# Reverse engineering of a universal remote control

Bouali Bechir

Advisor: Professor Aurélien Francillon

February 9, 2019

**Abstract**

This project explored an non-destructive method to dump a firmware of Remote Control Copier , reverse engineering the binary package using disassembler and expose how rolling codes can be broken or brute forced by a low cost microcontroller. To do so , we used a STLINK-v2-1 as a debugger for ARM cortex-M0 processor combined with openOCD to dump the firmware . This project contributes code to bypass some hardware protection features. We then used radare2 to reverse engineer the binary code.

## 1. Introduction

Remote controls (RC) for gate openers and garage doors are split in two categories: fixed codes, and rolling codes. Concerning rolling codes, several commercially available devices (Remote Control Copiers (RCC)) can clone original keyfob and generate rolling codes on the fly, replacing completely the original .

Purpose of this project is to reverse engineer a RCC, which embed a Bluetooth Low Energy chip and a RF transceiver, and is controlled by a Smartphone.

## 2. Related work

This section discusses work related to our project that provides useful background knowledge.

In project similar , Kris Brosch presented firmware dumping technique for an ARM Cortex-M0 to bypass the security feature that prevent firmware from read through the debugging interface.The technique consist of using a load instruction and to load data from memory.[1]

## 3. Analysis

The RRC that we will work on it ,Parkmatch[2],used to open parking doors. So in this section we conduct an analysis of the chip in order to find the best way to interact with it. First, we detail the component of the chip .Then we find the pins out that we will use later in debugging .

### 3.1 Component of the chip

- Micro controller : nRF51822-QFAC is an ultra-low power 2.4 GHz .It is built around the 32-bit ARM® Cortex™-M0 CPU with 256 KB flash and 32 KB RAM

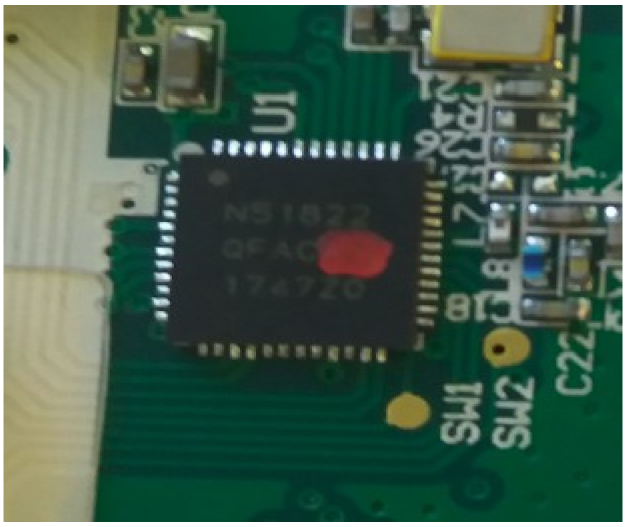


Figure1: micro controller nRF51822

- TheCC110L : RF transceiver for the 300–348 MHz

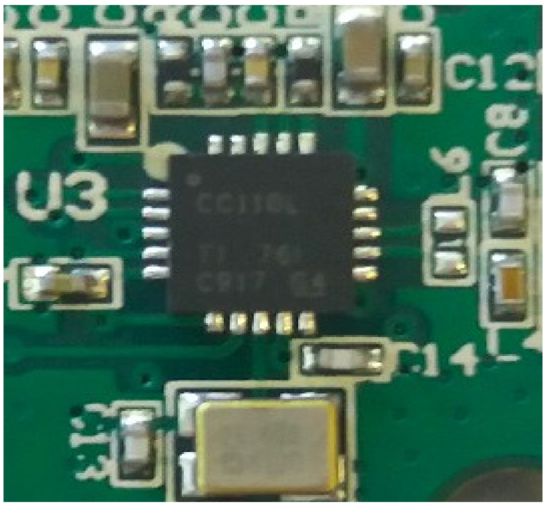


Figure 2: RF transceiver CC110L

- external connector with 6 pins connected to pins of the micro processor .This connector will be useful when we connect the debug adapter to micro processor .

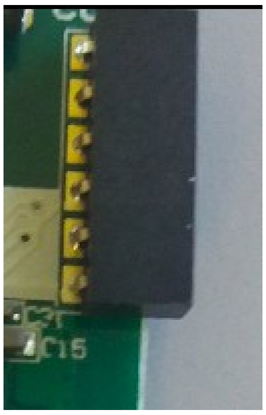


Figure 3: external connector

## 3.2 Debug interface

The nRF51822-QFAC is a product constructed by NORDIC SEMICONDUCTOR company [3]. After studying the data sheet[4] of this product , I found that The nRF chip could be connected to a Debug unit through an interface called Serial Wire Debug (SWD).

