

## 20:03 NFC Exploitation with the RF430RFL152 and 'TAL152

by Travis Goodspeed and Axelle Aprille

Lately we've been playing with the RF430FRL152H, a delightful chip from Texas Instruments that combines an MSP430 microcontroller with an ISO15693 NFC transponder. In this short paper, we'll show you a bit about how that chip works, and how to re-program it over the air to emulator other NFC Type V devices.

We'll also learn a little bit about how to reverse engineer medical products that use related chips, such as the RF430TAL152H, getting code execution and complete control of both devices. This article hasn't room for much background information on these medical sensors, and for that you should see our lecture *The Inner Guts of a Connected Glucose Sensor for Diabetes* from Black Alps 2019.

-----

First, a bit of background. The RF430, as we'll call these chips for short, uses an MSP430X core running near 1.5 volts, which are often supplied by an NFC reader, such as an Android phone. With no need for a battery, the devices can be very small and thin, and it's not inconvenient to carry a complete device in your wallet.

The chip has three memories: SRAM, ROM, and FRAM.

Four kilobytes of SRAM at 0x1C00 are the RAM you've known and loved for years. SRAM is nice and fast with no requirements for being refreshed, but its contents will be lost when the power is cut. Surprisingly, most of this SRAM is unused because of its volatility, and it seems to exist mostly for development, where just over three kilobytes can be remapped over the ROM.

At 0x4400 we find seven kilobytes of masked ROM, which are hard coded into the chip by the manufacturer. While this code can't be changed in the field, customers who find themselves in need of hundreds of thousands of units can certainly make their own arrangements with TI to have chips with custom ROM contents produced. In the FRL152H, this ROM contains a complete NFC stack and a sensor data acquisition stack that reads samples into FRAM for long term storage.

As SRAM is too volatile and ROM is too permanent for storing the application firmware of our device, we find nearly two kilobytes of FRAM at 0xF840. FRAM, Ferroelectric RAM, is a strange competitor to old fashioned core memory that recently became viable for small devices. It does *not* require power to retain its contents, and writes are orders of magnitude cheaper than Flash memory, with no requirements for expensive page erasures. There is also some FRAM at 0x1A00, which stores the device's serial number and calibration settings. The Interrupt Vector Table is stored as addresses at the end of FRAM, ending with the RESET handler's address at 0xFFFE.

In addition to the three memories, there is an IO region which begins at the null address, 0x0000. There are no IO instructions in the MSP430 architecture, and IO is performed by movs to and from this region. For more background information on MSP430 exploitation and reverse engineering, see PoC||GTFO 2:5 and 11:8.

### Tooling

Now that we know a little about the chip, it's necessary to write software tools and to order some hardware. Trying to skip this step will only lead to heartache and confusion.

On the software end, we first need a way to talk to the chip. Modern phones have support for the NFC Type V protocols used in this chip, so I tossed together an Android app called GoodV to take care of reading, writing, programming, and erasing these chips.<sup>5</sup> In addition to the standard command set, it also supports backdoor commands unique to each chip and the ability to execute temporary fragments of shellcode from SRAM.

Because the RF430 uses an awkwardly low voltage, I ordered some RF430FRL152HEVM evaluation boards and a matching MSP-FET debugger from Texas Instruments. This allows me to completely wreck the chip's FRAM contents, then restore the chip to functionality through JTAG. It's also handy for interactive debugging, provided your breakpoints respect the timing requirements of the NFC protocol.

---

<sup>5</sup>git clone <https://github.com/travisgoodspeed/GoodV>

We also need firmware to run inside of the chips, both from FRAM as a permanent application image and from SRAM as temporary shellcode. For this, I used TI's branch of GCC8 for the MSP430. In past projects Debian's fork of GCC4 has been nicer for this platform, but upgrading to GCC8 was necessary to have the same calling convention in our code as the ROM. This project is called GoodTag, and it also includes a PCB design for the RF430 in Kicad.<sup>6</sup> (Schematic on page 9.)

## GoodV for Android

Before we begin to play with the parts, let's take a brief interruption to discuss how NFC tags work in Android and how to write a tool to communicate wirelessly with the RF430.

