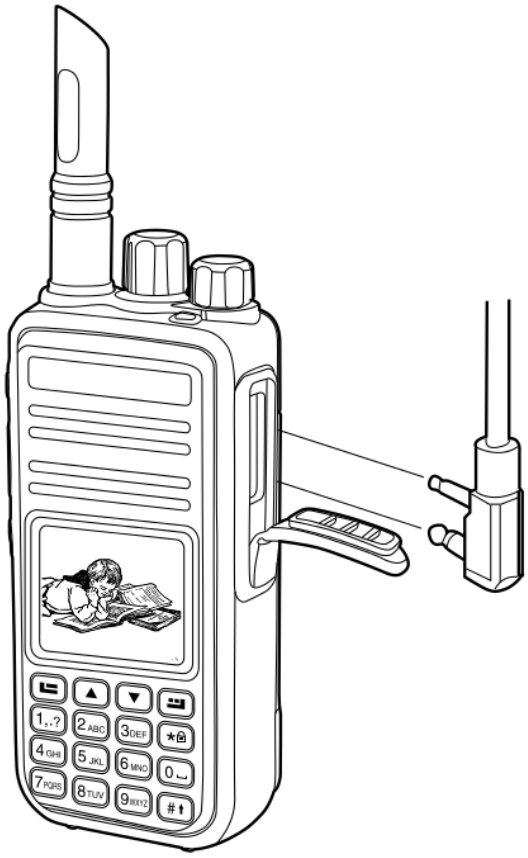
8 Reverse Engineering the Tytera MD380

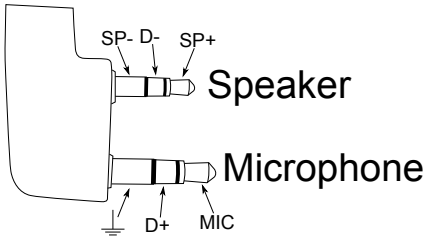
by Travis Goodspeed KK4VCZ,
with kind thanks to DD4CR and W7PCH.



The following is an adventure of reverse engineering the Tytera MD380, a digital hand-held radio that can be had for barely more than a hundred bucks. In this article, I explain how to read and write the radio's configuration over USB, and how to break the readout protection on its firmware, so that you fine readers can write your own strange and clever software for this nifty gizmo. I also present patches to promiscuously receive audio from unknown talkgroups, creating the first hardware scanner for DMR. Far more importantly, these notes will be handy when you attempt to reverse engineer something similar on your own.

This article does not go into the security problems of the DMR protocol, but those are sufficiently similar to P25 that I'll just refer you to *Why (Special Agent) Johnny (Still) Can't Encrypt* by Sandy Clark and Friends.[59]

8.1 Hardware Overview



The MD380 is a hand-held digital voice radio that uses either analog FM or Digital Mobile Radio (DMR). It is very similar to other DMR radios, such as the CS700 and CS750 from Connect Systems.[60]

DMR is a trunked radio protocol using two-slot TDMA, so a single repeater tower can be used by one user in Slot 1 while another user is having a completely different conversation on Slot 2. Just like GSM, the tower coordinates which radio should transmit when.

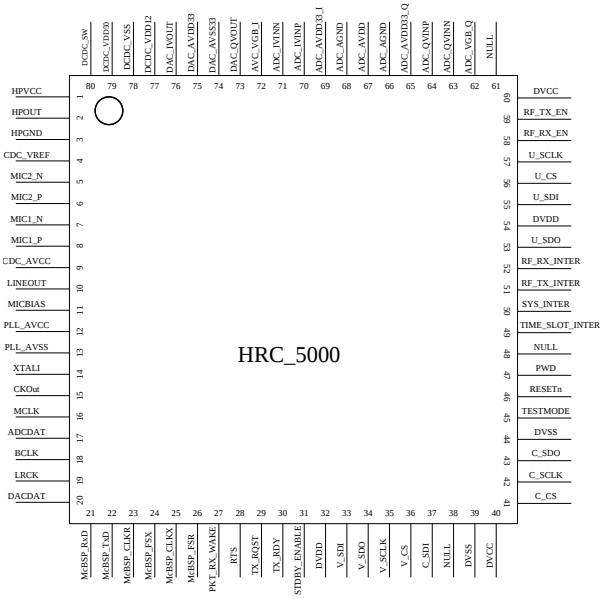
The CPU of this radio is an STM32F405 from STMicroelectronics. This contains a Cortex M4, so all instructions are Thumb and all function pointers are odd. The LQFP100 package of this chip is used. It has a megabyte of Flash and 192 kilobytes of RAM. The STM32 has both JTAG and a ROM bootloader, but both of these are protected by a Readout Device Protection (RDP) feature. In Section 8.8, I'll show you how to bypass these protections and jailbreak your radio.

There is also a radio baseband chip, the HR C5000. At first I was reconstructing the pinout of this chip from the CS700 Service Manual, but the full documentation can be had from DocIn, a Chinese PDF sharing website. 中国排名第一。

Aside from a bunch of support components that we can take for granted, there is an SPI Flash chip for storing the codeplug. "Codeplug" is a Motorola term for the radio settings, such as frequencies, contacts, and talk groups; I use the term here to distinguish the radio configuration in SPI Flash from the

[59] `unzip pocorgtfo10.pdf p25sec.pdf #from Proceedings of the 20th Usenix Security Symposium in 2011`
[60] The folks at Connect Systems are nice and neighborly, so please buy a radio from them.

code and data in CPU Flash.



