5 Decoding AMBE+2 in MD380 Firmware in Linux

by Travis Goodspeed KK4VCZ
with kind thanks to DD4CR, DF8AV, and AB3TL

Howdy y'all,

In PoC||GTFO 10:8, I shared with you fine folks a method for extracting a cleartext firmware dump from the Tytera MD380. Since then, a rag-tag gang of neighbors has joined me in hacking this device, and hundreds of amateur radio operators around the world are using our enhanced firmware for DMR communications.

AMBE+2 is a fixed bit-rate audio compression codec under some rather strict patents, for which the anonymously-authored Digital Speech Decoder (DSD) project[14] is the only open source decoder. It doesn't do encoding, so if for example you'd like to convert your favorite Rick Astley tunes to AMBE frames, you'll have to resort to expensive hardware converters.

In this article, I'll show you how I threw together a quick and dirty AMBE audio decompressor for Linux by wrapping the firmware into a 32-bit ARM executable, then running that executable either natively or through Qemu. The same tricks could be used to make an AMBE encoder, or to convert nifty libraries from other firmware images into handy command-line tools.

This article will use an MD380 firmware image version 2.032 for specific examples, but in the spirit of building our own bird feeders, the techniques ought to apply just as well to your own firmware images from other devices.

– — — – — — — – — – —

Suppose that you are reverse engineering a firmware image, and you've begun to make good progress. You know where plenty of useful functions are, and you've begun to hook them, but now you are ready to start implementing unit tests and debugging chunks of code. Wouldn't it be nicer to do that in Unix than inside of an embedded system?

As luck would have it, I'm writing this article on an aarch64 Linux machine with eight cores and a few gigs of RAM, but any old Raspberry Pi or Android phone has more than enough power to run this code natively.

Be sure to build statically, targeting `arm-linux-gnueabi`. The resulting binary will run on armel and aarch64 devices, as well as damned near any Linux platform through Qemu's userland compatibility layer.

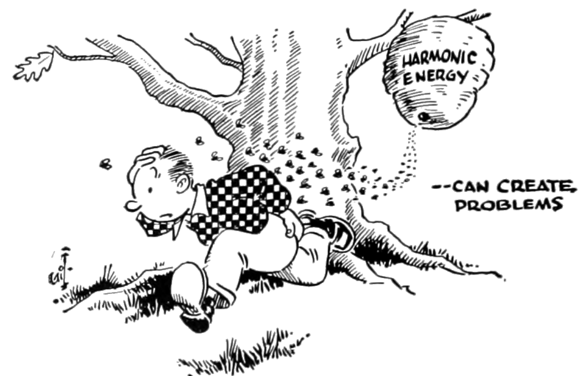
[14] `git clone https://github.com/szechyjs/dsd`

5.1 Dynamic Firmware Loading

First, we need to load the code into our process. While you can certainly link it into the executable, luck would have it that GCC puts its code sections very low in the executable, and we can politely ask `mmap(2)` to load the unpacked firmware image to the appropriate address. The first 48kB of Flash are used for a recovery bootloader, which we can conveniently skip without consequences, so the load address will be `0x0800c000`.

```
size_t length=994304;
int fd=open("experiment.img",0);
void *firmware=mmap(
     (void*) 0x0800c000, length,
     PROT_EXEC|PROT_READ|PROT_WRITE,
     MAP_PRIVATE,          //flags
     fd,                   //file
     0                     //offset
     );
```

Additionally, we need the 128kB of RAM at `0x20000000` and 64kB of TCRAM at `0x10000000` that the firmware expects on this platform. Since we'd like to have initialized variables, it's usually better go with dumps of live memory from a running system, but `/dev/zero` works for many functions if you're in a rush.



```c
//Load an SRAM image.
int fdram=open("ram.bin",0);
void *sram=mmap(
      (void*) 0x20000000,
      (size_t) 0x20000,
      PROT_EXEC|PROT_READ|PROT_WRITE,
      MAP_PRIVATE,           //flags
      fdram,                 //file
      0                      //offset
      );

//Create an empty TCRAM region.
int fdtcram=open("/dev/zero",0);
void *tcram=mmap(
      (void*) 0x10000000,
      (size_t) 0x10000,
      PROT_READ|PROT_WRITE,  //protections
      MAP_PRIVATE,           //flags
      fdtcram,               //file
      0                      //offset
      );
```