In Android, NFC Type V tags are accessed through the `android.nfc.tech.NfcV` class, whose `transceive()` function sends a byte array to the tag and returns the result. Because tags have such wildly varying properties as their command sets, block sizes and addressing modes, these raw commands are used rather than higher-level wrappers.

Commands are sent as first an option byte, which is usually `02`, and then a command byte and the optional command parameters. An explicit address can be stuck in the middle if indicated by the option bytes. Commands above `A0` require the manufacturer's number to follow, which for TI is `07`.

You can try out the low-level commands yourself in the NFC Tools app, whose Other/Advanced function accepts raw commands after a scary disclaimer. Just set the I/O Class to `NfcV` and then sent the following examples, before using them to implement our own high level functions for the chip.

We'll get into more commands later, but for now you should pay attention to the general format. Here, `20` is the standard command to read a block from an 8-bit block address and `C0` is the secret vendor command to read a block from a 16-bit block address. The first byte of each reply is zero for success, non-zero for failure.

```

1 02:20:00      — Reads block 00.
  00:E1:40:40:00 — Success, 4 bytes of data.
3
  02:C007:0000 — Reads block 0000
5 00:E1:40:40:00 — Success, same 4 bytes.
```

<sup>6</sup>`git clone https://github.com/travisgoodspeed/goodtag`

<sup>7</sup>See issue 86 on the Mspdebug github page if using that fine software. Uniflash is ugly and bloated, but it works with this chip out of the box.

This particular tag is configured to 4-byte blocks, and we might have gotten different results if configured to 8-byte blocks. The secret block `FF` contains these and other settings on the FRL152.

The `C0` read command and matching `C1` write command can read from a 16-bit block address, but they are still confined to a subset of FRAM and SRAM. To get the ROM, we'll go back to the hardware.

## RF430FRL152H

Once the parts have arrived, we can dump the FRL152's mask ROM through JTAG, and begin to reverse engineer it.<sup>7</sup> In the ROM, we aren't yet very interested in the taking of sensor measurements, but we would very much like to understand what commands are available and how they are implemented.

While IDA Pro, Radare2 or Binary Ninja would work fine for this, we chose GHIDRA for its decompiler and version control. In addition to the ROM, we also loaded dumps of SRAM and FRAM from an unused chip, so that there would be accurate function pointer tables and global variables.

After opening the firmware and carving out functions, we began by defining the `RF13MTXF` (`0x0808`) and `RF13MRXF` (`0x0806`) IO registers as volatiles. By searching for functions that access these registers, or for constants used in commands, we can quickly identify their implementations in the ROM.

```

; This handles a write to block 00FF, a
; region for just the Firmware System
; Control Register byte at 0xF867. When
; calling this over NfcV, you must send a
; password byte of 0x95 before the value you
; intend to write. See page 57 of SLAU603B.
rom_writesysctrlreg:
5d2c      CMP.B      #0x95,&RF13MRXF
          ; Is 0x95 read from the RF13 modem?
5d32      JNE          earlyret
5d34      MOV.B      &RF13MRXF,R12
5d38      CALL       #rom_writesysctrlreg
earlyret:
5d3c      RET
```

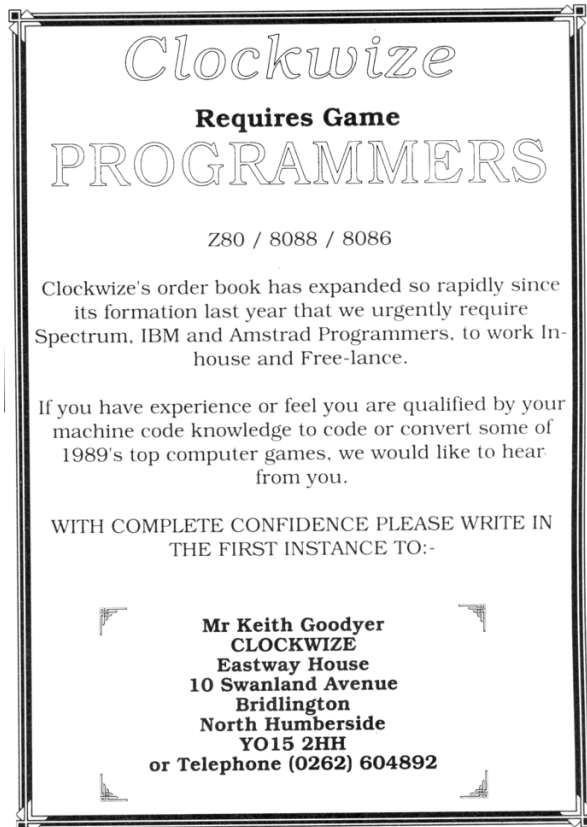


Soon enough we had a nice little understanding of how the ROM worked, and anything that was missing could easily be looked up. As we'll soon see, that was handy both for making our own firmware smaller and for injecting shellcode into SRAM to quickly perform complicated functions.