This interface defined by 4 pins:

- SWDCLK: clock line
- SWDIO: data line
- VDT: voltage detection
- GND: ground

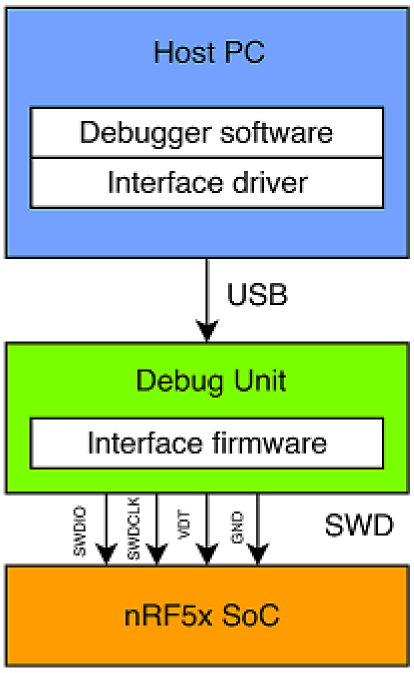


Figure 4: serial wire debugging connection

## 3.3 Pin out Assignment

### 3.3.1 Microprocessor pin out

As I mentioned in the previous section that the debug interface SWD work with 4 pins , so in this section I first figure out the pin out of micro processor then I match them with the external pins.

Using the data sheet section that describes the pin assignment and the pin functions:

- SWDLCK : Pin 24
- SWDIO : Pin 23
- GND : Pin 13
- VDT : Pin 1

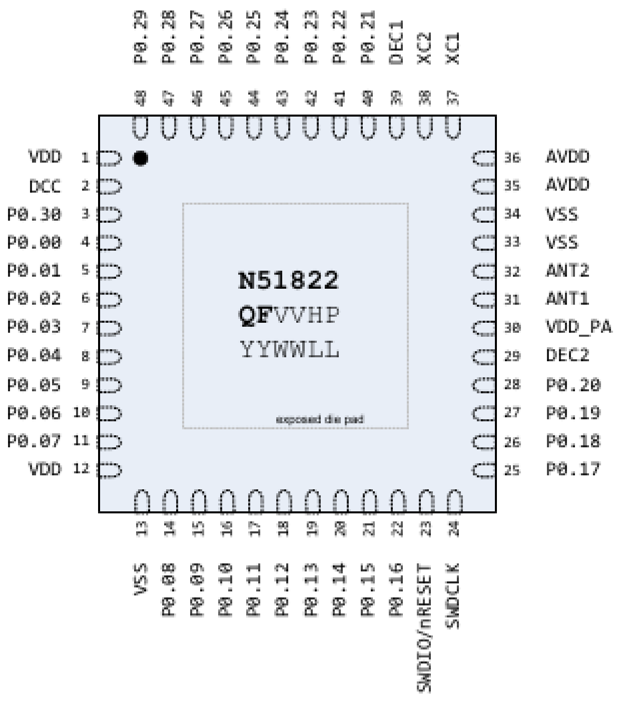


Figure 5 : Pin assignment

### 3.3.2 External connector pins assignment

The most difficult part is to match between the microprocessor pin out and the external connector's pins because the chip is too small and it 's hard to follow the printed buses .So after a lot of try I manage to take a good match like this.

| **SWD pins** | **Microcontroller’s pins** | **Connector’s pins** |
|---|---|---|
| SWDLCK | Pin 24 | Pin 2 |
| SWDIO | Pin 23 | Pin 1 |
| VDT | Pin 1 | Pin 4 |
| GND | Pin 13 | Pin 3 |

Table 1 : matching between microcontroller and connector pins
Pin1 of connector is where CON1 written on borad

## 4. Methodology

According to the previous section , We need a debug adapter that support SWD,such as STLINk-V2-1 of a nucleo board ,that I will use for debugging .

### 4.1 Setup connection between debug adapter and the board

According to the guide of the STLINK-V2-1 the SWD pins are like this

- VDT : Pin1
- SWDCLK: Pin 2
- GND : Pin 3
- SWDIO: Pin4

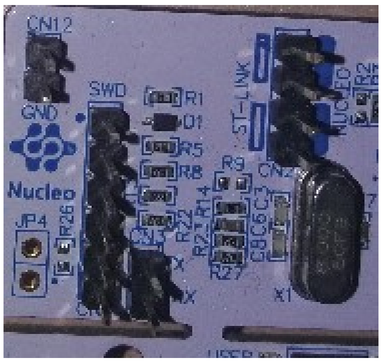


Figure 6 : STLINK-V2-1 SWD pins

Then we can connect the debug adapter to the chip as follows

| SWD pins | STLINK-V2-1 pins | CONNECTOR pins |
|---|---|---|
| SWDCLK | Pin 2 | Pin 2 |
| SWDIO | Pin 4 | Pin 1 |
| VDT | Pin 1 | Pin 4 |
| GND | Pin 3 | Pin 3 |