8.2 A Partial Dump

From `lsusb -v` on Linux, we can see that the device implements USB DFU, most likely as a fork of some STMicro example code. The MD380 appears as an STMicro DFU device with storage for Internal Flash and SPI Flash with a VID:PID of `0483:df11`.

```
iMac% dfu-util -list
Found DFU: [0483:df11]
    devnum=0, cfg=1, intf=0, alt=0,
    name="@Internal Flash
          /0x08000000/03*016Kg"
Found DFU: [0483:df11]
    devnum=0, cfg=1, intf=0, alt=1,
    name="@SPI Flash Memory
          /0x00000000/16*064Kg"
```

Further, the `.rdt` codeplug files are SPI Flash images in the DMU format, which is pretty much just wrapper with a bare minimum of metadata around a flat, uncompressed memory image. These codeplug files contain the radio's contact list, repeater frequencies, and other configuration info. We'll get back to this later, as what we really want to do is dump and patch the firmware.

Unfortunately, dumping memory from the device by the standard DFU protocol doesn't seem to yield useful results, just the same repeating binary string, regardless of the alternate we choose or the starting position.

```
iMac% dfu-util -d 0483:df11 --alt 1 -s 0:0x200000 -U
      first1k.bin
Filter on vendor = 0x0483 product = 0xdf11
Opening DFU capable USB device... ID 0483:df11
Run-time device DFU version 011a
Found DFU: [0483:df11] devnum=0, cfg=1, intf=0, alt=1,
      name="@SPI Flash Memory /0x00000000/16*064Kg"
Claiming USB DFU Interface...
Setting Alternate Setting #1 ...
Determining device status: state = dfuUPLOAD-IDLE
aborting previous incomplete transfer
Determining device status: state = dfuIDLE, status = 0
dfuIDLE, continuing
DFU mode device DFU version 011a
Device returned transfer size 1024
Limiting default upload to 2097152 bytes
bytes_per_hash=1024
Starting upload: [####...####] finished!
iMac% hexdump first1k.bin
0000000 30 1a 00 20 15 56 00 08 29 54 00 08 2b 54 00 08
0000010 2d 54 00 08 2f 54 00 08 31 54 00 08 00 00 00 00
0000020 00 00 00 00 00 00 00 00 00 00 00 00 33 54 00 08
0000030 35 54 00 08 00 00 00 00 83 30 00 08 37 54 00 08
0000040 61 56 00 08 65 56 00 08 69 56 00 08 5b 54 00 08
...
00003c0 10 eb 01 60 df f8 34 1a 08 60 df f8 1c 0c 00 78
00003d0 40 28 c0 f0 e6 81 df f8 24 0a 00 68 00 f0 0e ff
00003e0 df e1 df f8 10 1a 09 78 a2 29 0f d1 df f8 f8 19
00003f0 09 68 02 29 0a d1 df f8 00 0a 02 21 01 70 df f8
... [same 1024 bytes repeated]
```

In this brave new world, where folks break their bytes on the little side by order of Golbasto Momarem Evlame Gurdilo Shefin Mully Ully Gue, Tyrant of Lilliput and Eternal Enemy of Big Endians and Blefuscu, to break them on the little side, it's handy to spot four byte sequences that could be interrupt handlers. In this case, what we're looking at is the first few pointers of an interrupt vector table. This means that we are grabbing memory from the beginning of internal flash at `0x08000000`!

Note that the data repeats every kilobyte, and also that `dfu-util` is reporting a transfer size of 1,024 bytes. The `-t` switch will order `dfu-util` to dump more than a kilobyte per transfer, but everything after the first transfer remains corrupted.

This is because `dfu-util` isn't sending the proper commands to the radio firmware, and it's getting the page as a bug rather than through proper use of the protocol. (There are lots of weird variants of DFU, created by folks only using DFU with their own tools and never testing for compatibility with each other. This variant is particularly weird, but manageable.)

8.3 Tapping USB with VMWare

Before going further, it was necessary to learn the radio's custom dialect of DFU. Since my Total Phase USB sniffers weren't nearby, I used VMWare to sniff the transactions of both the MD380's firmware updater and codeplug configuration tools.

I did this by changing a few lines of my VMWare `.vmx` configuration to dump USB transactions out

to `vmware.log`, which I parsed with ugly regexes in Python. These are the additions to the `.vmx` file.

```
monitor = "debug"
usb.analyzer.enable = TRUE
usb.analyzer.maxLine = 8192
mouse.vusb.enable = FALSE
```

The logs showed that the MD380's variant of DFU included non-standard commands. In particular, the LCD screen would say "PC Program USB Mode" for the official client applications, but not for any 3rd party application. Before I could do a proper read, I had to find the commands that would enter this programming mode.

DFU normally hides extra commands in the `UPLOAD` and `DNLOAD` commands when the block address is less than two. (Hiding them in blocks `0xFFFF` and `0xFFFE` would make more sense, but if wishes were horses, then beggars would ride.)

To erase a block, a DFU host sends `0x41` followed by a little endian address. To set the address pointer (block 2's address), the host sends `0x21` followed by a little endian address.

In addition to those standard commands, the MD380 also uses a number of two-byte (rather than five-byte) `DNLOAD` transactions, none of which exist in the standard DMU protocol. I observed the following, which I still only partially understand.

