Creating the Longmynd Receiver

An exercise in reverse engineering

- Heather Lomond

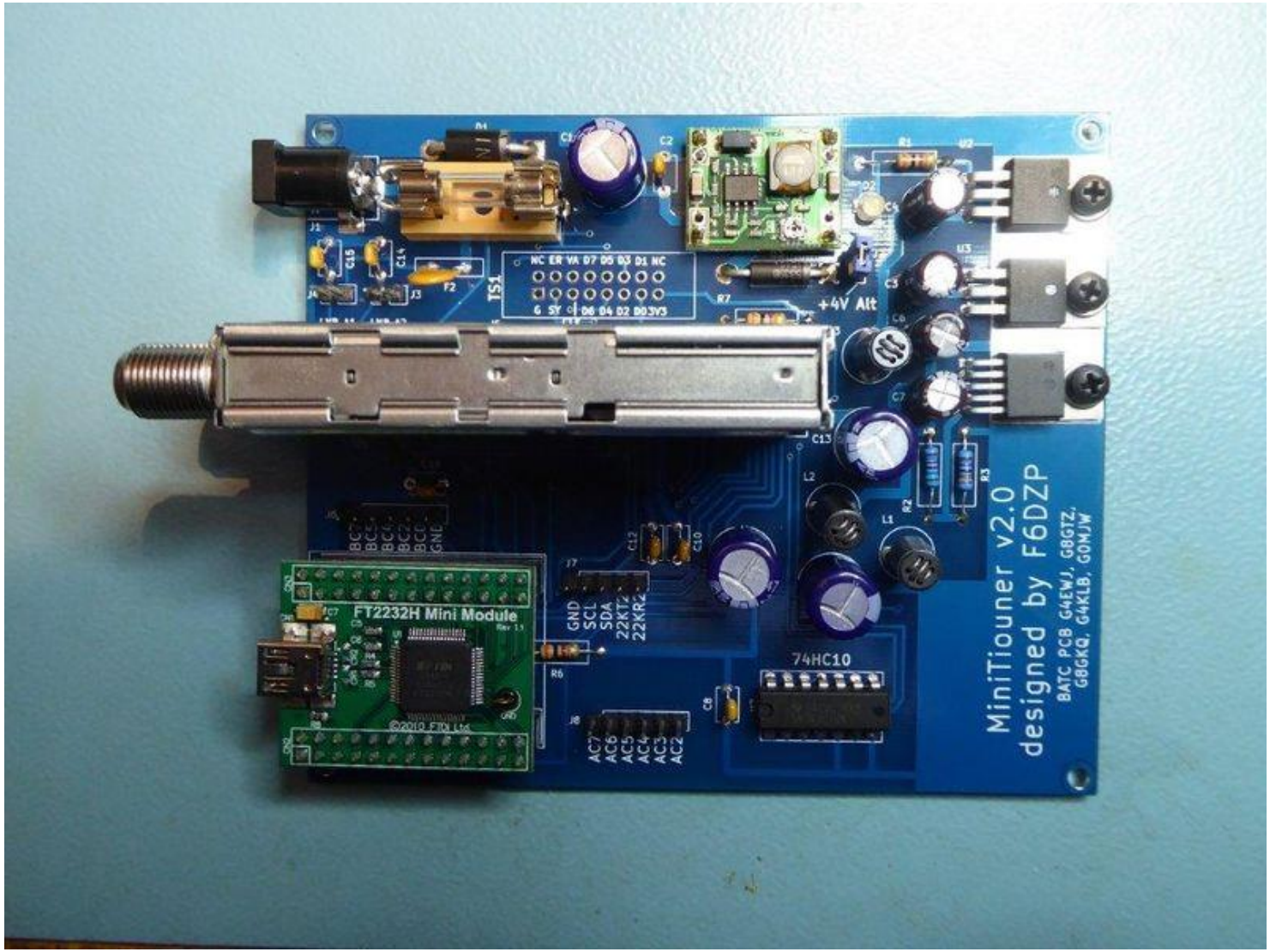
So What is the Problem?

For quite a few years, the Amateur TV community had relied on a product called Minitiouner: a nice hardware kit and Windows based, free software that is **Closed Source.** Developed by Jean-Pierre F6DZP.

