SOLVING THE T2'16 CHALLENGE

by Alexander Polyakov

The challenge begins with a link to a password-protected ZIP archive (password: t216).

The archive contains 17 files - 11 e-mails and 6 attachments.

All files have the same date and time: 2016-09-06 **18:18**:20 . (6:18 PM? that's the Judgment Day time from "Terminator 3"!)

The puzzles can be solved in any order.

Tools used:

- Google and Wolfram Alpha;
- FAR Manager for writing C code, analyzing the firmware disassembly and creating this write-up;
- Borland C++ command-line tools;
- IrfanView and Firefox for viewing pictures;
- Visual Studio for some debugging of the taximeter emulator;
- Wireshark for converting the PCAP to text;
- grep for exploring truth tables;
- Audacity.

mail0008: The USB capture

The mail0008-attachment.bin starts with bytes D4 C3 B2 A1.
I google "D4 C3 B2 A1" and find out that it's a PCAP file. I open the file in Wireshark.

This is an USB capture. It was done on 2016-08-11 between 20:34:50 and 20:39:10 EEST.
There were two devices: Logitech UltraX keyboard (Y-BL49) and Logitech optical wheel mouse.

I export the data to a text file for further processing.
As stated in Wireshark Q&A, the actual data are in the field "Leftover Capture Data" (8 bytes per field for keyboard; 4 bytes per field for mouse).

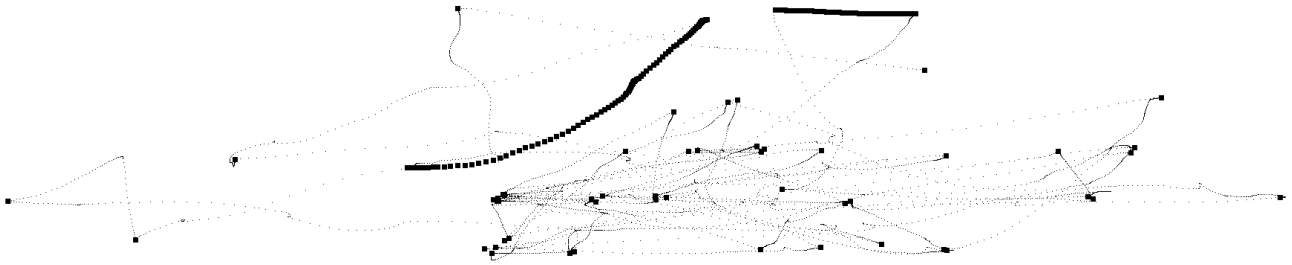
I hack together a decoder for keyboard data (see page #53 in this spec) and get this:

```
{Alt}+{Tab}
NEVER{Backspace} {Backspace} {Backspace} {Backspace} {Backspace}
When it rains it puo{Backspace} {Backspace} ours:
GONNA{Backspace} {Backspace} {Backspace} {Backspace} {Backspace}
the guys investigating the incident are now convinced that someone has installed physical keyloggers to our systems...
GIVE{Backspace} {Backspace} {Backspace} {Backspace}
Many seem to think AAA is behind this but I can't believe even Astley could stoop so low.
YOU{Backspace} {Backspace} {Backspace}
Anyway, the new instt{Backspace} ructions are to use the on-screen keyboard for typing all the sensitive stuff.
UP{Backspace} {Backspace}
Here's the new master key:{Enter} {Enter}
{Win}+r{Enter}
{Alt}+{Tab} {Alt}+{Tab} {Alt}+{Tab} {Alt}+{Tab} {Alt}+{Tab} {Alt}+{Tab} {Alt}+{Tab} {Alt}+{Tab} {Alt}+{Tab}
```

Note the deleted words - "NEVER GONNA GIVE YOU UP". That's a song by Rick Astley.

So the task is probably to get the master key. But it was entered using virtual keyboard! (Note the WIN+R combo - it was probably used to launch osk.exe)

I add code for decoding mouse movements/clicks (see this page for more info) and plotting them (tiny dots for mouse movements, squares for clicks). The result:

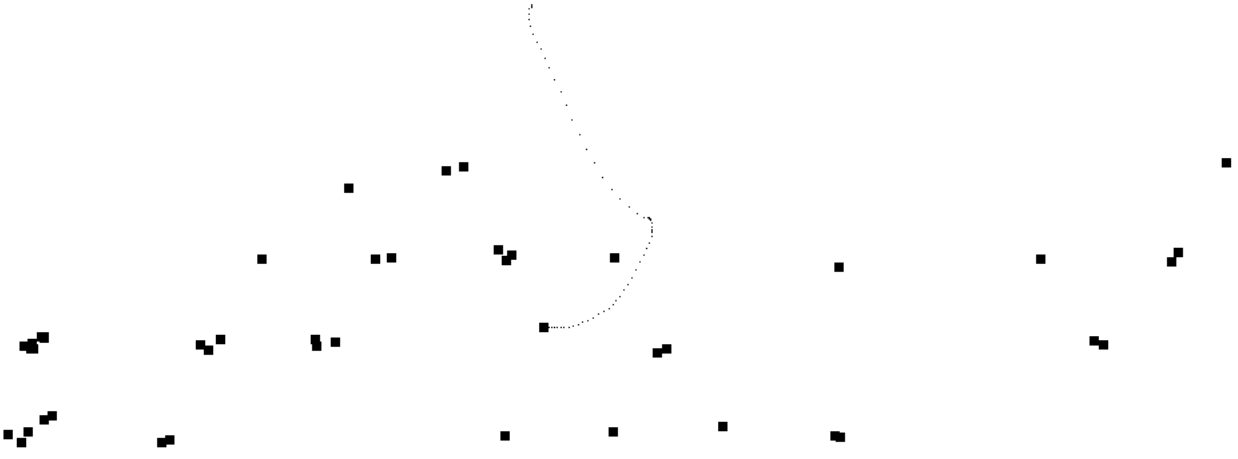


Oh, what a mess! Nevertheless, clicks on the virtual keys are visible.

I modify the code:

- exclude movements and clicks that are not related to the virtual keypresses;
- draw all clicks in each picture, but only one movement per picture;

and create 47 pictures. Here's the first one for example:



Using these pictures and standard US keyboard layout, I decode the keypresses manually and get the following result:

g;shzE4{Enter}TRm-DYtsMHRTd3IDN[vs{:pB4Z

Wait... what is that {Enter} keypress doing here???
This is wrong.
Needless to say, the site does not accept my answer.

I carefully re-analyze the keypresses on the real USB keyboard and notice that dot and colon are on the same key.
This isn't the case for US keyboard layout.
Which keyboard layout has . and : on the same key? German!
Oh, and Gruber is German surname!

I decode the virtual keypresses again using German keyboard layout and get this:

flag<W3#REn0STraNGERs2USBpcaPLoV3>

Now there's no {Enter} key, the password makes sense and is of flag<text> format! Hurray!
However, the site still does not accept my answer!!

So it's almost right, but not quite. The layout isn't German, but it's similar - in particular, there's a key with "less than" and "greater than" signs next to the Shift key, and there are 12 keys between CAPS LOCK and ENTER.

I find this site with keyboard layouts and start to examine them. Finally, I get to the Finnish layout and...

flag<W3'REn0STraNGERs2USBpcaPLoV3>

Compare the password ("We're no strangers to USB PCAP love") with the first line from "Never Gonna Give You Up" ("We're no strangers to love").

mail0007: The password

Obviously the task is to find the password.
But it contains 35 characters! That's a bit too much to bruteforce.
I need to reduce the search space first by applying rules from mail0007-attachment.bin.

Some observations:

- 35 - (17+11+5) = 2: there are 2 characters that are not letters nor digits (let's call them magic characters);
- no characters have bit 7 set - i.e. they are in the range [0..127] (ASCII);
- let's suppose that the password does not contain chars ranged [0..31] (they are non-printable control codes). This further reduces range to [32..127];
- there are 35 - 28 = 7 chars with bit 6 equal to 0. They cannot be letters (all letters have bit 6 set), so they are digits and magic chars; i.e. all magic chars are in the range [32..63];
- there are 35 characters and (16+17+19+12+18+24+28+0) = 134 bits set, that gives us 134/35 = 3.83 bits set per char in average;
- for 8-bit char, number of bits set ranges from 0 to 8. But in our case bit 7 is always clear; thus, the range is [0..7];
  The number of bits set for each char of the password is either:
  - minimal (N_{min});
  - greater than minimal, but less than average (N_{less});
  - greater than average but less than maximal (N_{more});
  - maximal (N_{max}).
- there are 4 chars with N_{min} bits set. They cannot be magic chars, because there are only 2 magic chars. And all letters and digits have at least 2 bits set, so $N_{min} = 2$, and $N_{less} = 3$;
- high nibble of char #18 is equal to its low nibble (check equality statements below), so N_{max} is even (4 or 6). But if N_{max} is 4, there are no possible values for N_{more}. So $N_{max} = 6$, and $N_{more} = 4$ or 5;
- the first character of the password is hexadecimal digit, i.e. one of these: 0123456789ABCDEFabcdef .