Table 2 :matching between STLINK-V2-1 and connector pins

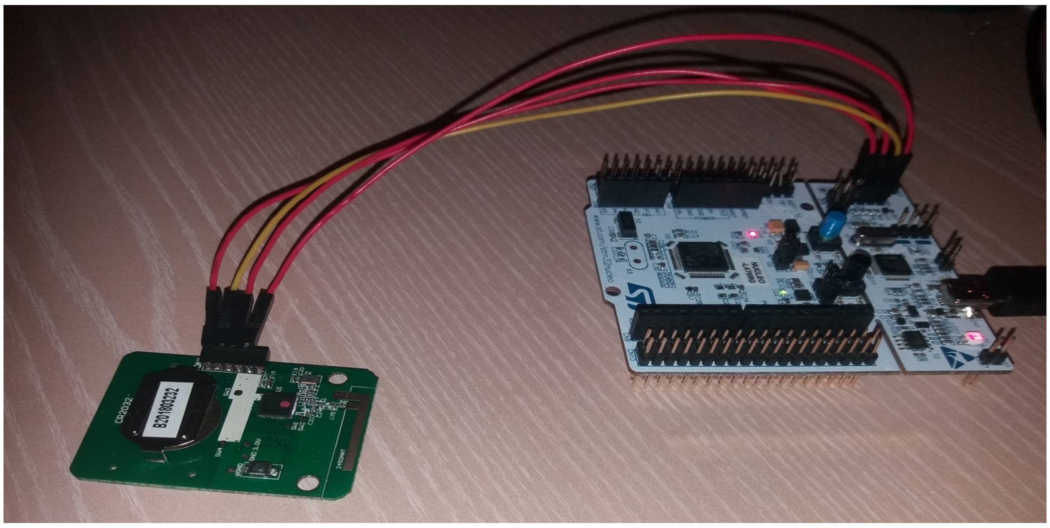
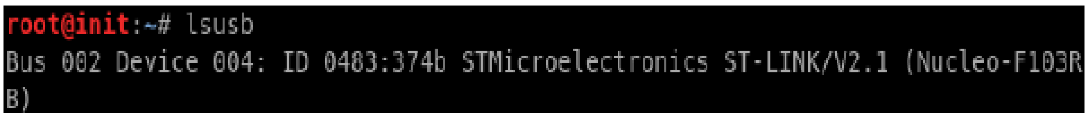


Figure 7: Wires connecting STLINKV2-1 to the nRF51822 micorcontroller

## 4.2 Configuration of debugger software

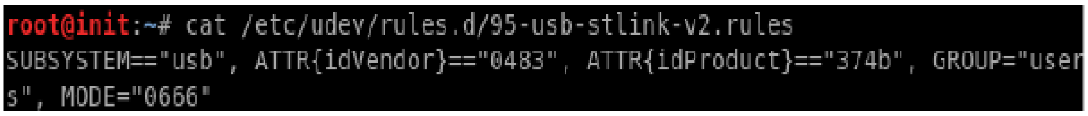
As a debugger software I choose to work with openOCD[5] is an on chip debugging, in-system programming and boundary-scan testing tool for ARM and MIPS systems. But before working with openOCD we need to enable user access to debugger. After connecting nucleao board to PC , we check characteristics of STLINK-V2-1 as follows :



```
root@init:~# lsusb
Bus 002 Device 004: ID 0483:374b STMicroelectronics ST-LINK/V2.1 (Nucleo-F103R
B)
```

Figure 8: output of “lsusb” command

then I add those information to a configuration file called 95-usb-stlink-v2.rules under "*etc*/udev/rules.d" as follows:



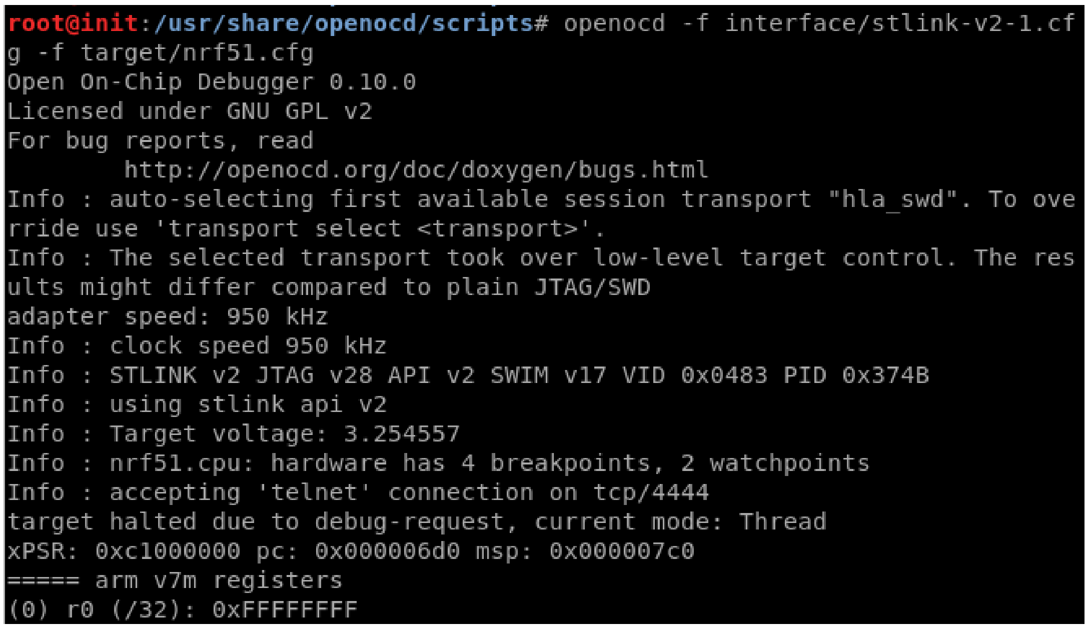
```
root@init:~# cat /etc/udev/rules.d/95-usb-stlink-v2.rules
SUBSYSTEM=="usb", ATTR{idVendor}=="0483", ATTR{idProduct}=="374b", GROUP="user
s", MODE="0666"
```