### Injecting Temporary Shellcode

So now that we understand the ROM, and we know that the C1 command can write to SRAM, we can have GoodV inject shellcode into the tag and execute it! Remote code execution is the name of the game.

From our memory dumps, it was clear that most of the little SRAM in use was used for a single table of function pointers, which is loaded from a master copy in ROM and then altered by patches which are loaded from FRAM. While in other cases we'll change that table permanently through modifying FRAM, for now we'd just like to be able to temporarily change it to run our shellcode once, with no permanent changes to the tag.



This was a better target than the call stack because it was a fixed target, and we could modify the pointer long before calling it. In the end, we chose the `rom_rf13_senderror()` function sends an error in response to an illegal block address. The Java code on page 11 calls a function at a given address by overwriting that pointer, triggering the error, and then restoring the original handler. It returns the NFC message returned by the error, which might be quite a few bytes.

Having the Java to run the shellcode is well and good, but we also need the shellcode itself. Rather than hand write it in assembly, we simply targeted the GNU linker to SRAM and also gave it a small region for parameters.

```

1  /* Parameters are loaded to 1E02 by the
3     linker. We take three 16-bit words as
4     little endian there for destination,
5     source, and length.
6
7     */
8     __attribute__((section(".params")))
9     uint16_t params[3];
10
11  /* This little bit of shellcode calls
12     memcpy() with the given parameters,
13     returning 0 on success, 1 on failure.
14
15     */
16  void __attribute__((noinline))
17     shellcode_main(){
18     //Return two bytes for continuation.
19     RF13MTXF= memcpy((void*) params[0],
20                     (void*) params[1], params[2]);
21     return;
22 }

```

This shellcode can then be expressed in a modified form of the TI-TXT file format, where the `x` keyword executes from the current working address. Simply change the six bytes at `0x1E02` to contain your destination, source, and length.

```

@1E02
00 00 00 00 00 00
@1E12
3C 40 02 1E 1E 4C 04 00 1D 4C 02 00 2C 4C B0 12
2A 1E 82 4C 08 08 30 41 0A 12 4B 43 0E 9B 03 20
4C 43 30 40 50 1E 0F 4C 0F 5B 6F 4F 1B 53 0A 4D
0A 5B 5A 4A FF FF 0F 9A F1 27 0C 4F 0C 8A 3A 41
30 41
@1E12
x
q

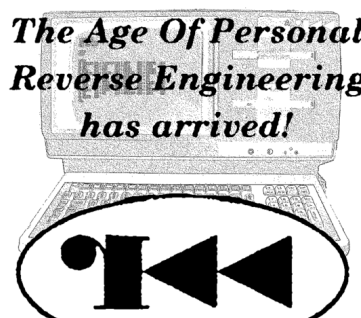
```

```

2  public byte[] exec(int adr) throws IOException {
3      /* While we could overwrite the call stack, it is much easier to overwrite the
4         function call table in early SRAM with a pointer to our function, because we
5         can only perform writes of 4 or 8 bytes at a time, and the call stack within a
6         write handler will be quite different from the one in a read handler.
7
8         There are plenty of functions to choose from, and an ideal hook would be one that
9         won't be missed by normal functions. We'd also prefer to have continuation wherever
10        possible, so that executing the code doesn't crash our target.
11
12        The function pointer we'll overwrite is at 0x1C5C, pointing to rom_rf13_senderror()
13        at 0x4FF6. For proper continuation, you can just write two bytes to RF13MTXF and
14        return. Without proper continuation, an IOException will be thrown in the reply
15        timeout. To unhook, write 0x4FF6 to 0x1C5C, restoring the original handler.
16
17        As a handy side effect, we return the two bytes that need to be transmitted for
18        continuation, so you can get a bit of data back from your shellcode.
19    */
20    Log.v("GoodV", String.format("Asked to call shellcode at %04x", adr));
21
22    // First we replace the read error reply handler.
23    write(0x1C5C, new byte[] {(byte) (adr & 0xFF), (byte) (adr >> 8)});
24
25    // Then we read from an illegal address to trigger an error,
26    // returning the two bytes of its handler.
27    byte[] shellcodereturn = transceive(new byte[] {
28        0x02, // Flags
29        (byte) 0xC0, // MFG Raw Read Command
30        0x07, // MFG Code
31        (byte) (0xbe), (byte) (0xba) //16-bit block number, little endian.
32    });
33    Log.v("GoodV", "Shellcode returned: " + GoodVUtil.byteArrayToHex(shellcodereturn));
34
35    //And finally, we repair the original handler address, like nothing ever happened.
36    write(0x1C5C, new byte[] {(byte) (0xf6), (byte) (0x4f)});
37
38    return shellcodereturn;
39 }