Each char equality statement can be turned into two nibble equality statements.
Here are the equality statements (H - high and L - low nibble of char #NN):

- `0aaa = H#06 = L#23 = H#10 = L#06 = H#20 = H#26`
- `0bbb = H#08 = L#11 = L#17 = L#21 = L#29 = L#32 = H#24 = H#28 = H#29 = H#31 = H#32 = H#34`
- `0ccc = H#09 = L#13 = L#16 = L#25 = L#30 = H#13 = H#17 = H#19 = H#25`
- `0ddd = H#11 = L#10 = L#18 = H#14 = H#16 = H#18 = H#22 = H#23 = H#30`
- `eeee = L#09 = L#19`
- `ffff = L#08 = L#28`
- `0ggg = H#03 = H#15 = H#27`
- `hhhh = L#03 = L#15 = L#27`

All H-nibbles have high bit set to 0, so if some L-nibble is equal to an H-nibble, its high bit is also 0.
Interestingly, some chars are not mentioned in these rules at all (chars #1..#5, char #35 and some others).

So I take all this info and write a program. (See Appendix B for the code.) The program applies rules and outputs results.

The first results: char #18 is 'w'. So 0ddd is 0111. I add this to the program and re-run.

Char #14 is 'p'. Now look at char #29 (or #32). It's 0x22 or 0x44. So 0bbb = 2 or 4. But if 0bbb = 2, there are more than two magic chars! So 0bbb = 4. Add this to the program, re-run.

Char #11 is 't', char #29/#32 is 'D'. Now look at char #13 and note that 0ccc is either 3, 5 or 6. Now look at char #9. Only 2 bits set, and both belong to 0ccc, so eeee is 0000. This also means that 0ccc can't be 6, because then char #9 will be 0x60 (magic char > 0x3F). So 0ccc = 3 or 5, i.e. 0ccc = 0cc1. Add, re-run.

Now look at char #6. It has 0aaa in both nibbles. That means it has 4 bits set (not 5), and 0aaa has 2 bits set, so 0aaa is 3, 5 or 6. Add, re-run.

Umm... now what? 0cc1 = 3 or 5, and 0aaa = 3, 5 or 6.

- Suppose 0cc1 = 3, 0aaa = 3. Add, re-run... Oops. No valid characters left for #13.
- Suppose 0cc1 = 5, 0aaa = 3. Add, re-run. Too many uppercase letters.
- Suppose 0cc1 = 5, 0aaa = 3. Add, re-run. Again, no valid characters left for #13.
- Suppose 0cc1 = 5, 0aaa = 6. Add, re-run. Too many uppercase letters!

So 0cc1 can't be 5. 0cc1 = 3, and 0aaa = 5 or 6 (i.e. 01aa). Add, re-run.

Now all five digits have been found! Good. And yet there are more than 6.5 sextillion variants. How to proceed?

I recall that "All flags are of the format of flag<text>". The answer to USB PCAP task was in this format.

Maybe this password, too, begins with "flag<" and ends with ">"?
(This also would explain why chars #1..#5 and #35 weren't mentioned in equality rules - that would be a giveaway!)

I add this info to program. And 01aa, hhhh and 0ggg are now known, so I add them too. Re-run.

Now there are only 21168000 variants left. Cool! This can be bruteforced.

Without SHA256 check there are 1050 passwords.
With SHA256 check added there's only one password:

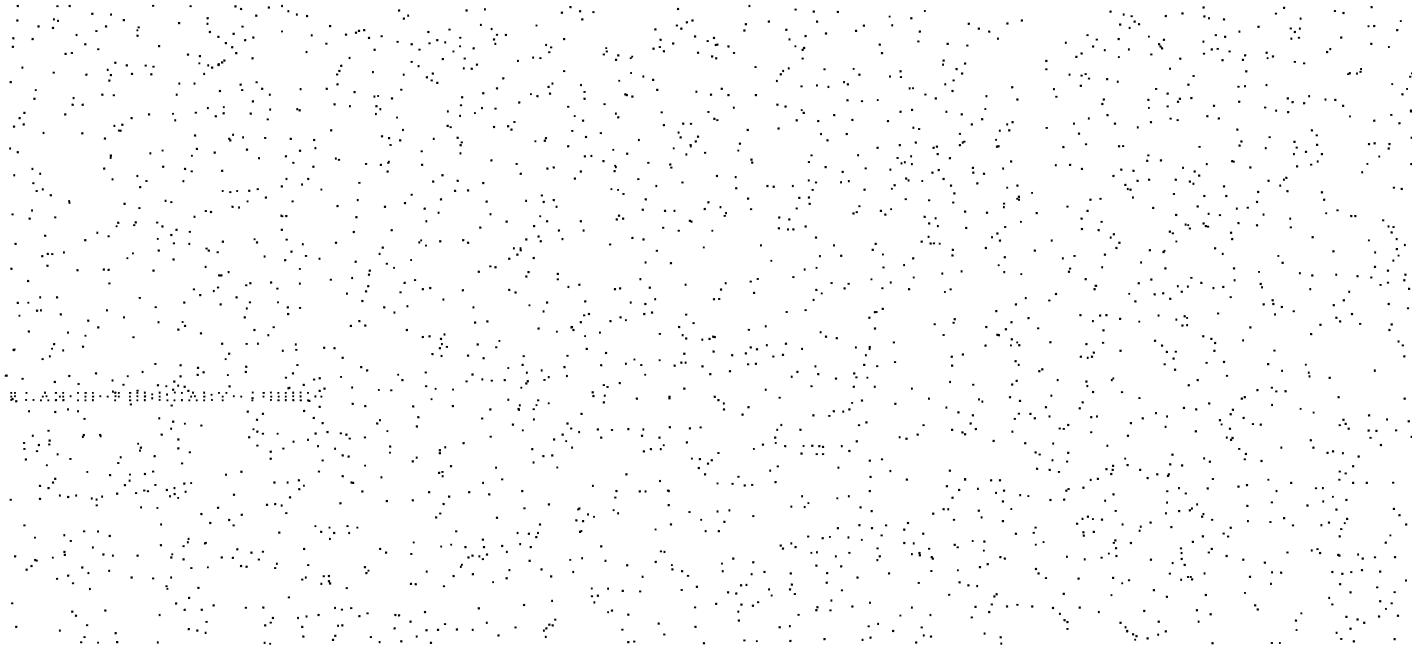
flag<UkN0Wth3pas4w0RdruL3ZaNDsODoI>

Compare the password ("You know the password rules and so do I") with the second line from "Never Gonna Give You Up" ("You know the rules and so do I").

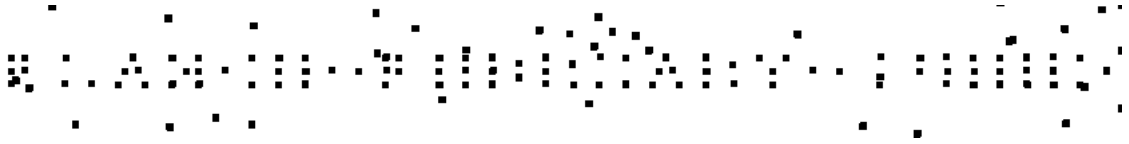
mail0009: The driver loyalty survey

mail0009-attachment.bin is a text file containing data for 9395 rides. Each ride is identified by the driver # (there are 123 drivers), starting point and ending point.

I write code to plot all points in one picture:



Wait, what's that?!



I add code to generate separate pictures for each driver. I view each picture. And this is picture #88:

Looks like "FLAG..."! However, the text isn't completely legible, so I add code to generate an SVG picture (with a line for each ride):

FLAG<6-FEBRUARY-1966>

That's Rick Astley's birthday!

mail0004: The WAV file

mail0004-attachment.bin is an audio file in RIFF WAVE format (72000 Hz, mono, 2082884 samples, 16 bits per sample, 2082884 / 72000 ≈ 28.93 seconds).

First of all, why 72000 Hz? Probably because the highest frequency in the recorded signal was 36000 Hz (see Nyquist-Shannon theorem).

I try to denoise the signal in Audacity, but to no avail.

I do some googling on infrared transmission and find this page with pictures and explanations. So now I know how clean IR signal looks like. It's square.

OK, time to create some pictures from sample data.

First, let's simply plot each sample (this is a fragment):

What if I plot the absolute difference between two neighboring samples?

Interesting. Now what if I choose some cutoff value? If Y is less than cutoff value, set it to 0; else, set it to maximum.