Non-Standard DNLOAD Extensions

`91 01`	Enables programming mode on LCD.
`a2 01`	Seems to return model number.
`a2 02`	Sent only by config read.
`a2 31`	Sent only by firmware update.
`a2 03`	Sent by both.
`a2 04`	Sent only by config read.
`a2 07`	Sent by both.
`91 31`	Sent only by firmware update.
`91 05`	Reboots, exiting programming mode.

8.4 Custom Codeplug Client

Once I knew the extra commands, I built a custom DFU client that would send them to read and write codeplug memory. With a little luck, this might have given me control of firmware, but as you'll see, it only got me half way.

Because I'm familiar with the code from a prior target, I forked the DFU client from an old version of Michael Ossmann's Ubertooth project.[61]

Sure enough, changing the VID and PID of the `ubertooth-dfu` script was enough to start dumping memory, but just like `dfu-util`, the result was a repeating sequence of the first block's contents. Because the block size was 256 bytes, I received only the first `0x100` bytes repeated.

Adding support for the non-standard commands in the same order as the official software, I got a copy of the complete 256K codeplug from SPI Flash instead of the beginning of Internal Flash. Hooray!

To upload a codeplug back into the radio, I modified the `download()` function to enable programming mode and properly wait for the state to return to `dfuDNLOAD_IDLE` before sending each block.

This was enough to write my own codeplug from one radio into a second, but it had a nasty little bug! I forgot to erase the codeplug memory, so the radio got a bitwise AND of two valid codeplugs.[62]

A second trip with the USB sniffer shows that these four blocks were erased, and that the upload address must be set to zero *after* the erasure.
`0x00000000 0x00010000 0x00020000 0x00030000`

Erasing the blocks properly gave me a tool that correctly reads and writes the radio codeplug!

8.5 Codeplug Format

Now that I could read and write the codeplug memory of my MD380, I wanted to be able to edit it. Parts of the codeplug are nice and easy to reverse, with strings as UTF16L and numbers being either integers or BCD. Checksums don't seem to matter, and I've not yet been able to brick my radios by uploading damaged firmware images.

The Radio Name is stored as a string at `0x20b0`, while the Radio ID Number is an integer at `0x2080`. The intro screen's text is stored as two strings at `0x2040` and `0x2054`.

```
#seekto 0x5F80;
struct {
  ul24 callid;    //DMR Account Number
  u8   flags;     //c2 private, no tone
                  //e1 group, with rx tone
  char name[32]; //U16L chars
} contacts[1000];
```

[61] In particular, I used r543 of the old SVN repository, a version from 4 July 2012.
[62] See PoC‖GTFO 2:5.
[63] `http://chirp.danplanet.com`

CHIRP,[63] a ham radio application for editing radio codeplugs, has a bitwise library that expects memory formats to be defined as C structs with base addresses. By loading a bunch of contacts into my radio and looking at the resulting structure, it was easy to rewrite it for CHIRP.

Repeatedly changing the codeplug with the manufacturer's application, then comparing the hexdumps gave me most of the radio's important features. Patience and a few more rounds will give me the rest of them, and then my CHIRP plugin can be cleaned up for inclusion.

Unfortunately, not everything of importance exists within the codeplug. It would be nice to export the call log or the text messages, but such commands don't exist and the messages themselves are nowhere to be found inside of the codeplug. For that, we'll need to break into the firmware.

8.6 Dumping the Bootloader

Now that I had a working codeplug tool, I'd like a cleartext dump of firmware. Recall from Section 8.2 that forgetting to send the custom command `0x91 0x01` leaves the radio in a state where the beginning of code memory is returned for every read. This is an interrupt table!

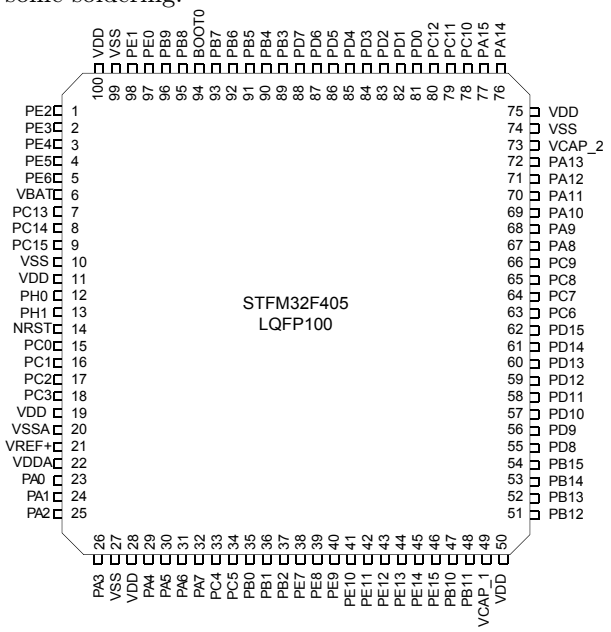
MD380 Recovery Bootloader Interrupts

`0x20001a30`	Top of the call stack.
`0x08005615`	Reset Handler
`0x08005429`	Non-Maskable Interrupt (NMI)
`0x0800542b`	Hard Fault
`0x0800542d`	MMU Fault
`0x0800542f`	Bus Fault
`0x08005431`	Usage Fault

From this table and the STM32F405 datasheet, we know the code flash begins at `0x08000000` and RAM begins at `0x20000000`. Because the firmware updater only writes to regions at and after `0x0800-C000`, we can guess that the first 48k are a recovery bootloader, with the region after that holding the application firmware. As all of the interrupts are odd, and because the radio uses a Cortex M4 core, we know that the firmware is composed exclusively of Thumb (and Thumb2) code, with no old fashioned ARM instructions.