Figure 9: The content of 95-usb-stlink-v2.rules

Now as the environment setup is finished we can connect to the micro controller using openocd as follows

# openocd -f interface/stlink-v2-1.cfg -f target/nrf51.cfg

- -f interface/stlink-v2-1 : use the configuration file of stlink-v2-1 interface
- -f target/nrf51.cfg : use the configuration file of nRF51822 microprocessor

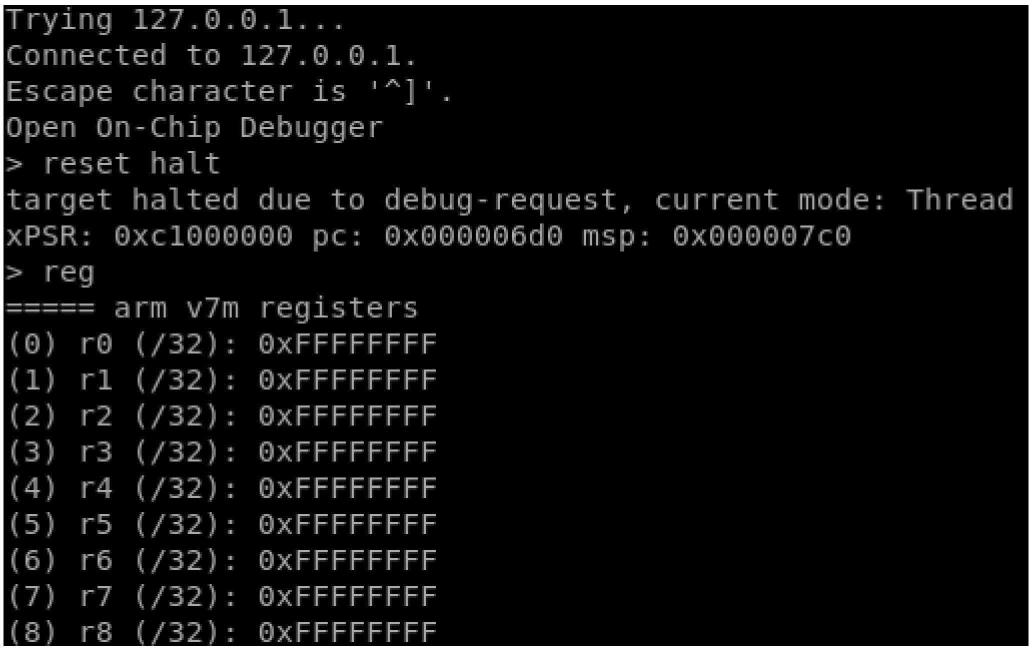


```
root@init:/usr/share/openocd/scripts# openocd -f interface/stlink-v2-1.cf
g -f target/nrf51.cfg
Open On-Chip Debugger 0.10.0
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
Info : auto-selecting first available session transport "hla_swd". To ove
rride use 'transport select <transport>'.
Info : The selected transport took over low-level target control. The res
ults might differ compared to plain JTAG/SWD
adapter speed: 950 kHz
Info : clock speed 950 kHz
Info : STLINK v2 JTAG v28 API v2 SWIM v17 VID 0x0483 PID 0x374B
Info : using stlink api v2
Info : Target voltage: 3.254557
Info : nrf51.cpu: hardware has 4 breakpoints, 2 watchpoints
Info : accepting 'telnet' connection on tcp/4444
target halted due to debug-request, current mode: Thread
xPSR: 0xc1000000 pc: 0x000006d0 msp: 0x000007c0
===== arm v7m registers
(0) r0 (/32): 0xFFFFFFFF
```

Figure 10: success connection between the deugger and the chip

Once the OpenOCD daemon is started, we need to use telnet to connect to the device. For this we can connect to the local machine on port 4444, and can directly start issuing commands to the device

# telnet 127.0.0.1 4444



```
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
Open On-Chip Debugger
> reset halt
target halted due to debug-request, current mode: Thread
xPSR: 0xc1000000 pc: 0x000006d0 msp: 0x000007c0
> reg
===== arm v7m registers
(0) r0 (/32): 0xFFFFFFFF
(1) r1 (/32): 0xFFFFFFFF
(2) r2 (/32): 0xFFFFFFFF
(3) r3 (/32): 0xFFFFFFFF
(4) r4 (/32): 0xFFFFFFFF
(5) r5 (/32): 0xFFFFFFFF
(6) r6 (/32): 0xFFFFFFFF
(7) r7 (/32): 0xFFFFFFFF
(8) r8 (/32): 0xFFFFFFFF
```

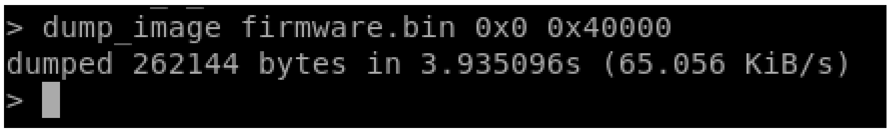
Figure 11 : direct interaction with the chip using telnet

## 5. Results

In this section I will present several method that I tried to dump the firmware from memory accompanied with problems and solutions.