5.2 Symbol Imports

Now that we've got the code loaded, calling it is as simple as calling any other function, except that our C program doesn't yet know the symbol addresses. There are two ways around this:

The quick but dirty solution is to simply cast a data or function pointer. For a concrete example, there is a null function at `0x08098e14` that simply returns without doing anything. Because it's a Thumb function and not an ARM function, we'll have to add one to that address before calling it at `0x08098e15`.

```c
void (*nullsub)()=(void*) 0x08098e15;

printf("Trying to call nullsub().\n");
nullsub();
printf("Success!\n");
```

Similarly, you can access data that's in Flash or RAM.

```c
printf("Manufacturer is: '%s'\n",
       0x080f9e4c);
```

Casting function pointers gets us part of the way, but it's rather tedious and wastes a bit of memory. Instead, it's more efficient to pass a textfile of symbols to the linker. Because this is just a textfile, you can easily export symbols by script from IDA Pro or Radare2.

The symbol file is just a collection of assignments of names to addresses in roughly C syntax.

```c
/* Populates the audio buffer */
ambe_decode_wav = 0x08051249;
/* Just returns. */
nullsub = 0x08098e15;
```

You can include it in the executable by passing GCC parameters to the linker, or by calling `ld` directly.

```
CC=arm-linux-gnueabi-gcc-6 -static -g
$(CC) -o test test.c \
      -Xlinker --just-symbols=symbols
```

Now that we can load the firmware into process memory and call its functions, let's take a step back and see a second way to do the linking, by rewriting the firmware dump into an ELF object and then linking it. After that, we'll get along to decoding some audio.

5.3 Static Firmware Linking

While it's nice and easy to load firmware with `mmap(2)` at runtime, it would be nice and correct to convert the firmware dump into an object file for static linking, so that our resulting executable has no external dependencies at all. This requires both a bit of `objcopy` wizardry and a custom script for `ld`.

First, let's convert our firmware image dump to an ELF that loads at the proper address.

```
arm-linux-gnueabi-objcopy                  \
  -I binary experiment.img                 \
  --change-addresses=0x0800C000            \
  --rename-section .data=.experiment       \
  -O elf32-littlearm -B arm experiment.o
```

Sadly, `ld` will ignore our polite request to load this image at `0x08000C000`, because load addresses in Unix are just polite suggestions, to be thrown away by the linker. We can fix this by passing `-Xlinker -section-start=.experiment=0x0800C000` to `gcc` at compile time, so `ld` knows to place the section at the right address.

Similarly, the SRAM image can be embedded at its own load address.

5.4 Decoding the Audio

To decode the audio, I decided to begin with the same `.amb` format that DSD uses. This way, I could work from their reference files and compare my decoding to theirs.

The `.amb` format consists of a four byte header (`2e 61 6d 62`) followed by eight-byte frames. Each frame begins with a zero byte and is followed by 49 bits of data, stored most significant bit first with the final bit in the least significant bit of its own byte.

To have as few surprises as possible, I take the eight packed bytes and extract them into an array of 49 shorts located at `0x20011c8e`, because this is the address that the firmware uses to store its buffer. Shorts are used for convenience in addressing during computation, even if they are a bit more verbose than they would be in a convenient calling convention.

```
//Re-use the firmware's own AMBE buffer.
short *ambe=(short*) 0x20011c8e;

int ambei=0;
for(int i=1;i<7;i++){//Skip first byte.
  for(int j=0;j<8;j++){
    //MSBit first
    ambe[ambei++]=(packed[i]>>(7-j))&1;
  }
}
//Final bit in its own frame as LSBit.
ambe[ambei++]=packed[7]&1;
```

Additionally, I re-use the output buffers to store the resulting WAV audio. In the MD380, there are two buffers of audio produced from each frame of AMBE.

```
//80 samples for each audio buffer
short *outbuf0=(short*) 0x20011aa8;
short *outbuf1=(short*) 0x20011b48;
```

The thread that does the decoding in firmware is tied into the MicroC/OS-II realtime operating system of the MD380. Since I don't have the timers and interrupts to call that thread, nor the I/O ports to support it, I'll instead just call the decoding routines that it calls.