Sure enough, I was able to dump the whole bootloader by reading a single page of `0xC000` bytes from the application mode. This bootloader is the one used for firmware updates, which can be started by holding PTT and the unlabeled button above it when turning on the power switch.[64]

This trick doesn't expose enough memory to dump the application, but it was valuable to me for two very important reasons. First, this bootloader gave me some proper code to begin reverse engineering, instead of just external behavioral observations. Second, the recovery bootloader contains the keys and code needed to decrypt an application image, but to get at that decrypted image, I first had to do some soldering.



8.7 Radio Disassembly (BOOT0 Pin)

As I stress elsewhere, the MD380 has *three* applications in it: (1) Tytera's Radio Application, (2) Tytera's Recovery Bootloader, and (3) STMicro's Bootloader ROM. The default boot process is for the Recovery Bootloader to immediately start the Radio Application unless Push-To-Talk (PTT) and the button above it are held during boot, in which case it waits to accept a firmware update. There is no key sequence to start the STMicro Bootloader ROM, so a bit of disassembly and soldering is required.

This ROM contains commands to read and write all of memory, as well as to begin execution at any arbitrary address. These commands are initially locked down, but in Section 8.8, I'll show how to get around the restrictions.

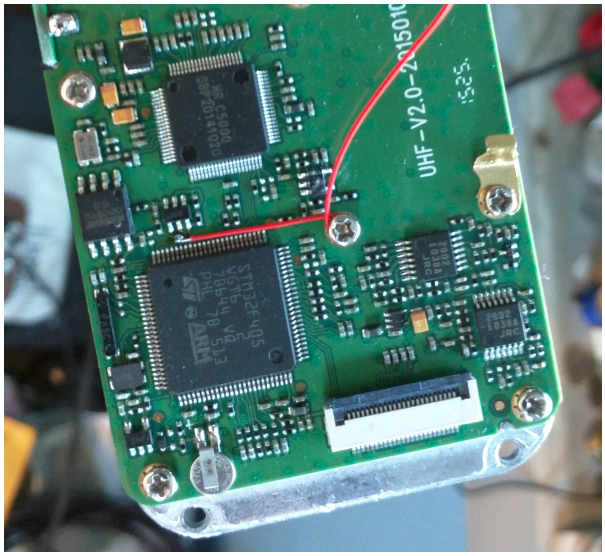
[64] Transfers this large work on Mac but not Linux.

To open your radio, first remove the battery and the four Torx screws that are visible from the back of the device. Then unscrew the antenna and carefully pry off the two knob covers. Beneath each knob and the antenna, there are rings that screw in place to secure them against the radio case; these should be moved by turning them counter-clockwise using a pair of sturdy, dull tweezers.

Once the rings have been removed, the radio's main board can be levered up at the bottom of the radio, then pulled out. Be careful when removing it, as it is attached with a Zero Insertion Force (ZIF) connector to the LCD/Keypad board, as well as by a short connector to the speaker.

The STMicro Bootloader is started by pulling the BOOT0 pin of the STM32F405 high while restarting the radio. I did this by soldering a thin wire to the test pad near that pin, wrapping the wire around a screw for strain relief, then carefully feeding it out through the microphone/speaker port.

(An alternate method involves removing BOOT0's pull-down resistor, then fly-wiring it to the pull-up on the PTT button. Thanks to tricky power management, this causes the radio to boot normally, but to *reboot* into the Mask ROM.)



8.8 Bootloader RE

Once I finally had a dump of Tytera's bootloader, it was time to reverse engineer it.[65]

The image is 48K in size and should be loaded to `0x08000000`. Additionally, I placed 192K of RAM at `0x20000000`. It's also handy to create regions for the I/O banks of the chip, in order to help track those accesses. (IDA and Radare2 will think that peripherals are global variables near `0x40000000`.)

After wasting a few days exploring the command set, I had a decent, if imperfect, understanding of the Tytera Bootloader but did not yet have a clear-text copy of the application image. Getting a bit impatient, I decided to patch the bootloader to keep the device unprotected while loading the application image using the official tools.

I had to first explore the STM32 Standard Peripheral Library to find the registers responsible for locking the chip, then hunt for matching code.

```
/* STM32F4xx flash regs from stm32f4xx.h */
#@0x40023c00
typedef struct {
  __IO uint32_t ACR;      //access ctrl    0x00
  __IO uint32_t KEYR;     //key            0x04
  __IO uint32_t OPTKEYR;  //option key     0x08
  __IO uint32_t SR;       //status         0x0C
  __IO uint32_t CR;       //control        0x10
  __IO uint32_t OPTCR;    //option ctrl    0x14
  __IO uint32_t OPTCR1;   //option ctrl 1  0x18
} FLASH;
```

[65] The MD5 of my image is `721df1f98425b66954da8be58c7e5d55`, but you might have a different one in your radio.

The way flash protection works is that byte 1 of `FLASH->OPTCR` (at `0x40023C15`) is set to the protection level. `0xAA` is the unprotected state, while `0xCC` is the permanent lock. Anything else, such as `0x55`, is a sort of temporary lock that allows the application to be wiped away by the Mask ROM bootloader, but does not allow the application to be read out.

Tytera is using this semi-protected mode, so you can pull the `BOOT0` pin of the STM32F4xx chip high to enter the Mask ROM bootloader.[66] This process is described in Section 8.7.