Looks square enough! How wide are those pulses? About 60 pixels, or 60/72000 = 0.83 microseconds. (Some are twice that width.) And data are transmitted in packets, each packet is about 1700..1800 pixels wide, or roughly 25 microseconds. But I still don't know the protocol.

I add code to remove stray dots, to locate all packets and to generate separate picture for each packet.

It turns out that there are 210 packets. I notice that pictures #0, #1 and #2 are very similar. I check other pictures. Yes, each packet is transmitted three times. I google for infrared protocol "three times" and find this page with the phrase "The Sony and RC5/6 protocols specify that messages must be sent three times". So there are, in fact, 70 packets.

I then google for "RC5 protocol" and find this page. I compare my pictures with the description - yes, looks like I have to decode RC5 packets.

So I write a decoder for Manchester coding, decode 70 packets and get 70 commands. But I have no idea what they mean, so I simply print them as digits:

71797968327479664632707665716077898384698273798583658568737950484954 62

And then I realize that each pair of digits is a code for an ASCII char! I decode the characters and get the following string:

GOOD JOB. FLAG<MYSTERIOUSAUDIO2016>

mail0002: The taximeter emulator

So I have source code for the taximeter emulator - but not for the firmware (j1.bin)!
Apparently, the task is to reverse-engineer the firmware to get the flag. I google some strings from the emulator source ("dstack", "rstack", "dsp", "rsp") and find info on J1 Forth CPU.

I take the emulator code and turn it into a disassembler.

I try to find some strings in the firmware, but there's only *SAESTROI NAFLIDE* (i.e. "*ASSERTION FAILED*"). But there are ports for writing data to 7-segment indicators. So I look for addresses of these ports (0x5010 and others) in the disassembled code and find routines that output data to indicators. Then I find some other code that pushes constants to stack and calls the aforementioned routines to display data. I spend some time converting these constants into normal digits and letters. Yes, there are strings - "StAt UnSUPP", "dAtE", "brIdGE", "AIrPrt", "tUnnEL", "rESEt", "GEt tHE JACPot" (OMG! the taximeter is also a CASINO!)... and then 'fLAG' (in the routine that starts at 0x083F)!

The routine at 0x083F takes data from 0x0016..0x0019 and writes to indicator. So I look for code that writes data to these variables and find routine at 0x081F. After some trial and error I recreate this code in C and get the flag:

flag<0D66584E7874FB66>

The tune played when the casino starts is a refrain from "Never Gonna Give You Up".

mail0005: The PIN code

mail0005-attachment.bin is a ZIP archive.
It contains two files (both are dated 2016-06-28):
SCHEMATIC.PNG - picture of a circuit diagram;
NETLIST.NAND - text file (a netlist).

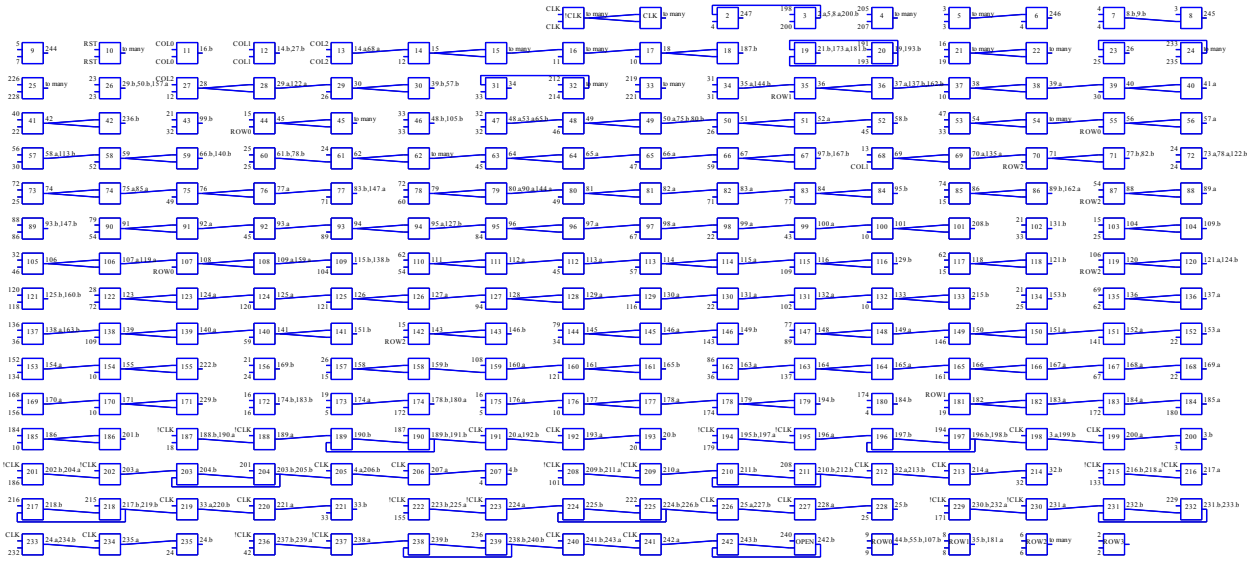
Citing Wikipedia: "In electronic design, a netlist is a description of the connectivity of an electronic circuit." There's a "custom ASIC" in the diagram.