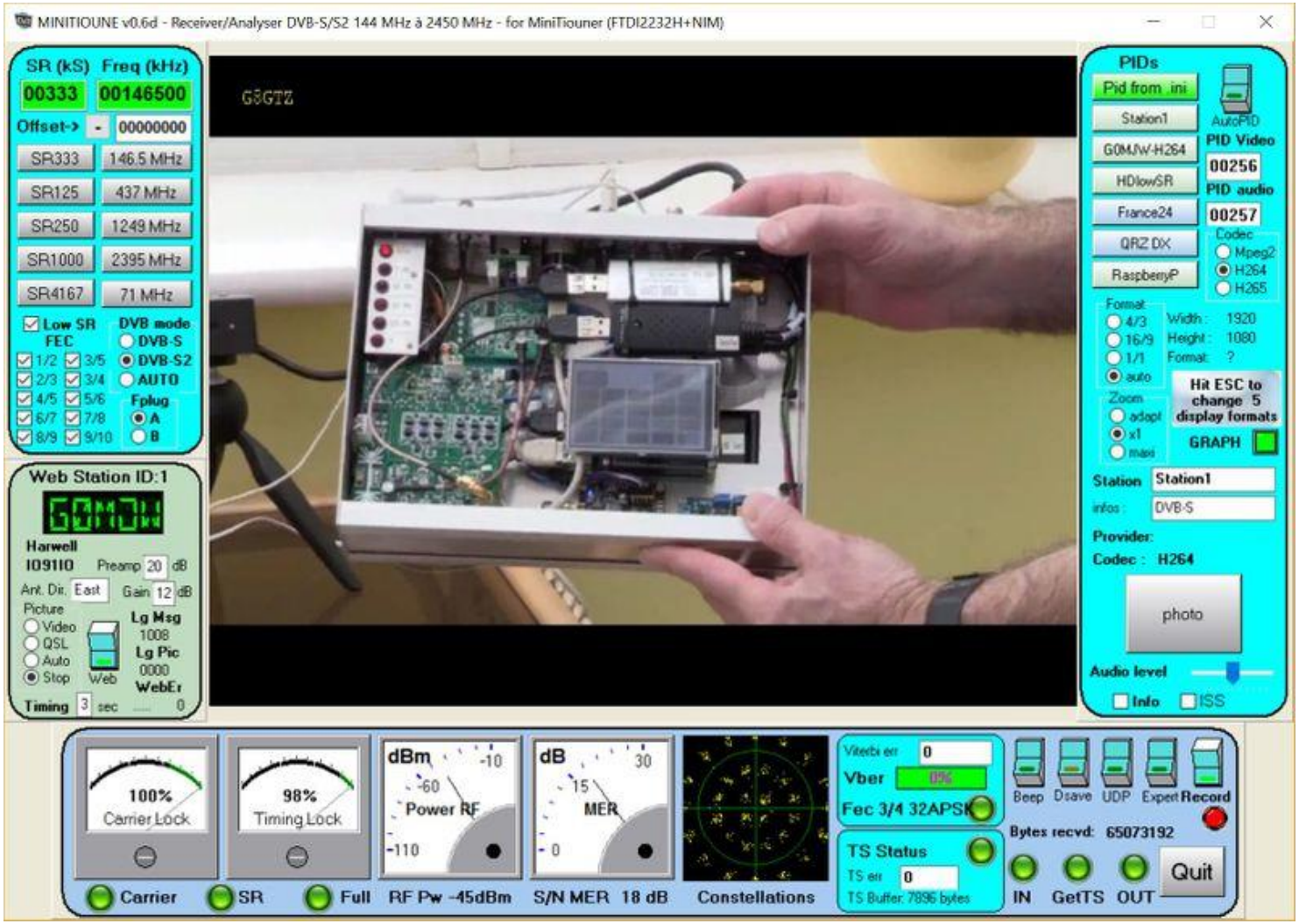
Since most ATV is done portable this meant that you had to have a **Windows PC** , a **PC display** and (mostly) **Mains Power** to see the TV pictures.

Amateur TV people love to tinker and play with things but with no source available this couldn’t happen.

Minitiouner HW



Minitioune SW



Enter Portsdown

About 5 years ago, a group of ATV enthusiasts developed the Portsdown ATV Transmitter based on a **Raspberry Pi**.

This was **Open Source**, suitable for **Portable** operation, could be run off **Batteries** and had a small **Touch Screen.**

Wouldn't it be nice if we could turn Portsdown into a TRX (Transmitter AND Receiver).