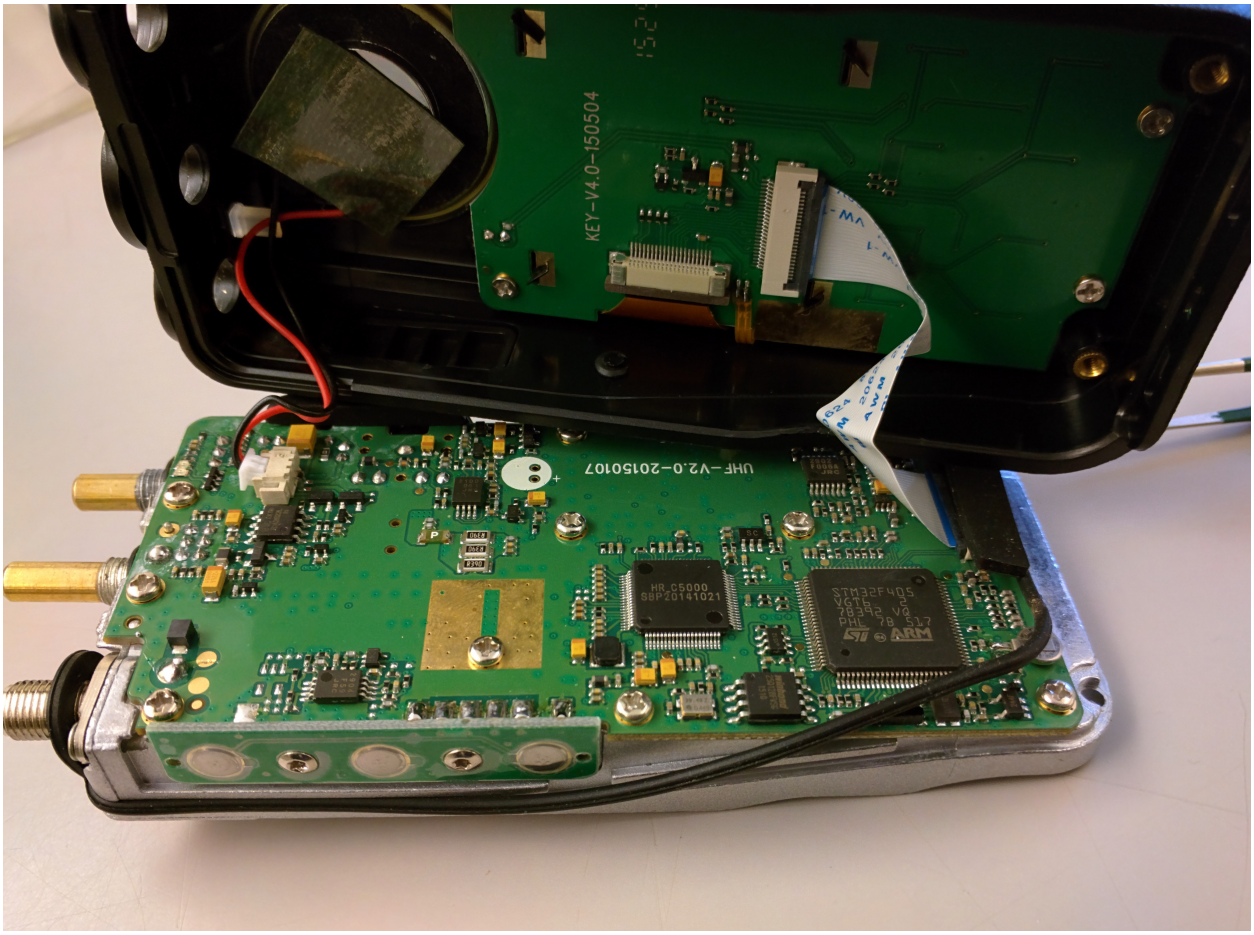
Sure enough, at `0x08001FB0`, I found a function that's very much like the example `FLASH_OB_RDP-Config` function from `stm32f4xx_flash.c`. I call the local variant `rdp_lock()`.

```
/* Sets the read protection level.
 * OB_RDP specifies the protection level.
 *          AA: No protection.
 *          55: Read protection memory.
 *          CC: Full chip protection.
 * WARNING: When enabling OB_RDP level 2
 *          it's no longer possible to go
 *          back to level 1 or 0.
 */
void FLASH_OB_RDPConfig(uint8_t OB_RDP){
  FLASH_Status status = FLASH_COMPLETE;

  /* Check the parameters */
  assert_param(IS_OB_RDP(OB_RDP));

  status = FLASH_WaitForLastOperation();
  if(status == FLASH_COMPLETE)
    *(__IO uint8_t*)
      OPTCR_BYTE1_ADDRESS = OB_RDP;
}
```

[66] Confusingly enough, this is the *third* implementation of DFU for this project! The radio application, the recovery bootloader, and the ROM bootloader all implement different variants of DFU. Take care not to confuse the them.