Note the names of its inputs/outputs (ROWx/COLx/OPEN/RESET) - they appear in the netlist, so the netlist describes that custom ASIC.

The whole ASIC is built from NAND gates. Logically, NAND can be expressed as follows: dst = !(src_A & src_B).

Also note that Vcc, OSC1, OSC2 and GND are not mentioned in the netlist. Instead there's a CLK (a clock signal).

I want to visualize the scheme somehow, so I write a program that generates an SVG file from the netlist.



Note the structures in lower part of the scheme. E.g. look at four NAND gates 236, 237, 238 and 239. They make a gated D-latch, or D-flip-flop.
Each D-flip-flop has two inputs (one for data, one for clock - which enables/disables data input) and one output. (Well, actually TWO outputs, but the second one is never used in this scheme.) There are 16 D-flip-flops. And they are organized in pairs: two D-flip-flops make one master-slave flip-flop.
So there are 8 master-slave flip-flops. Each master-slave flip-flop stores 1 bit of information - the scheme contains whopping 8 bits of memory!

Data input from gate #	Gates that make the flip-flop	Data output
42	236, 237, 238, 239, 240, 241, 242, OPEN	OPEN
171	229, 230, 231, 232, 233, 234, 235, 24	24
155	222, 223, 224, 225, 226, 227, 228, 25	25
133	215, 216, 217, 218, 219, 220, 221, 33	33
101	208, 209, 210, 211, 212, 213, 214, 32	32
186	201, 202, 203, 204, 205, 206, 207, 4	4
179	194, 195, 196, 197, 198, 199, 200, 3	3
18	187, 188, 189, 190, 191, 192, 193, 20	20

CLK is needed only for the flip-flops. All this means I can abstract these flip-flops away and treat them simply as 1-bit registers (boolean variables)!

I write a simple program that can create an expression for calculating output of any gate.
Recall that the NAND gate can be expressed as dst = !(src_A and src_B).
To calculate an expression for dst I simply substitute src_A and src_B with recursively calculated expressions. The recursion stops at registers and scheme inputs (i.e. they aren't substituted).

Using this program I create expressions for all outputs (ROWs) and all gates that write data to the registers:

ROW0 = !(!(!(REG3) & !(REG4)));

ROW1 = !(!(REG3 & !(REG4)));

ROW2 = !(!(!(REG3) & REG4));

ROW3 = !(!(REG3 & REG4));

OPEN = GATE42 = !(!(!(!(!(!(!(!(REG32 & REG33)) & !(!(REG3 & !(REG4))))) & !(RESET))) & !(!(!(!(COL2 & !(COL1))) & !(!(REG24 & REG25)))))) & !(!(!(!(!(!(COL2) & !(COL1))) & !(COL0)) & !(REG20)))));

REG24 = GATE171 = !(!(!(!(!(!(!(!(!(!(!(!(REG24) & REG25)) & !(!(!(COL2) & !(COL1))))) & !(!(!(!(REG32 & REG33)) & ROW1))) & !(!(!(!(!(COL2) & COL1)) & !(!(REG24 & !(REG25))))) & !(!(!(!(!(REG32 & REG33)) & ROW1))))) & !(!(!(!(!(!(REG32 & !(REG33))) & ROW0)) & !(!(!(!(REG24 & REG25)) & !(!(!(COL2) & !(COL1))))))) & !(!(!(!(!(REG32 & !(REG33))) & ROW2)) & !(!(!(!(REG24 & !(REG25))) & !(!(!(COL2) & !(COL1))))))))) & !(!(!(!(!(!(!(!(REG24 & !(REG25))) & !(!(!(!(COL2) & !(COL1))) & ROW0)))) & !(REG32)) & !(!(!(!(!(!(!(!(REG32) & REG33)) & ROW0)) & !(!(!(!(COL2 & !(COL1))) & !(!(REG24 & REG25))))) & !(!(!(!(!(!(REG32)) & !(REG33))) & !(!(REG24 & REG25)))) & !(!(!(!(!(COL2) & !(COL1))) & ROW0))))))))) & !(!(!(!(!(!(COL2) & !(COL1))) & !(COL0)) & !(REG20)))) & !(!(!(!(!(!(COL2) & !(COL1))) & !(COL0)) & !(REG20)) & REG24)) & !(RESET));