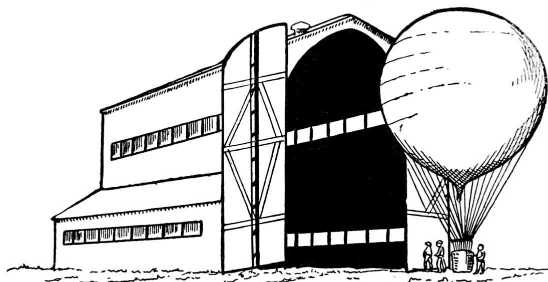
```
//Placed at 0x08051249
int ambe_decode_wav(
    signed short *wavbuffer,
    signed int eighty, //always 80
    short *bitbuffer,  //0x20011c8e
    int a4,            //0
    short a5,          //0
    short a6,          //timeslot, 0 or 1
    int a7             //0x20011224
    );
```

For any parameter that I don't understand, I just copy the value that I've seen called through my hooks in the firmware running on real hardware. For example, `0x20011224` is some structure used by the AMBE code, but I can simply re-use it thanks to my handy RAM dump.

Since everything is now in the right position, we can decode a frame of AMBE to two audio frames in quick succession.

```
//One AMBE frame becomes two audio frames.
ambe_decode_wav(
    outbuf0, 80, ambe,
    0, 0, 0,
    0x20011224
    );
ambe_decode_wav(
    outbuf1, 80, ambe,
    0, 0, 1,
    0x20011224
    );
```

After dumping these to disk and converting to a `.wav` file with `sox -r 8000 -e signed-integer -L -b 16 -c 1 out.raw out.wav`, a proper audio file is produced that is easily played. We can now decode AMBE in Linux!



5.5 Runtime Hooks

So now we're able to decode audio frames, but this is firmware, and damned near everything of value except the audio routines will eventually call a function that deals with I/O—a function we'd better replace if we don't want to implement all of the STM32's I/O devices.

Luckily, hooking a function is nice and easy. We can simply scan through the entire image, replacing all `BX` (Branch and eXchange) instructions to the old functions with ones that direct to the new functions. False positives are a possibility, but we'll ignore them for now, as the alternative would be to list every branch that must be hooked.

The `BL` instruction in Thumb is actually two adjacent 16-bit instructions, which load a low and high half of the address difference into the link register, then `BX` against that register. (This clobbers the link register, but so does *any* `BL`, so the register use is effectively free.)

```
/* Calculates Thumb code to branch from
   one address to another. */
int calcbl(int adr, int target){
  /* Begin with the difference of the target
     and the PC, which points to just after
     the current instruction.*/
  int offset=target-adr-4;
  //LSBit doesn't count.
  offset=(offset>>1);

  /* The BL instruction is actually two
     Thumb instructions, with one setting
     the high part of the LR and the other
     setting the low part while swapping
     LR and PC. */
  int hi=0xF000 | ((offset&0xFFF800)>>11);
  int lo=0xF800 | (offset&0x7FF);

  //Return the pair as a single 32-bit word.
  return (lo<<16)|hi;
}
```

Now that we can calculate function call instructions, a simple loop can patch all calls from one address into calls to a second address. You can use this to hook the I/O functions live, rather than trapping them.

5.6 I/O Traps

What about those I/O functions that we've forgotten to hook, or ones that have been inlined to a dozen places that we'd rather not hook? Wouldn't it sometimes be easier to trap the access and fake the result, rather than hooking the same function?

You're in luck! Because this is Unix, we can simply create a handler for SIGSEGV, much as Jeffball did in PoC||GTFO 8:8. Your segfault handler can then fake the action of the I/O device and return.

Alternately, you might not bother with a proper handler. Instead, you can use GDB to debug the process, printing a backtrace when the I/O region at `0x40000000` is accessed. While GDB in Qemu doesn't support `ptrace(2)`, it has no trouble trapping out the segmentation fault and letting you know which function attempted to perform I/O.

5.7 Conclusion

Thank you kindly for reading my ramblings about ARM firmware. I hope that you will find them handy in your own work, whenever you need to work with reverse engineered firmware away from its own hardware.

If you'd like to similarly instrument Linux applications, take a look at Jonathan Brossard's Witchcraft Compiler Collection,[15] an interactive ELF shell that makes it nice and easy to turn an executable into a linkable library.

The emulator from this article has now been incorporated into my md380tools[16] project, for use in Linux.

Cheers from Varaždin, Croatia,
–Travis 6A/KK4VCZ

[15] `git clone https://github.com/endrazine/wcc`
`unzip pocorgtfo13.pdf wcc.tar.bz2`

[16] `git clone https://github.com/travisgoodspeed/md380tools`