### 5.1 Direct method

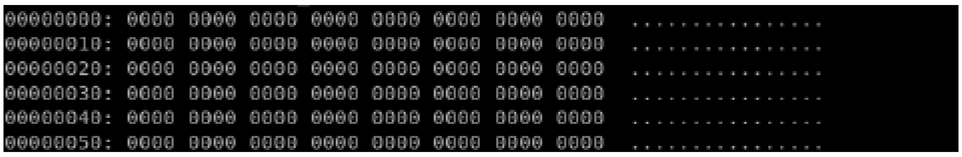
OpenOCD provide a commande  could  be used to dump the firmware from the target .



```
> dump_image firmware.bin 0x0 0x40000
dumped 262144 bytes in 3.935096s (65.056 KiB/s)
> 
```

- dump_image : openocd command
- firmware.bin : output file
- 0x0 : the address from where you want to begin dumping
- 0x40000 : size ofthez firmware is about 256KB

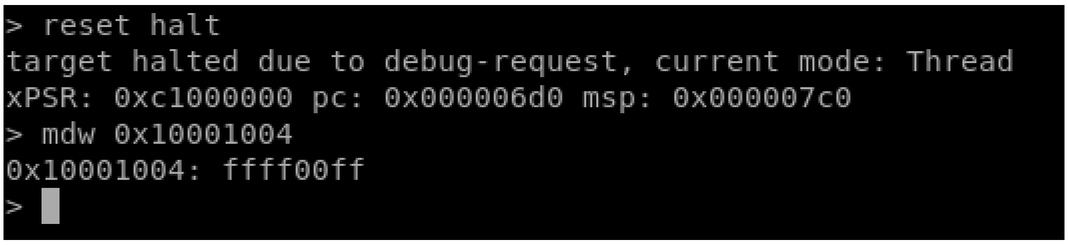
Unfortunately, the result file  was full with 0x0 which mean that the firmware.bin is empty and the dumping operation was unsuccessful.



```
00000000: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Figure 12 : The hexadecimal format of the content of firmware.bin

The problem ,here, that there is a protection mechanism was enabled .To be more sure we can read the content of a register named   "RBPCONF" (readback protection)



```
> reset halt
target halted due to debug-request, current mode: Thread
xPSR: 0xc1000000 pc: 0x000006d0 msp: 0x000007c0
> mdw 0x10001004
0x10001004: ffff00ff
> 
```

Figure 13 : read the content of the readback register

As we see the content of the register is “0xffff00ff” and according to the data sheet this mean that “protection all” (PALL) is enabled. This security feature  is responsible for preventing us from accessing the code section and subsequently causing our read commands to return zeros.

## 5.2 Bypass protection

The best way to read from the protected memory is to find a load instruction that we can use it to load data from main memory to general purpose registers of CPU in a legal way . First step will be searching for load instruction in a specific register. Second step overwrite that specific register with the value of address that you want to load from memory. Finally jump to that load instruction and read data loaded to that specific register.

**Step1**: To find the load instruction each time overwrite all the registers with value 0x4 then doing a step then  check if the content of one of registers is what should be in memory. According to the vector table of cortex-M0 the reset vector is at address 0x4

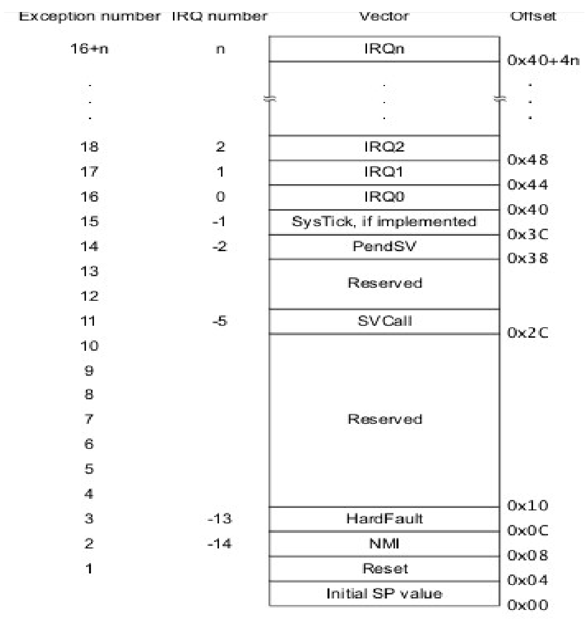


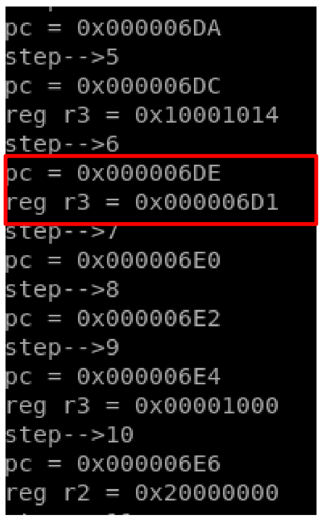
Figure 14: vector table

I wrote a script that help to find load instruction