The concept of **Longmynd** was born.

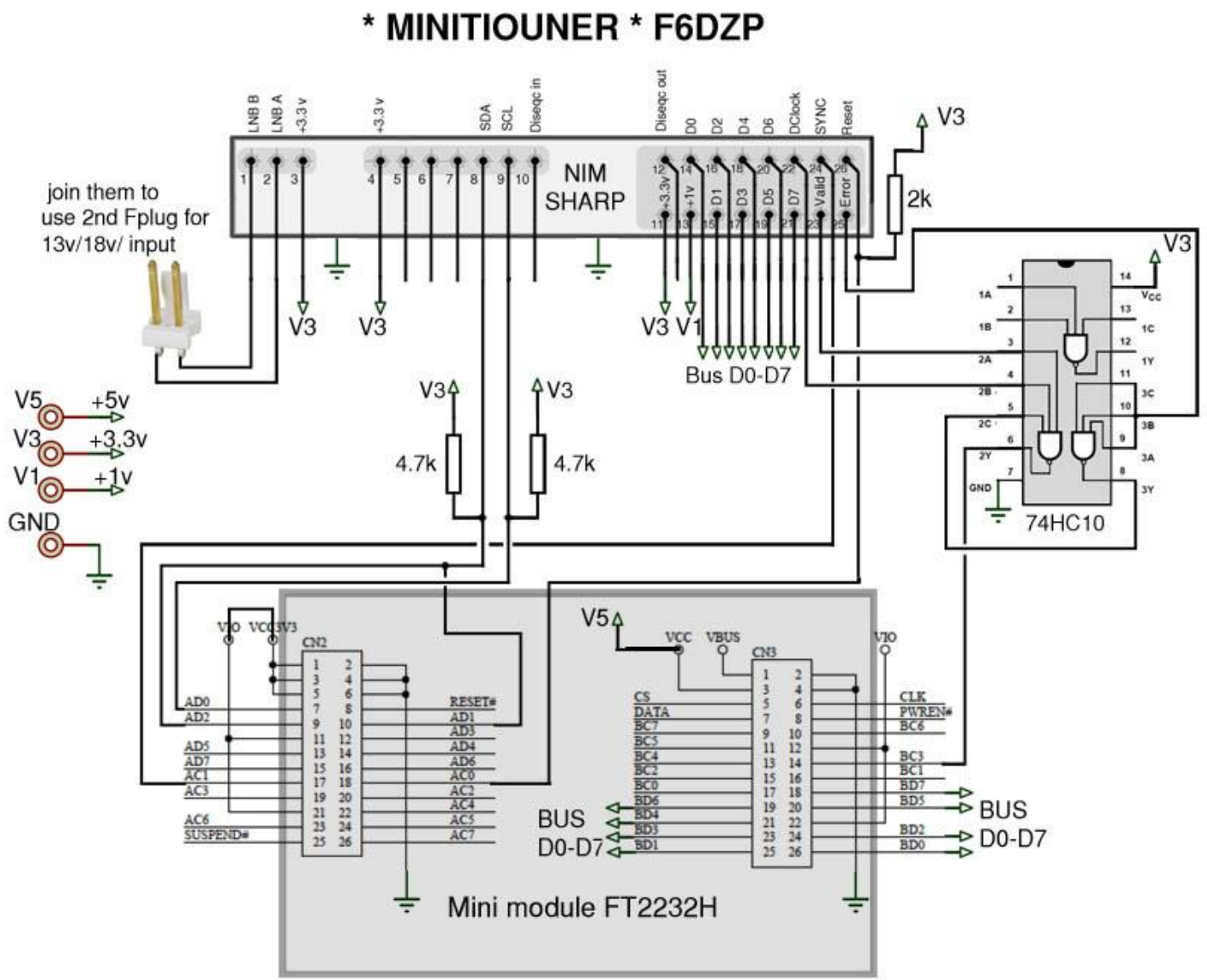
First off, is the RPI up to it?

As a test, I wrote some code for the Portsdown that would allow it to stream video from the internet and display it using one of the TS (transport steam) decoders available for the RPI.

Tests on this showed the RPI peaking at 20% CPU. Since USB is more than capable of streaming video, then the overall system should be able to cope quite easily.

Final test results showed **20% CPU** utilisation when streaming a 2MS video.