This function is called from `main()` with a parameter of `0x55` in the instruction at `0x080044A8`.

```
    0x080044a0     fdf7a0fd        bl rdp_isnotlocked
    0x080044a4     0028            cmp r0, 0
,=< 0x080044a6     04d1            bne 0x80044b2
|   ; Change this immediate from 0x55 to 0xAA
|   ; to jailbreak the bootloader.
|   0x080044a8     5520            movs r0, 0x55
|   0x080044aa     fdf781fd        bl rdp_lock
|   0x080044ae     fdf78bfd        bl rdp_applylock
`-> 0x080044b2     fdf776fd        bl 0x8001fa2
    0x080044b6     00f097fa        bl bootloader_pin_test
```

Patching that instruction to instead send `0xAA` as a parameter prevents the bootloader from locking the device. (We're just swapping `aa 20` in where `55 20` used to be.)

```
iMac% diff old.txt jailbreak.txt
< 00044a0 fd f7 a0 fd 00 28 04 d1
          55 20 fd f7 81 fd fd f7
---
> 00044a0 fd f7 a0 fd 00 28 04 d1
          aa 20 fd f7 81 fd fd f7
```

8.9 Dumping the Application

Once I had a jailbroken version of the recovery bootloader, I flashed it to a development board and installed an encrypted MD380 firmware update using the official Windows tool. Sure enough, the application installed successfully!

After the update was installed, I rebooted the board into its ROM by holding the `BOOT0` pin high. Since the recovery bootloader has been patched to leave the chip unlocked, I was free to dump all of Flash to a file for reverse engineering and patching.

8.10 Reversing the Application

Reverse engineering the application isn't terribly difficult, provided a few tricks are employed. In this section, I'll share a few; note that all pointers in this section are specific to Version 2.032, but similar functionality exists in newer firmware revisions.

At the beginning, the image appears almost entirely without symbols. Not one function or system call comes with a name, but it's easy to identify a few strings and I/O ports. Starting from those, related functions—those in the same `.C` source file—are often located next to one another in memory, providing hints as to their meaning.

[67] `unzip pocorgtfo10.pdf hrc5000.pdf`

The operating system for the application is an ARM port of MicroC/OS-II, an embedded real-time operating system that's quite well documented in the book of the same name by Jean J. Labrosse. A large function at `0x0804429C` that calls the operating system's `OSTaskCreateExt` function to make a baker's dozen of threads. Each of these conveniently has a name, conveniently describing the system interrupt, the real-time clock timer, the RF PLL, and other useful functions.

As I had already reverse engineered most of the SPI Flash codeplug, it was handy to work backward from codeplug addresses to identify function behavior. I did this by identifying `spiflash_read` at `0x0802fd82` and `spiflash_write` at `0x0802fbea`, then tracing all calls to these functions. Once these have been identified, finding codeplug functions is easy. Knowing that the top line of startup text is 32 bytes stored at `0x2040` in the codeplug, finding the code that prints the text is as simple as looking for calls to `spiflash_read(&foo, 0x2040, 20)`.

Thanks to the firmware author's stubborn insistence on 1-indexing, many of the structures in the codeplug are indexed by an address just before the real one. For example, the list of radio channel settings is an array that begins at `0x1ee00`, but the functions that access this array have code along the lines of `spiflash_read(&foo, 64*index+0x1edc0, 64)`.

One mystery that struck me when reverse engineering the codeplug was that I didn't find a missed call list or any sent or received text messages. Sure enough, the firmware shows that text messages are stored after the end of the 256K image that the radio exposes to the world.

Code that accesses the C5000 baseband chip can be reverse engineered in a similar fashion to the codeplug. The chip's datasheet[67] is very well handled by Google Translate, and plenty of dandy functions can be identified by writes to C5000 registers of similar functions.

Be careful to note that the C5000 has multiple memories on its primary SPI bus; if you're not careful, you'll confuse the registers, internal RAM, and the Vocoder buffers. Also note that a lot of registers are missing from the datasheet; please get in touch with me if you happen to know what they do.

Finally, it is crucially important to be able to sort through the DMR packet parsing and construction routines quickly. For this, I've found it handy

to keep paper printouts of the DMR standard, which are freely available from ETSI.[68] Link-Local addresses (LLIDs) are 24 bits wide in DMR, and you can often locate them by searching for code that masks against `0xFFFFFF`.[69]

8.11 Patching for Promiscuity

While it's fun to reverse engineer code, it's all a bit pointless until we write a nifty patch. Complex patches can be introduced by hooking function calls, but let's start with some useful patches that only require changing a couple of bits. Let's enable promiscuous receive mode, so the MD380 can receive from all talk groups on a known repeater and timeslot.

In DMR, audio is sent to either a Public Talkgroup or a Private Contact. These each have a 24-bit LLID, and they are distinguished by a bit flag elsewhere in the packet. For a concrete example, 3172 is used for the Northeast Regional amateur talkgroup, while 444 is used for the Bronx TRBO talkgroup. If an unmodified MD380 is programmed for just 3172, it won't decode audio addressed to 444.

There is a function at `0x0803ec86` that takes a DMR audio header as its first parameter and decides whether to play the audio or mute it as addressed to another group or user. I found it by looking for access to the user's local address, which is held in RAM at `0x2001c65c`, and the list of LLIDs for incoming listen addresses, stored at `0x2001c44c`.

To enable promiscuous reception to unknown talkgroups, the following talkgroup search routine can be patched to always match on the first element of `listengroup[]`. This is accomplished by changing the instruction at `0x0803ee36` from `0xd1ef` (`JNE`) to `0x46c0` (`NOP`).

```
for ( i = 0; i < 0x20u; ++i ){
  if ( (listengroup[i] & 0xFFFFFF)
       == dst_llid_adr ) {
    something = 16;
    recognized_llid_dst = dst_llid_adr;
    current_llid_group = var_lgroup[i+16];
    sub_803EF6C();
    dmr_squelch_thing = 9;
    if ( *(v4 + 4) & 0x80 )
      byte_2001D0C0 |= 4u;
    break;
  }
}
```

A similar `JNE` instruction at `0x0803ef10` can be replaced with a NOP to enable promiscuous reception of private calls. Care in real-world patches should be taken to reduce side effects, such as by forcing a match only when there's no correct match, or by skipping the missed-call logic when promiscuously receiving private calls.