```

## Java Function to Execute RF430 Shellcode from Android




**The Age Of Personal  
Reverse Engineering  
has arrived!**

Solved: That when tongues turn white, breath feverish, stomach sour and bowels constipated, that our mothers give us tiny portions of love and sugar, we claim pills and shells in exotic architectures in order to port the thing everywhere.

No need to wait more for this to happen! The era of personal reverse engineering has finally arrived. No taxes or country restrictions involved! Free radare2 licenses is a commodity that everybody can enjoy

With radare2 you can disassemble, analyze, debug, patch any binary for a wide range of CPUs and OSs even for your shiny 4004 running PC/M!





## RF430TAL152H

We'll get back to programming the RF430FRL152H in a bit, but now that we can reverse engineer, program, and exploit that chip, let's take a look at its commercial variant, the RF430TAL152H.

The TAL152 is very similar in layout and appearance to the FRL152, with the principle difference being the contents of mask ROM and the JTAG configuration. It can be found in a popular brand of continuous glucose monitor,<sup>8</sup> and there is precious little to be found about the chip online, with no public datasheet and all conversation shut down in TI's E2E forums.

In this section, we'll trace the long road from first examining this chip to finally dumping its ROM and then writing custom firmware to FRAM.

### Reading, but not Writing, to FRAM

When first experimenting with the chip, we find that there is one extra block of FRAM exposed by NFC, and that there is no secret page of the configuration at page FF. Every last page is write protected, and we cannot change any of them with the standard write command, 21.

But all is not lost! There is a table of function pointers on the final page, and the value of the RESET vector tells us that this ROM is different from the FRL152, so we know that the two devices have different software in their ROMs.

We also see this table, which begins at 0xFFCE with the magic word 0xABAB and then grows downward to the same word at a lower address, 0xFFB8.<sup>9</sup> Each entry in this table is a custom vendor command, and we see that much like the C0 and C1 commands that have been so handy on the FRL152, the TAL152 has commands A0, A1, A2, A3, and A4.

<sup>8</sup>See our lecture, *The Inner Guts of a Connected Glucose Sensor for Diabetes* at Black Alps 2019 for details of the sensor in a medical context.

<sup>9</sup>The location and format are the same as the FRL152, except that the magic word is ABAB instead of CECE.

We also see that A1 and A3 are in FRAM, where we can read at least part of their code.

1	ffac	ab ab	dw	ABABh
	ffae	4a fb	addr	fram_e2
3	ffb0	e2 00	dw	E2h
	ffb2	3c fa	addr	fram_e1
5	ffb4	e1 00	dw	E1h
	ffb6	ae fb	addr	fram_e0
7	ffb8	ab ab	dw	ABABh
	ffba	2c 5a	addr	rom_a4
9	ffbc	a4 00	dw	A4h
	ffbe	ca fb	addr	fram_a3
11	ffc0	a3 00	dw	A3h
	ffc2	56 5a	addr	rom_a2
13	ffc4	a2 00	dw	A2h
	ffc6	ba f9	addr	fram_a1
15	ffc8	a1 00	dw	A1h
	ffca	24 57	addr	rom_a0
17	ffcc	a0 00	undefined2	00A0h
	ffce	ab ab	dw	ABABh

The table ends early, of course, with E0, E1, and E2 being disabled by E0's command number having been overwritten by the table end marker. These commands were available at some point in the manufacturing process, and we can read their command handlers from FRAM, but we cannot execute them.