So, What is the Hardware Doing



Our Mission

The hardware interfaces with the outside world via USB. So, there are 2 parts to our problem:

- Replicate the FTDI API
- Replicate the NIM API
- Implement an RPI based decoder and display

Starting on the FDTI API

The FTDI module is programmable but all I had was a binary image for the program. So I needed to take a look at the USB traffic to see what was going on.

Enter **Wireshark**

Wireshark for the PC has a USB option. Since we had no idea what we were going to see, The first run was with no filters, just the USB port of interest.

The first system run

The data was gathered by:

- setup the Portsdown to transmit a video signal
- start Wireshark watching the USB traffic
- start the Minitioune software
- Instruct Minitioune to find the signal, lock on to it and display it
- Once we had video, stop Wireshark

(total data collection time about 21 seconds)

Wireshark output

There are 2 data views available from Wireshark:

- The **txt** file (interpretation)
- And the **bin** file of the raw data

Wireshark – 144 Mbytes txt file

```
      1 0.000000       host                  2.3.0                 USB      36     URB_CONTROL out

0000  1c 00 40 1b f0 b5 85 95 ff ff 00 00 00 00 17 00   ..@.............
0010  00 02 00 03 00 00 02 08 00 00 00 00 40 00 00 00   ............@...
0020  02 00 00 00                                       ....

      2 0.000228       2.3.0                 host                  USB      28     GET STATUS Status

0000  1c 00 40 1b f0 b5 85 95 ff ff 00 00 00 00 08 00   ..@.............
0010  01 02 00 03 00 00 02 00 00 00 00 02               ............

      3 0.000313       host                  2.1.0                 USBHUB   36     GET_STATUS Request
                                                                                      [Port 1]

0000  1c 00 a0 54 80 aa 85 95 ff ff 00 00 00 00 32 00   ...T..........2.
0010  00 02 00 01 00 80 02 08 00 00 00 00 a3 00 00 00   ................
0020  01 00 04 00                                       ....

      4 0.000475       2.1.0                 host                  USBHUB   32     GET_STATUS Response
                                                                                      [Port 1]

0000  1c 00 a0 54 80 aa 85 95 ff ff 00 00 00 00 32 00   ...T..........2.
0010  01 02 00 01 00 80 02 04 00 00 00 01 03 05 00 00   ................
```

A more useful format: trace.bin

```
  0: 0A 0D 0D 0A BC 00 00 00  4D 3C 2B 1A 01 00 00 00  ....¼...M<+.....
 10: FF FF FF FF FF FF FF FF  02 00 3D 00 20 20 20 20  ÿÿÿÿÿÿÿÿ..=.
 20: 20 20 49 6E 74 65 6C 28  52 29 20 43 6F 72 65 28    Intel(R) Core(
 30: 54 4D 29 20 69 37 2D 32  36 37 30 51 4D 20 43 50  TM) i7-2670QM CP
 40: 55 20 40 20 32 2E 32 30  47 48 7A 20 28 77 69 74  U @ 2.20GHz (wit
 50: 68 20 53 53 45 34 2E 32  29 00 00 00 03 00 1E 00  h SSE4.2).......
 60: 36 34 2D 62 69 74 20 57  69 6E 64 6F 77 73 20 31  64-bit Windows 1
 70: 30 2C 20 62 75 69 6C 64  20 31 37 31 33 34 00 00  0, build 17134..
 80: 04 00 2E 00 44 75 6D 70  63 61 70 20 28 57 69 72  ....Dumpcap (Wir
 90: 65 73 68 61 72 6B 29 20  32 2E 36 2E 33 20 28 76  eshark) 2.6.3 (v
 A0: 32 2E 36 2E 33 2D 30 2D  67 61 36 32 65 36 63 32  2.6.3-0-ga62e6c2
 B0: 37 29 00 00 00 00 00 00  BC 00 00 00 01 00 00 00  7).....¼.......
 C0: 80 00 00 00 F9 00 00 00  FF FF 00 00 02 00 35 00  ...ù...ÿÿ....5.
 D0: 5C 5C 2E 5C 70 69 70 65  5C 77 69 72 65 73 68 61  \\.\pipe\wiresha
 E0: 72 6B 5F 65 78 74 63 61  70 5F 5C 5C 2E 5C 55 53  rk_extcap_\\.\US
 F0: 42 50 63 61 70 32 5F 32  30 31 38 31 30 32 34 31  BPcap2_201810241
100: 31 34 39 30 34 00 00 00  09 00 01 00 06 00 00 00  14904...........
110: 0C 00 1E 00 36 34 2D 62  69 74 20 57 69 6E 64 6F  ....64-bit Windo
120: 77 73 20 31 30 2C 20 62  75 69 6C 64 20 31 37 31  ws 10, build 171
130: 33 34 00 00 00 00 00 00  80 00 00 00 06 00 00 00  34............
```