8.12 DMR Scanning

After testing to ensure that my patches worked, I used Radio Reference to find a few local DMR stations and write them into a codeplug for my modified MD380. Soon enough, I was hearing the best gossip from a university's radio dispatch.[70]

Later, I managed to find a DMR network that used the private calling feature. Sure enough, my radio would ring as if I were the one being called, and my missed call list quickly grew beyond my two local friends with DMR radios.

8.13 A New Bootloader

Unfortunately, the MD380's application consumes all but the first 48K of Flash, and that 48K is consumed by the recovery bootloader. Since we neighbors have jailbroken radios with a ROM bootloader accessible, we might as well wipe the Tytera bootloader and replace it with something completely new, while keeping the application intact.

Luckily, the fine folks at Tytera have made this easy for us! The application has its own interrupt table at `0x0800C000`, and the RESET handler—whose address is stored at `0x0800C004`—automatically moved the interrupt table, cleans up the stack, and performs other necessary chores.

```
1  //Minimalist bootloader.
   void main(){
3    //Function pointer to the application.
     void (*appmain)();
5    //The handler address is the stored in the
     //interrupt table.
7    uint32_t *resethandler =
           (uint32_t*) 0x0800C004;
9    //Set the function pointer to that value.
     appmain = (void (*)()) *resethandler;
11   //Away we go!
     appmain();
13 }
```

[68] ETSI TS 102 361, Parts 1 to 4.

[69] In assembly, this looks like `LSLS r0, r0, #8; LSRS r0, r0, #8`.

[70] Two days of scanning presented nothing more interesting than a damaged elevator and an undergrad too drunk to remember his dorm room keys. Almost gives me some sympathy for those poor bastards who have to listen to wiretaps.

8.14 Firmware Distribution

Since this article was written, DD4CR has managed to free up 200K of the application by gutting the Chinese font. She also broke the (terrible) update encryption scheme, so patched or rewritten firmware can be packaged to work with the official updater tools from the manufacturer.

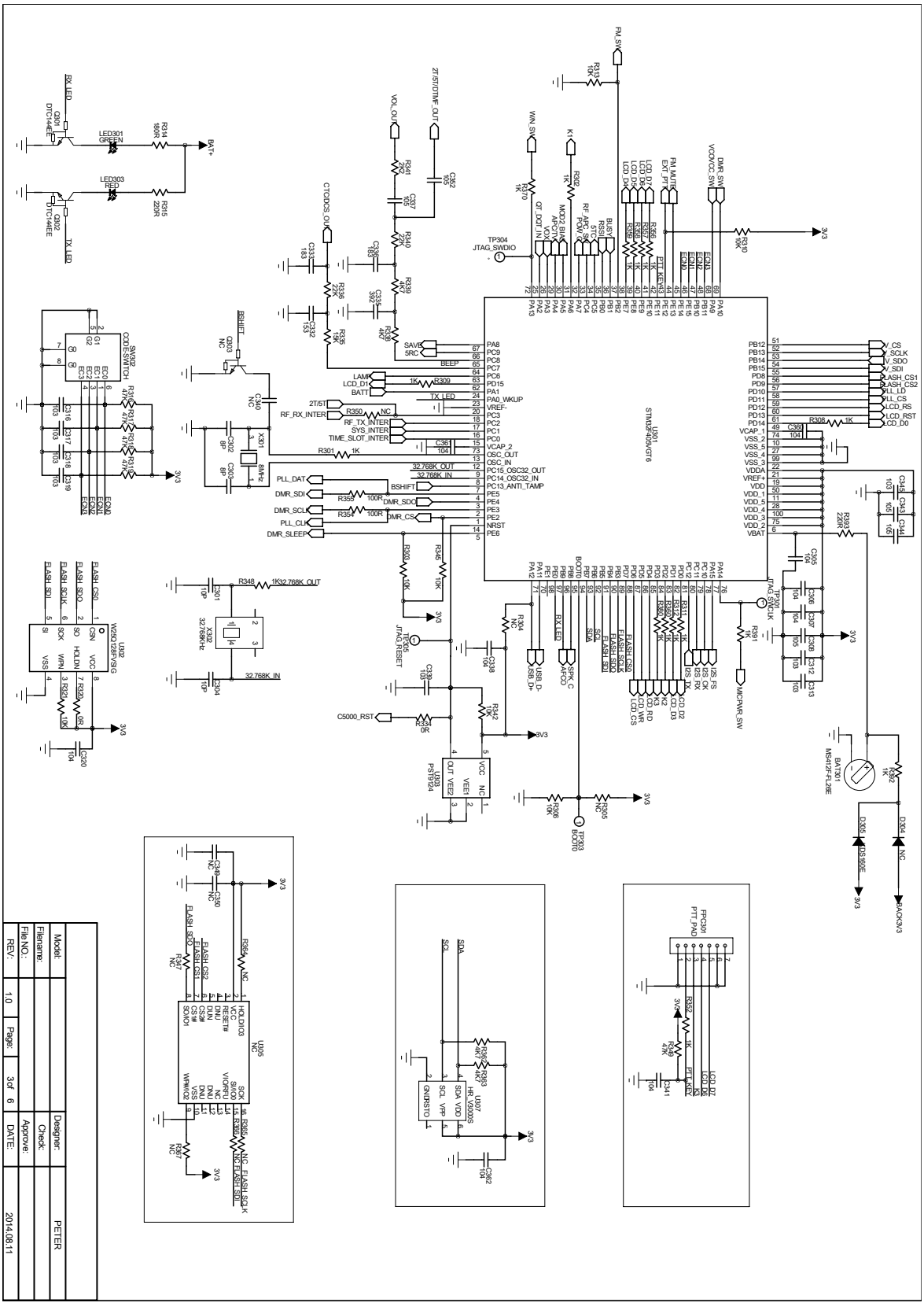
Patrick Hickey W7PCH has been playing around with from-scratch firmware for this platform, built around the FreeRTOS scheduler. His code is already linking into the memory that DD4CR freed up, and it's only a matter of time before fully-functional community firmware can be dual-booted on the MD380.