```
import telnetlib
import re
import struct

HOST = "127.0.0.1"
PORT = "4444"

s = telnetlib.Telnet(HOST , PORT)
s.set_debuglevel(0)
s.read_until(">")

def overwrite_registers():

 for i in xrange(13):

    s.write("reg r" + str(i) + " " +  hex(0x04) + "\n")
    s.read_until(">")

def print_registers():

 for i in xrange(13):

    s.write("reg r" + str(i) + "\n")
    data = re.findall(r'0x[0-9afA-F]+', s.read_until(">"))
    if (data[0] != "0x00000004") and (data[0] != "0xFFFFFFFF"):
        print "-----------reg r" + str(i) + " = " + data[0]

s.write("reset halt\n")
```

```
s.read_until(">")

for x in xrange(0,100,2):

    print "====step==== " + str(x)

    pc = int("0x6d0",16)
    pc += x
    print " pc ",hex(pc)
    s.write("reg pc " + hex(pc) +"\n")
    s.read_until(">")

    overwrite_registers()

    s.write("step\n")
    s.read_until(">")

    s.write("reg pc\n")
    data = re.findall(r'0x[0-9afA-F]+', s.read_until(">"))
    print "pc = " + data[0]

    print_registers()
```

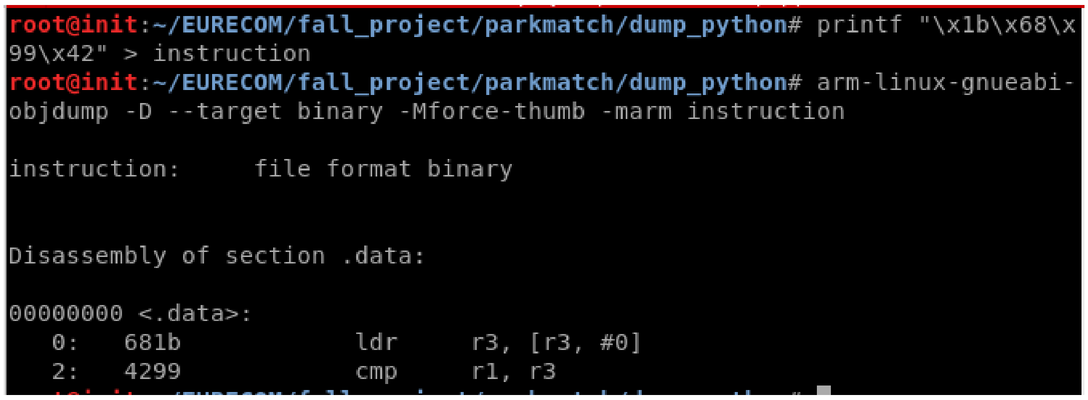
I found that after doing 5 steps and overwrite all registers to 0x4 ,then do step the register r3 = 0x6D1 which is the reset vector because when I reset the chip ,the PC is set to 0x6D0 and because the Coretx-M0 operates in thumb mode the least significant bit is set in the reset vector changing 0 to 1.



```
pc = 0x000006DA
step-->5
pc = 0x000006DC
reg r3 = 0x10001014
step-->6
pc = 0x000006DE
reg r3 = 0x000006D1
step-->7
pc = 0x000006E0
step-->8
pc = 0x000006E2
step-->9
pc = 0x000006E4
reg r3 = 0x00001000
step-->10
pc = 0x000006E6
reg r2 = 0x20000000
```

Figure 15: part from output of the python script

To be more sure that this the right load instruction I used a disassembler to disassemble the content of that register.



```
root@init:~/EURECOM/fall_project/parkmatch/dump_python# printf "\x1b\x68\x
99\x42" > instruction
root@init:~/EURECOM/fall_project/parkmatch/dump_python# arm-linux-gnueabi-
objdump -D --target binary -Mforce-thumb -marm instruction

instruction:     file format binary


Disassembly of section .data:

00000000 <.data>:
   0:   681b            ldr     r3, [r3, #0]
   2:   4299            cmp     r1, r3
```

Figure 15: the result of disassembling  the content of register 3

**step2**: each time I will set PC to 0x6DC and r3 to an address then a step and finally store the content of r3 in a file . So to dump the firmware I wrote this script.