And on and on … 32 Mbytes

```
140: 44 00 00 00 00 00 00 00  F7 78 05 00 17 81 B3 3C  D.......÷x...³<
150: 24 00 00 00 24 00 00 00  1C 00 40 1B F0 B5 85 95  $...$.....@.ðµ
160: FF FF 00 00 00 00 17 00  00 02 00 03 00 00 02 08  ÿÿ..............
170: 00 00 00 00 40 00 00 00  02 00 00 00 44 00 00 00  ....@.......D...
180: 06 00 00 00 3C 00 00 00  00 00 00 00 F7 78 05 00  ....<.......÷x..
190: FB 81 B3 3C 1C 00 00 00  1C 00 00 00 1C 00 40 1B  û³<..........@.
1A0: F0 B5 85 95 FF FF 00 00  00 00 08 00 01 02 00 03  ðµÿÿ............
1B0: 00 00 02 00 00 00 00 02  3C 00 00 00 06 00 00 00  ........<.......
1C0: 44 00 00 00 00 00 00 00  F7 78 05 00 50 82 B3 3C  D.......÷x..P³<
1D0: 24 00 00 00 24 00 00 00  1C 00 A0 54 80 AA 85 95  $...$..... Tª
1E0: FF FF 00 00 00 00 32 00  00 02 00 01 00 80 02 08  ÿÿ....2.........
1F0: 00 00 00 00 A3 00 00 00  01 00 04 00 44 00 00 00  ....£.......D...
200: 06 00 00 00 40 00 00 00  00 00 00 00 F7 78 05 00  ....@.......÷x..
210: F2 82 B3 3C 20 00 00 00  20 00 00 00 1C 00 A0 54  ò³< ... ..... T
220: 80 AA 85 95 FF FF 00 00  00 00 32 00 01 02 00 01  ªÿÿ....2.....
230: 00 80 02 04 00 00 00 01  03 05 00 00 40 00 00 00  ............@...
240: 06 00 00 00 3C 00 00 00  00 00 00 00 F7 78 05 00  ....<.......÷x..
250: F2 82 B3 3C 1C 00 00 00  1C 00 00 00 1C 00 A0 54  ò³<..........  T
260: 80 AA 85 95 FF FF 00 00  00 00 32 00 01 02 00 01  ªÿÿ....2.....
```

Deducing Wireshark's packet shape

The txt file shows the USB traffic data is packetized.

The bin file has a header, followed by the packet data which consists of some Wireshark inserted data (timing etc.) then the actual transmitted/received USB data.

As we study the data, we can see that the Wireshark packets contain `0x1c` bytes of timing data, followed by a 4 byte packet size, then followed by that amount of data. Everything is happening on 4 byte boundaries.

So, now we can start coding

First off I wrote a program to process the bin file.
So:

Read the Wireshark header and throw it away then:

- Read and discard each Wireshark timing header
- Read the 4 byte size counter
- Read the actual data
- Read to the end of the 4 byte boundary
- Repeat for each packet

USB messages - Types

We now have just the packets sent over USB.

Next step is to understand what they are instructing the FTDI chip to do..

USB protocol has a whole set of pre-defined messages that are sent to devices to start them, stop them, send data, request data etc.

The one that sends data is the **USB BULK OUT** message.

USB messages - Endpoints

USB protocol also defines **EndPoints**.

These are basically ways of splitting a single physical USB port into different virtual devices.

Usually there is a Control endpoint and a data endpoint.

In the code that we see, there are a number of endpoints as the data out from the NIM comes via it's own endpoint (and associated control endpoint).

An example Bulk Out

```
 74171 14.062989      host                    2.3.2                   USB      64     URB_BULK out

0000  1b 00 40 1b f0 b5 85 95 ff ff 00 00 00 00 09 00   ..@.............
0010  00 02 00 03 00 02 03 25 00 00 00 80 03 13 80 03   .......%........
0020  13 80 03 13 80 03 13 80 01 13 80 01 13 80 01 13   ................
0030  80 01 13 80 00 13 11 00 00 d2 80 00 11 27 00 87   .............'..
```

Here we can see an example BULK_OUT message.