– – – – – – – – – – – –

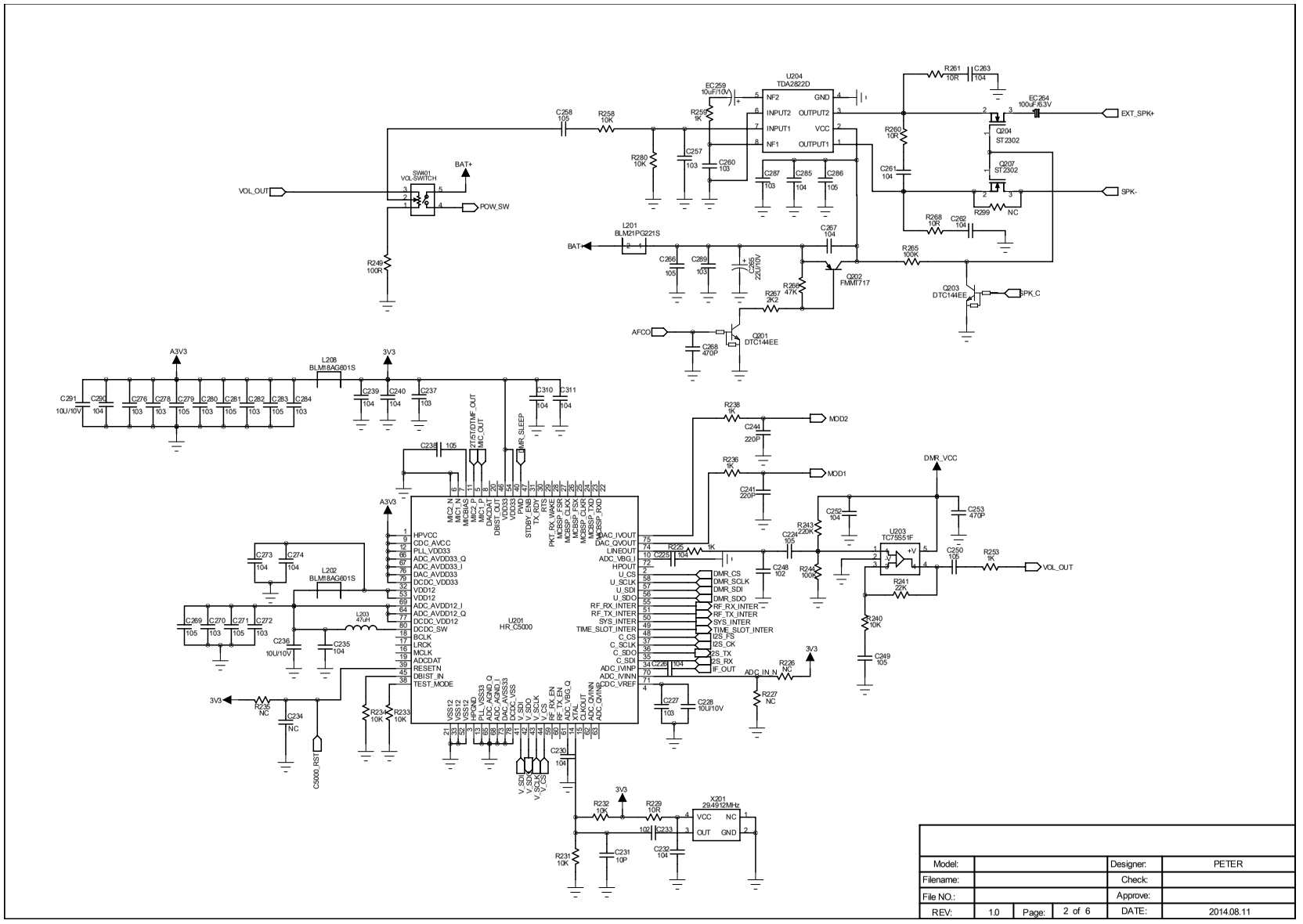
In this article, you have learned how to jailbreak your MD380 radio, dump a copy of its application, and begin patching that application or writing your own, new application.

Perhaps you will add support for P25, D-Star, or System Fusion. Perhaps you will write a proper scanner, to identify unknown stations at a whim. Perhaps you will make DMR adapter firmware, so that a desktop could send and receiver DMR frames in the raw over USB. If you do any of these things, please tell me about it!

Your neighbor,
Travis



Model:			
Filename:			
File NO.:			
REV:	1.0	Page:	3of 6
DATE:	2014.08.11	Approve:	
		Check:	
	PETER	Designer:	



Model:				Designer:	PETER
Filename:				Check:	
File NO.:				Approve:	
REV:	1.0	Page:	2 of 6	DATE:	2014.08.11