```
import telnetlib
import re
import struct
import time

HOST = "127.0.0.1"
PORT = "4444"

s = telnetlib.Telnet(HOST , PORT)
s.set_debuglevel(0)
s.read_until(">")


f = open("firmware.bin","a+")

s.write("reset halt\n")
s.read_until(">")

for address in xrange(offset,int("0x40000",16),4):

      print "[+]" + hex(address)

      s.write("reset halt\n")
      s.read_until(">")
      s.write("reg pc 0x6DC\n")
      s.read_until(">")

      s.write("reg r3 " + hex(address) + "\n")
      s.read_until(">")

     s.write("step\n")
      s.read_until(">")

      s.write("reg pc\n")
      pc_value = re.findall(r'0x[0-9afA-F]+', show)

      s.write("reg r3\n")
      data = re.findall(r'0x[0-9afA-F]+', s.read_until(">"))
      print "pc = " ,pc_value ," || "+ "r3 = " ,data

      if data:

            f.write(struct.pack("I",int(data[0],16)))

f.close()
```

This time the firmware.bin was not empty and it contain valid data. But the firmware was not totally dumped as after a while the session  the device crash .The cause of this error is  synchronization problem between the clock of the chip and the debugger .

## 5.2 Bypass synchronization problems

**solution 1:**To avoid this problem as a first solution I added “time.sleep(1)” after and before “s.write("reset halt\n")” and “ s.write("step\n")” to the previous script .This solution had improved  the dumping operation as the device crash after dumping a good part from the firmware then it stabilize and it complete the operation . The inconvenient here that the operation take a long time

**solution 2:** Using ykush [6]  board I can activate and disactivate the USB port using python so each time the device crash I rerun the whole process so the device can be reset properly and I start dumping for the address where the device crash .

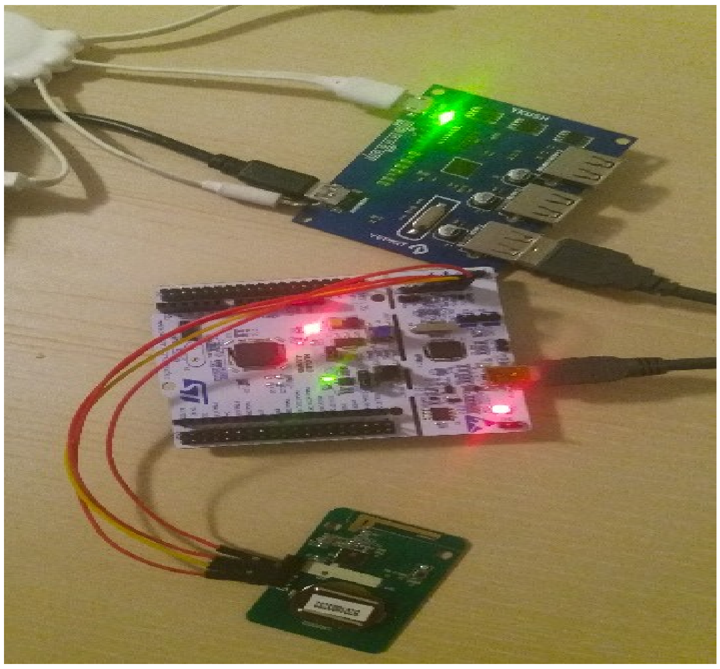


Figure 16 : connection between nucleo and ykush board

In order to try the solution 2 I wrote a python script as follows:

```
import telnetlib
import re
import struct
import time
import pykush
import subprocess

HOST = "127.0.0.1"
PORT = "4444"

confUsbPort = pykush.YKUSH()

offset = 0
address = 0
while address < 262144:

    print "[+] Waiting for USB to be UP"
    time.sleep(1)
    confUsbPort.set_port_state(1, pykush.YKUSH_PORT_STATE_UP)
    time.sleep(1)
    p = subprocess.Popen('openocd -f
/usr/share/openocd/scripts/interface/stlink-v2-1.cfg -f /usr/share/
openocd/scripts/target/nrf51.cfg', shell=True,
stdout=subprocess.PIPE, stderr=subprocess.STDOUT)
```

```
    print "[+] USB Port up"
    time.sleep(3)
    s = telnetlib.Telnet(HOST , PORT)

    s.set_debuglevel(0)
    s.read_until(">")


    f = open("firmware.bin","a+")

    print "====the file opened===="

    for address in xrange(offset,int("0x40000",16),4):


        print "[+]" + hex(address)
        s.write("reset halt\n")
        s.read_until(">")

        time.sleep(1)

        s.write("reg pc 0x6DC\n")
        s.read_until(">")

        time.sleep(1)

        s.write("reg r3 " + hex(address) + "\n")
        s.read_until(">")

        time.sleep(1)

        s.write("step\n")
        s.read_until(">")

        time.sleep(2)

        s.write("reg pc\n")
        pc_value = re.findall(r'0x[0-9afA-F]+', s.read_until(">"))

        print pc_value

        if pc_value != ['0x000006DE']:
            offset = address
            subprocess.Popen('kill -9 p.pid', shell=True,
stdout=subprocess.PIPE, stderr=subprocess.STDOUT)



            time.sleep(0.5)
            confUsbPort.set_port_state(1,
pykush.YKUSH_PORT_STATE_DOWN)
            time.sleep(0.5)
            s.close()
            break

        s.write("reg r3\n")
        data = re.findall(r'0x[0-9afA-F]+', s.read_until(">"))
        print "pc = " ,pc_value ," || "+ "r3 = " ,data
        if data:

            f.write(struct.pack("I",int(data[0],16)))

    f.close()
```

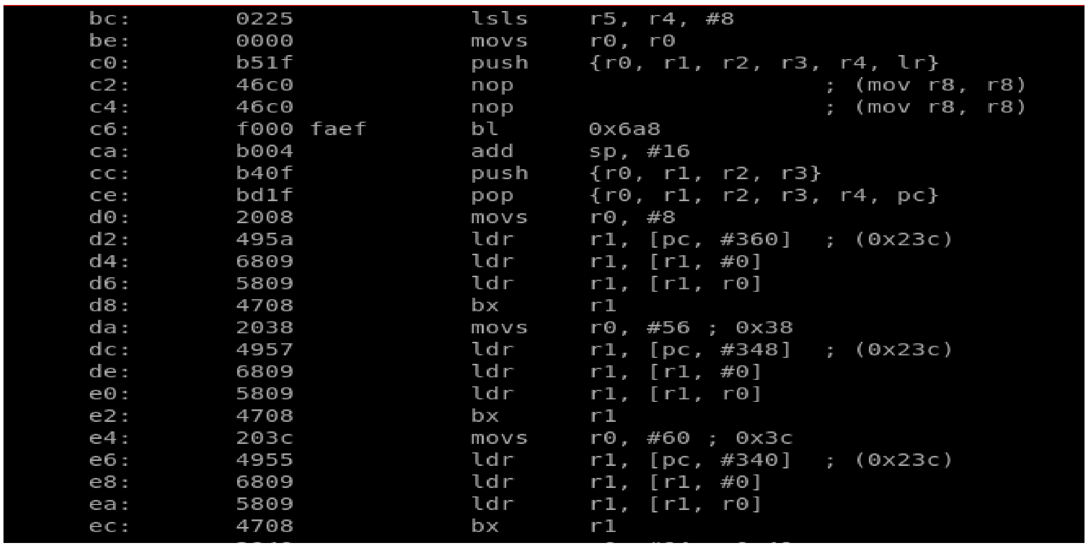
Finally I encounter the same problem  always the device crash after dumping a part of the firmware even with these method.

So for now the best way to dump the firmware is solution1 and each time restart dumping from the address of last crash like this we can get the total firmware, but will take so much time .

## 5.3 disassemble the firmware

After dumping the firmware we can extract the assmebly code from this binary file.

# arm-linux-gnueabi-objdump -D --target binary -Mforce-thumb -marm firmware.bin > firmware.hex



```
      bc:       0225            lsls    r5, r4, #8
      be:       0000            movs    r0, r0
      c0:       b51f            push    {r0, r1, r2, r3, r4, lr}
      c2:       46c0            nop                     ; (mov r8, r8)
      c4:       46c0            nop                     ; (mov r8, r8)
      c6:       f000 faef       bl      0x6a8
      ca:       b004            add     sp, #16
      cc:       b40f            push    {r0, r1, r2, r3}
      ce:       bd1f            pop     {r0, r1, r2, r3, r4, pc}
      d0:       2008            movs    r0, #8
      d2:       495a            ldr     r1, [pc, #360]  ; (0x23c)
      d4:       6809            ldr     r1, [r1, #0]
      d6:       5809            ldr     r1, [r1, r0]
      d8:       4708            bx      r1
      da:       2038            movs    r0, #56 ; 0x38
      dc:       4957            ldr     r1, [pc, #348]  ; (0x23c)
      de:       6809            ldr     r1, [r1, #0]
      e0:       5809            ldr     r1, [r1, r0]
      e2:       4708            bx      r1
      e4:       203c            movs    r0, #60 ; 0x3c
      e6:       4955            ldr     r1, [pc, #340]  ; (0x23c)
      e8:       6809            ldr     r1, [r1, #0]
      ea:       5809            ldr     r1, [r1, r0]
      ec:       4708            bx      r1
```

Figure 17: part from firmware after disassembling

## 6. Further work :

In order to better understand the problem of crashing that happen we can use logic analyzer to determine the quality of connection between the device and the debug adapter. We can try another debug adapter such as Jlink.

As the objective of this project not only dump the firmware but also reverse engineering it . This task could be done using IDA PRO or radare2 which will help in understanding the assembly code of the firmware and help to generate a pseudo code form it.

## 7. Conclusion

This project present a whole procedure to dump a firmware . In which I present a how to find the pin out then how to find the right debugger finally how to bypass some problems as protection mechanism and synchronization .

## References

[1] Kris Brosch , "Firmware dumping technique for an ARM Cortex-M0 SoC",2015. Available http://blog.includesecurity.com/2015/11/NordicSemi-ARM-SoC-Firmware-dumping-technique.html

[2] parkmatch ,Available https://www.parkmatch.eu/fr/

[3] NORDIC SEMICONDUCTOR
Available https://www.nordicsemi.com/Products

[4] nRF51822 data sheet
Available http://infocenter.nordicsemi.com/pdf/nRF51822_PS_v3.1.pdf

[5] openOCD
Avaialable http://openocd.org/

[6] ykush
Available https://www.yepkit.com/products/ykush