It starts with a USB header byte counter (`1b`).
Then the header that specifies it is a bulk out.
Then a 4 byte data size field (`25 00 00 00`).

Following this header comes the actual data:

```
80 03 13 80 03 13 …
11 00 00 d2 80 00 11 27 00 87
```

Decoding the FTDI API

So, what do these data bytes tell the FTDI module?

There is a data sheet for the NIM itself – this tell us that there are a number of I2C devices inside it (Amplifiers, Demodulators (address=`0xd2`), Tuners etc.).

So, the FTDI code must be taking in data and outputting it to I2C. A search of the app notes found FTDI AN_255:

USB to I2C Example using the FT232H and FT201X devices - Bingo!

The app note formats:

In this app note we find a series of subroutines that will allow control of the I2C bus. Here is an example:

```
BOOL SendByteAndCheckACK(BYTE dwDataSend)
{
        dwNumBytesToSend = 0;
        OutputBuffer[dwNumBytesToSend++] = 0x11;
        OutputBuffer[dwNumBytesToSend++] = 0x00;
        OutputBuffer[dwNumBytesToSend++] = 0x00;
        OutputBuffer[dwNumBytesToSend++] = dwDataSend;
        OutputBuffer[dwNumBytesToSend++] = 0x80;
        OutputBuffer[dwNumBytesToSend++] = 0xFE;

        ...
```

Each USB transfer has a specific command at the start (e.g. `0x11` in this example).

Decoding our first message

```
80 03 13 80 13 80 03 13 80 03 13 80 01 13 80
01 13 80 01 80 01 13 80 00 13 11 00 00 d2 80
00 11 27 00 87
```

Based on the earlier send byte routine, we can see that this message (almost) has that sequence in it:

```
11 00 00 d2 80 00 11 27
```

But, it isn't quite right. (the red bits).

What is going on

Apart from the command byte (the first one) and the data bytes, the other messages are controlling the states of the FTDI GPIO pins.

Clearly the Minitiouner is using some of these pins in a slightly different way … but we can see from the datasheet that it is controlling the USB lines exactly as the example code does … now we are cooking on GAS (but not for too many more years).

Long Story Short

We can now extend our parser to actually understand the commands being sent to the FTDI module. These are going to be in 2 parts:

- Setup of the FTDI chip
- Data to send over the I2C

This is a long, iterative, slog of coding but basically means that we can extract all the USB messages as per the FTDI app note.

Creating the state machine

Once we have decoded each FTDI message, we can implement a state machine to interpret the USB traffic.

E.g. an I2C register write to the demodulator looks like:

- Set I2C Start state
- Send `0xd2` and check ACK
- Send data byte and check ACK
- Set I2C stop state
- Set I2C pins idle

So if we see all of these commands in this order we know we are writing to the demodulator register.

Our parser output (662 kBytes)

Now, we can convert our huge data file into something a little more readable (note the output is pseudo code):

```
…
nim_write_demod(0xf12a,0x38)
nim_write_demod(0xf12a,0xb8)
nim_read_tuner(0x05,&val) = 0x1a
nim_write_demod(0xf12a,0x38)
nim_write_demod(0xf12a,0xb8)
nim_read_tuner(0x06,&val) = 0x80
nim_write_demod(0xf12a,0x38)
nim_write_demod(0xf12a,0xb8)
nim_read_tuner(0x08,&val) = 0x0b
nim_write_demod(0xf12a,0x38)
nim_write_demod(0xf12a,0xb8)
nim_read_tuner(0x06,&val) = 0x80
nim_write_demod(0xf12a,0x38)
nim_write_demod(0xf12a,0xb8)
nim_read_tuner(0x01,&val) = 0x30
nim_write_demod(0xf12a,0x38)
…
```

What do we observe in the file

The parse starts off with a load of setup stuff (as expected).

```
…
Received 230 fn=0x08 in message
ftdi_send_byte(0xaa)
ftdi_send_byte(0xaa)
Replied with error code 0xfa 0xaa
ftdi_send_byte(0xab)
Replied with error code 0xfa 0xaa 0xfa 0xab
ftdi_setup(0x8a,0x97,0x8d,0x80,0x13,0x13,0x82,0x6f,0xf1,0x86
,0x95,0x00,0x85)
ftdi_read_highbyte(0x83,&val)=0x65
…
```

This is all in keeping with the FTDI App note.