REG25 = GATE155 = !(!(!(!(!(!(!(!(!(!(!(!(REG24) & REG25)) & !(!(!(REG32) & !(REG33))))) & !(!(!(!(COL2) & COL1)) & ROW2))) & !(!(!(!(!(REG32) & REG33)) & ROW2)) & !(!(!(!(!(REG24) & REG25)) & !(!(!(COL2) & !(COL1))))))) & !(!(!(!(!(!(REG24) & !(REG25))) & !(!(REG32 & REG33)))) & !(!(!(!(!(COL2) & !(COL1))) & ROW2))))) & !(!(!(!(!(!(!(!(!(REG32 & REG33)) & ROW1))) & !(!(!(!(!(REG32 & !(REG33))) & ROW0)) & !(!(!(!(!(COL2) & !(COL1))) & REG25))))) & !(!(!(!(!(!(!(!(REG32) & REG33)) & ROW0)) & !(!(!(!(COL2 & !(COL1))) & !(!(REG24 & REG25))))) & !(!(!(!(!(!(REG32)) & !(REG33))) & !(!(REG24 & REG25)))) & !(!(!(!(!(COL2) & !(COL1))) & ROW0))))))))) & !(!(!(!(!(!(COL2) & !(COL1))) & !(COL0)) & !(REG20)))) & !(!(!(!(!(!(COL2) & !(COL1))) & !(COL0)) & !(REG20)) & REG25)) & !(RESET));

REG33 = GATE133 = !(!(!(!(!(!(!(!(!(!(!(COL2 & !(COL1))) & !(REG24))) & !(!(!(!(REG32 & !(REG33))) & ROW2))) & !(!(!(!(REG32 & !(REG33))) & ROW2)) & !(!(!(!(REG24 & !(REG25))) & !(!(!(COL2) & !(COL1))))))))) & !(!(!(!(!(!(!(!(REG24) & !(REG25))) & !(!(!(REG32) & REG33)))) & !(!(!(!(!(COL2) & !(COL1))) & ROW0))) & !(!(!(!(!(!(REG32) & REG33)) & ROW2)) & !(!(!(!(!(REG24) & REG25)) & !(!(!(COL2) & !(COL1))))))))) & !(!(!(!(!(!(!(!(REG24) & !(REG25))) & !(!(!(REG32) & REG33)))) & !(!(!(!(!(COL2) & !(COL1))) & ROW0))) & !(!(!(!(!(!(REG32) & REG33)) & ROW0)) & !(!(!(!(COL2 & !(COL1))) & !(!(REG24 & REG25))))))) & !(!(!(!(!(!(REG32 & !(REG33))) & ROW0)) & !(!(!(!(COL2) & !(COL1))) & REG25))))))) & !(!(!(!(!(!(COL2) & !

(COL1))) & !(COL0)) & !(REG20))))) & !(!(!(!(!(!(COL2) & !(COL1))) & !(COL0)) & !(REG20)) & REG33)) & !(RESET)));

REG32 = GATE101 = !(!(!(!(!(!(!(!(!(!(!(!(REG24) & !(REG25))))) & !(!(!(REG32) & REG33)))))) & !(!(!(!(!(COL2) & !(COL1))) & ROW0))) & !(!(!(!(!(REG32) & REG33)) & ROW2)) & !(!(!(!(!(REG24) & REG25)) & !(!(!(COL2) & !(COL1)))))))) & !(!(!(!(!(!(!(!(!(!(REG24) & !(REG25))))) & !(!(!(REG32) & !(REG33)))))) & !(!(!(!(!(COL2) & COL1)) & ROW2))) & !(!(!(!(!(!(REG24) & REG25)) & !(!(!(REG32) & !(REG33)))))) & !(!(!(!(!(COL2) & COL1)) & ROW2))))))) & !(!(!(!(!(!(!(!(REG24 & !(REG25))))) & !(!(!(!(!(COL2) & !(COL1))) & ROW0)))) & !(REG32)) & !(!(!(!(!(!(!(!(REG32) & REG33)) & ROW0)) & !(!(!(!(COL2 & !(COL1))) & !(!(REG24 & REG25))))) & !(!(!(!(!(REG32) & !(REG33))) & !(!(REG24 & REG25)))) & !(!(!(!(!(COL2) & !(COL1))) & ROW0)))))))) & !(!(!(!(!(!(COL2) & !(COL1))) & !(COL0)) & !(REG20)))) & !(!(!(!(!(!(COL2) & !(COL1))) & !(COL0)) & !(REG20)) & REG32)) & !(RESET)));