Calling these functions is a bit disappointing. A1 returns the device status of some sort, but the other Ax commands don't even grace us with an error message in reply. The reason for this is hard to see from the partial assembly, but we later learned that they require a safety password.

So not yet being able to run A3, we read its disassembly. The function begins by calling another function at 0x1C20 and then proceeds to read a raw address and length before sending the requested number of 16-bit words out the RF13 modem to the reader. If we could just call this command, we could dump the ROM and reverse engineer the behavior of the other commands!

### Sniffing the Readers

To get the password, we had to sniff a legitimate reader's attempts to call any Ax command other than A1, so that we could learn the password and use A3 to dump raw memory. We found this both by tapping the SPI bus of the manufacturer's dedicated hardware reader and separately by observing the vendor's Android app in Frida.

The 32-bit password came as a parameter to the A0 command, which initializes the glucose sensor after injection into a patient's arm. Trying this same password in A3, followed by an address and length, gave us the ability to read raw memory. Looping this gave complete dumps of ROM and SRAM, as well as a complete dump of the FRAM regions which are not exposed by the standard read command, 20.

### Inside the ROM

Loading this complete dump into GHIDRA shows that the ROM is related to that of the FRL152H, but that they have diverged quite a bit. The TAL152 implements no vendor commands directly; rather, they must be added through the patch table. It has no secret pages.

Lacking the ability to write directly to pages, and finding no new commands, we explored the remaining commands. Sure enough, A2 write protects every FRAM page that is exposed by NFC, and A4 unlocks almost all of those same pages!

### Unlocking and Patching

Calling the A4 command, we can then unlock pages and begin mucking around. A simple write to 0xFFB8 will re-enable the Ex commands, allowing us to experiment with restoring old sensors. Or we can compile our own firmware to run inside of the TAL152, turning a glucose sensor into some other device.

### Some Other Unlocking Techniques

While trying to dump the TAL152, we hit a few dead ends that might possible work for you on other targets.

First, the JTAG of the TAL152 appears to be unlocked if it follows the same convention as the FRL152. This might very well be caused by a custom activation key,<sup>10</sup> but whether it is a different locking mechanism or a different key, we were unable to get a connection.

We also tried to wipe these chips back to a factory setting by raising them above their Curie point, which Texas Instruments Application Report SLAA526A, *MSP430 FRAM Quality and Reliability*, leads us to believe is near 430°C. Short experiments involving a hot air gun and strong magnets

were unsuccessful, but by summer I hope to mill a metal case for the RF430 then back a chip in a regulated kiln for many hours to look for bit failures. Custom firmware might also allow visibility into the error correcting bits of the FRAM, to better recognize partial success at introducing errors.

There are also some test pins on the chip which aroused our curiosity, as other chips use them to enter a bootloader and these chips might use them to reset to a factory state. This could be as effective as overheating the FRAM, without the hassles of extreme temperatures.

It's also worth noting that our successful method—using the A3 command with the manufacturer's password—could be accomplished *either* by tapping the hardware reader's SPI bus *or* by reading that same password out of the manufacturer's Android application. In reverse engineering, any technique that works is a good one, and there's often more than one way to win the game.

**New for your TV!** computerized **PING PONG**

Assemble your own electronic Ping-Pong unit that connects to any TV. It's easy! Complete plans, p/c boards, preassembled & finished units. Our designs include challenging game action, a computer-control paddle sound effects & on-screen scoring. Exciting!

Build the basic unit for about \$40 in common components.

Send \$27.50 for "Superset" p/c board (with aligned horiz. & vert. oscillators) & plans . . . or . . . send \$1.00 (refundable) for circuit diagram & info packet of p/c boards, plans, accessories & completed units.

**visulex**  
P.O. BOX 4204B  
MOUNTAIN VIEW, CA 94040

\$1 for schematic diagram & info pack (refundable on purchase).

<sup>10</sup>See issue 86 on the Mspdebug project for details on the activation key. <https://github.com/dlbeer/mspdebug/issues/86>