More Observations

Then we have over 600 repetitions of this **weirdness**:

```
…
nim_send_d0()
nim_send_d0()
nim_send_d0()
…
```

Then about 700 consecutive writes to the demodulator (presumably initialisation) such as (**remember this for later**):

```
nim_write_demod(0xf113,0x00)
nim_write_demod(0xf114,0x00)
nim_write_demod(0xf11a,0x05)
```

Yet More Observations

Followed by 20 writes to the tuner:

```
…
nim_write_tuner(0x00,0x75)
nim_write_tuner(0x01,0x50)
nim_write_tuner(0x02,0xce)
…
```

And then a few writes to each of the 2 Amplifiers (LNAs):

```
…
nim_write_c8(0x00,0x20)
nim_write_c8(0x01,0x0f)
nim_write_c8(0x02,0x50)
…
```

Finally we find a general mix of reads and writes until we start getting huge BULK IN messages which we can assume is the actual video stream.

A couple of notes

When you are sending USB data, sometimes there will be a bus clash and you won't get a valid response from the USB controller. When parsing, we sometimes see that things go out of sync and the Windows code retries the send. This makes life a little more complex.

Also, there are a lot of individual setup commands required to get the FTDI chip into the right state to do the I2C handling. Parsing this was a lot of work, but basically followed the FTDI data sheet.

Decoding the NIM API

Once we have all of these register writes, we can start to look for repeating patterns (which will hopefully correspond to loops in the original code).

Each time I found a pattern, I wrote a program to replace the pattern with a single line e.g.

```
…
nim_read_demod(0xf201,&val) = 0x16
nim_read_demod(0xf201,&val) = 0x16
Got rw 1
Got rw 1
Got rw 1
…
```

First steps into decoding register reads

The first thing to note when I did this was that the end of the file now looked like:

```
…
Got loop 4
Got loop 4
Got loop 4
…
```

This I deduced was the monitoring phase after it had started receiving video data.

By comparing the values coming back from the register reads I could deduce some of the register meanings.

So where are we?

The code structure appears to be:

- Set up the hardware first (with the FTDI writes)
- Then it initialises the demodulator, the tuner and the amplifiers with default values.
- Finally, when it is instructed to scan for a video picture, it starts the scanning process
- Monitors until it finds a signal
- and begins displaying the video stream and monitoring the status for display

First steps in coding

From the FTDI datasheet and the parsed code, we can now do everything we need to set up the FTDI module.

- That was easily coded on the RPI.

Obviously this required a USB interface so that was easily implemented with **libusb**.

Since this was to be integrated into the Portsdown (which will handle all the user interactions) all parameters required can be passed on the command line.

USB – VID/PID

When a USB device is attached to a host machine, it identifies itself with a VID/PID pair.

Since I wanted flexibility in my design, I wanted to allow 2 Minitiouners to be attached.

To solve this I identify the Minitiouners from their PID and VID but access them via their Bus/Port numbers on the actual device.

Read/Write Testing

My first coding target was to read a single register in each of the tuner, amplifiers and the demodulator.

From the parsing, I knew what the registers should contain so coding this was quite simple. It all worked well, apart from reads of the tuner and the amps.

Going back to the parse file, I noted that each such read was bracketed by modulator writes:

```
nim_write_demod(0xf12a,0xb8)
nim_read_tuner(0x06,&val) = 0x80
nim_write_demod(0xf12a,0x38)
```

Revelations

It appears that the I2C on the Tuner and the Amplifiers are accessed via a “gateway” on the demodulator.

This makes sense from an electrical point of view as I2C traffic can be **isolated** (and thus not cause interference) with the tuner and amps which are very sensitive to electrical noise.

This fix meant that I could now read (and write!) to all the registers.

Data Output

As a test, I implemented a simple interface (using Linux FIFOs) that would read video stream data from a file and output it to a FIFO. This formed the basis of the video output and display.

I also added a FIFO to output status information (remember the status reporting used by Minitioune when it had a lock on the video).

