi@voidstar:~/labs/firmware $ python3 spi.py
327680
EEPROM ID: EF4018
```

After this, the chip will be erased and reflashed. Note that it will take a decent amount of time to erase the entire chip; after erasing, you should see the following:

```
Erased ENTIRE chip, Are you mad?!
```

After this, you will see page program messages begin to scroll by; note that these should be occurring roughly once a second; if a page program operation stalls for more than 10-15 seconds, re-enter the reflashing mode and try to rerun the script.

Once the chip is programmed, we attempt to read the UBoot partition back out of the flash.

With the reflash/read completed, you should be able to power your router back up with the UART connected and break into UBoot!

## Conclusion / Follow Up

Something about the flashrom implementation appears to be struggling to erase the chip properly. Unfortunately, the only working method for this particular chip seemed to be erasing the entire chip. We then write back the bootloader using the page program command, and that allows us to break into UBoot and reflash over TFTP.

# Appendix / Code

```
import os
import sys
import time
import spidev

'''
Docs for spidev can be found here: https://pypi.org/project/spidev/
This is a test program hacked together in an evening to help recover a
router, reflashing seems to work but has not been tested with large files
Use at your own risk!
'''

'''
get_id
Return the ID of the SPI flash chip, use this to make sure that comms are
good
Arguments:
    spi - spidev object
'''
def get_id(spi):
    # 0x9F is our command, we add three additional 0xFF's because we expect
three bytes to be returned
    msg = [0x9F,0xFF,0xFF,0xFF]
    result = spi.xfer2(msg)
    id = result[1:]
    print("EEPROM ID: ",end='')
    for element in id:
        print("{:X}".format(element),end='')
    print()
```

```
'''
enable_writes
Used to enable writes on the SPI flash, fire and forget!
Arguments:
    spi - spidev object
'''
def enable_writes(spi):
    spi.xfer2([0x6])

'''
get_sr
Returns the contents of the status register for the flash chip
Arguments:
    spi - spidev object
'''
def get_sr(spi):
    return spi.xfer2([0x5,0xff])

'''
page_program
Write 256 bytes back to the target, address needs to be 256 byte alligned and
you should of course provide 256 bytes to be programmed
Arguments:
    spi - spidev object
    addr - address of page to program
    data - data to write at the specified page
'''
def page_program(spi,addr,data):
    enable_writes(spi)
    sr_val = get_sr(spi)
    while(((sr_val[1] & 2) >> 1) != 1):
        sr_val = get_sr(spi)
    write = [0x2]
    write.append(addr >> 16 & 0xFF)
    write.append(addr >> 8 & 0xFF)
    write.append(addr & 0xFF)
    for element in data:
        write.append(element)
    spi.xfer2(write)
    # Wait for the busy bit to clear
    while(((sr_val[1] & 1)) != 0):
        sr_val = get_sr(spi)
    print(f"Programmed page at {addr:x}")

'''
erase_block
Erases one block of memory on the flash chip (32kb)
```

```
    Arguments:
        spi - spidev object
        addr - address of block to erase
    '''
    def erase_block(spi,addr):
        enable_writes(spi)
        sr_val = get_sr(spi)
        while(((sr_val[1] & 2) >> 1) != 1):
            sr_val = get_sr(spi)
        erase = [0x52]
        erase.append(addr >> 16 & 0xFF)
        erase.append(addr >> 8 & 0xFF)
        erase.append(addr & 0xFF)
        spi.xfer2(erase)
        sr_val = get_sr(spi)
        # Wait for the busy bit to clear
        while(((sr_val[1] & 1)) != 0):
            sr_val = get_sr(spi)
        print(f"Erased block at addr {addr:X} complete")

    '''
    erase_sector
    Erases one sector of memory on the flash chip (4kb)
    Arguments:
        spi - spidev object
        addr - address of sector to erase
    '''
    def erase_sector(spi,addr):
        enable_writes(spi)
        sr_val = get_sr(spi)
        while(((sr_val[1] & 2) >> 1) != 1):
            sr_val = get_sr(spi)
        erase = [0x20]
        erase.append(addr >> 16 & 0xFF)
        erase.append(addr >> 8 & 0xFF)
        erase.append(addr & 0xFF)
        spi.xfer2(erase)
        sr_val = get_sr(spi)
        # Wait for the busy bit to clear
        while(((sr_val[1] & 1)) != 0):
            sr_val = get_sr(spi)
        print(f"Erased sector at addr {addr:X} complete")

    '''
    chip_erase
    Erase the entire chip!
    Arguments:
        spi - spidev object
```

```
'''
def chip_erase(spi):
    enable_writes(spi)
    sr_val = get_sr(spi)
    while(((sr_val[1] & 2) >> 1) != 1):
        sr_val = get_sr(spi)
    spi.xfer2([0xC7])
    sr_val = get_sr(spi)
    # Wait for the busy bit to clear
    while(((sr_val[1] & 1)) != 0):
        sr_val = get_sr(spi)
    print(f"Erased ENTIRE chip, are you mad?!")

'''
read
Reads one 256 byte page from the EEPROM
Arguments:
    spi - spidev object
    address - 256 byte alligned address
    count - how much data to read
'''
def read(spi,addr,count):
    print(f"Reading from address: {addr:x}")
    msg = [0x3]
    msg.append(addr >> 16 & 0xFF)
    msg.append(addr >> 8 & 0xFF)
    msg.append(addr & 0xFF)
    for x in range(0,count):
        msg.append(0xFF)
    result = spi.xfer2(msg)
    # Remember - the first four bytes of our transfer will be empty, because
these are read while we're senging the initial read command, so we'll cut
those off before returning.
    return result[4:]

'''
read_file_for_flash
Hacky way to make sure that our  input files are 256 byte aligned
Arguments:
    path - path to input file to flash back to eeprom
'''
def read_file_for_flash(path):
    uboot_data = open(path,'rb').read()
    test = bytes(uboot_data)
    print(len(test))
    if len(test) % 256 != 0:
        print("Please provide properly alligned (256 byte) data to write...")
        sys.exit()
```

```
    return test

if __name__ == "__main__":
    uboot_data = read_file_for_flash("uboot.bin")
    bus = 0
    device = 0
    spi = spidev.SpiDev()
    spi.open(bus, device)
    spi.max_speed_hz = 4000
    spi.mode = 0
    # Ping the chip and wipe it
    get_id(spi)
    chip_erase(spi)
    # Write back our UBoot image
    bytes_written = 0
    bytes_to_write = len(uboot_data)
    while(bytes_written < bytes_to_write ):

page_program(spi,bytes_written,uboot_data[bytes_written:bytes_written+256])
        bytes_written += 256
    print("UBoot binary written!")
    # Read it back out, compare with the original uboot.bin file to make sure
that everything worked
    bytestr = b''
    total_read = len(uboot_data)
    bytes_read = 0
    while(bytes_read < total_read):
        bytestr +=  bytes(read(spi,bytes_read,0x100))
        bytes_read += 0x100
    with open('uboot-flash-test.bin','wb') as ofile:
        ofile.write(bytestr)
```