Linux FIFOs

Linux FIFOs are a very efficient implementation allowing cross process communication.

```
fd_status_fifo = open(status_fifo, O_WRONLY);
```

But, the fill up and choke, so I wrote a program to keep emptying them (note the read and write options)!

```
ret=mkfifo("longmynd_main_status", 0666);
fd_status_fifo = open("longmynd_main_status", O_RDONLY);

num=read(fd_status_fifo, status_message, 1);
```

Video Rendering

I then hooked up the video streaming software (already available for the RPI and installed via Portsdown) to this FIFO so that I could prove out the mechanisms.

```
mkfifo fifo.264
mkfifo longmynd_main_ts
/home/pi/longmynd/longmynd 436868 250 &
/home/pi/rpidatv/bin/ts2es -video longmynd_main_ts fifo.264 &
/home/pi/rpidatv/bin/hello_video.bin fifo.264 &
```

Once this worked, I knew that I could display TS video when I managed to get it out of the NIM.

Error Handling

Normally with Embedded projects, there is no point in doing any error handling as there is no way to do anything about them if they occur.

However, in Open Source code, it is very much simpler to understand code if, when playing with it, it reports errors.

So **Every Routine Reports Errors** and **Call Stacks** associated with them

Also all init routines report their status for **Trace** purposes

And, because I could …

I added an interface to the longmynd code to allow the video stream to be send over UDP to another host – such as a cellphone or tablet – in case people wanted a bigger picture!

```
sudo /home/pi/longmynd/longmynd -i 192.168.1.9 1234 436868
250 &
```

Note that some Routers don’t pass UDP very well!

Decoding the NIM API

Already available on Linux was an STV910 driver (the Tuner). But it was incomplete.

ST Micro are only prepared to release the datasheets under NDA.

- I managed to get taken on as an unpaid member of staff of a small company to get them

API Overview

- 977 registers on the NIM
- Over 2000 fields
- 2 LNAs
- 2 Tuners
- 2 Demodulators
- Just under 10,000 lines of code

Some highlights of the code development

Do you remember the parse file showed a section where ever register was written as part of the initialisation?

One thing puzzled me, he wrote to all the registers in turn but on one or two occasions he did a write that was completely out of sequence and duplicated elsewhere This is an extract from the datasheet init code.

```
…
{ RSTV0910_P1_PDELCTRL2,    0x00 }     ,/* P1_PDELCTRL2 */
{ RSTV0910_P1_HYSTTHRESH,   0x41 }     ,/* P1_HYSTTHRESH */
{ RSTV0910_P1_UPLCCST0,     0xe6 }     ,/* P2_UPLCCST0 */
{ RSTV0910_P1_ISIENTRY,     0x00 }     ,/* P1_ISIENTRY */
{ RSTV0910_P1_ISIBITENA,    0x00 }     ,/* P1_ISIBITENA */
…
```

Coding principles

- Detailed code **Documentation** explaining every action so that others could climb the learning curve.
- **Modular** design so that new NIMs could be easily accommodated.
- **Common structure** to each driver code.
- **Flexible initialisation** routines to allow all possible configurations (allow all devices to interact with all other devices (E.g. allow LNA1 - Tuner2 - Demod1 - USB2 etc).

Example code

```
/* the global rdiv has already been set up in the init routines */

/* p is defined from the datasheet (note, this is reg value, not P) */
if      (freq<=STV6120_P_THRESHOLD_1) p=3; /* P=16 */
else if (freq<=STV6120_P_THRESHOLD_2) p=2; /* P= 8 */
else if (freq<=STV6120_P_THRESHOLD_3) p=1; /* P= 4 */
else                                  p=0; /* P= 2 */

/* we have to be careful of the size of the typesi in the following  */
/* F.vco=F.rf*P where F.rf=F.lo    all in KHz */
/* f_vco is uint32_t, so p_max is 3 (i.e P_max is 16), freq_max is    */
/* 2500000KHz, results is 0x02625a00 ... OK */
f_vco = freq<<(p+1);
/* n=integer(f_vco/f_xtal*R)  note: f_xtal and f_vco both in KHz */
/* we do the *R first (a shift by rdiv), and max is 0x04c4b400, then */
/* the divide and we are OK */
```

Success So Far

- At the RSGB Construction Contest in 2019 there were 4 Portsdown's entered. All 4 were running Longmynd as the receiver. **Longmynd won the software section**.
- The **Ryde Set Top Box** was developed on top of Longmynd.
- The **Winterhill Multichannel Recevier** was developed on top of Longmynd.
- Longmynd has been ported to **Windows** as Open Source.
- An Estimated **200 Instances** are installed worldwide.

Questions