REG4 = GATE186 = !(!(!(!(!(!(!(!(REG3 & !(REG4))) & !(REG20))) & !(!(!(!(!(COL2) & !(COL1))) & !(COL0)))) & !(!(!(!(REG20) & !(REG3)) & !(!(!(!(!(COL2) & !(COL1))) & !(COL0)))) & REG4)) & !(RESET)));

REG3 = GATE179 = !(!(!(!(!(!(!(!(!(COL2) & !(COL1))) & !(COL0)) & !(REG3)) & !(RESET))) & !(!(!(REG20) & !(REG3)) & !(!(!(!(!(COL2) & !(COL1))) & !(COL0))))));

REG20 = GATE18 = !(!(!(!(!(!(COL2) & !(COL1))) & !(COL0)) & !(RESET)));

Some observations:

- if RESET is 1 then all registers will became 0;
- ROWs depend only on REG3 and REG4. Whatever the values in REG3 and REG4 are, only one of four ROWs will be 1. This is an example of binary decoder;
- REG20 becames 1 when any of the COLx inputs is 1 (given enough time). If REG20 is 0 and some COLx is 1 then it means a new button has been just pressed;
- ROW3 is never used as an argument, i.e. PIN code does not contain '0', '*' or '#'.

Now I write code to generate truth tables for some of these expressions. If the function contains N arguments, the truth table will contain 2^N lines. However, there's no need to read the whole table - grep can be used to find interesting lines!

Analysis of REG4_REG3 pair truth table gives the following:

- when all COLs are 0 and REG20 == 0, REG4_REG3 increments;
- if any of the COLs is 1, REG4_REG3 does not change;
- if all COLs are 0, but REG20 == 1 (i.e. keys were just released), REG4_REG3 becames 0.

So REG4_REG3 pair is simply a 2-bit counter that switches ROWs via binary decoder. This is necessary for keyboard matrix to work.

Now let's look at truth table for OPEN. It has 9 arguments (not counting RESET - it's assumed to be 0), so it contains 512 lines. But only lines with res=1 are interesting. And there are only TWO such lines:

```
truth-open.exe | grep res=1
COL=001 ROW1=1 REG20=0 REG24_REG25_REG32_REG33=1111 res=1
COL=101 ROW1=1 REG20=0 REG24_REG25_REG32_REG33=1111 res=1
```

So the door will open when REG24, REG25, REG32 and REG33 are all 1 _and_ the last pressed digit is '6'. The last digit of the PIN code is '6'!

Initially registers REG24..REG33 contain zeros. How to get from 0000 to 1111? By entering the correct PIN code, of course!

```
truth-4regs.exe | grep OLD=0000 | grep -v NEW=0000
OLD=0000 REG20=0 COL=010 ROW=001 NEW=0001 Buttons=8
OLD=0000 REG20=0 COL=110 ROW=001 NEW=0001 Buttons=78
```

So the first digit of the PIN code must be '8'.

```
truth-4regs.exe | grep OLD=0001 | grep -v NEW=0001 | grep -v NEW=0000
OLD=0001 REG20=0 COL=001 ROW=001 NEW=0010 Buttons=9
OLD=0001 REG20=0 COL=101 ROW=001 NEW=0010 Buttons=79
```

The second digit is '9'. Note: I specify **grep -v NEW=0000** because the scheme will write 0 to all 4 registers if the wrong digit has been entered, and I want to exclude such cases.

And so on...

```
OLD=0010 REG20=0 COL=100 ROW=100 NEW=0011 Buttons=1

OLD=0011 REG20=0 COL=100 ROW=001 NEW=0100 Buttons=7

OLD=0100 REG20=0 COL=010 ROW=001 NEW=0101 Buttons=8
OLD=0100 REG20=0 COL=110 ROW=001 NEW=0101 Buttons=78

OLD=0101 REG20=0 COL=100 ROW=100 NEW=0110 Buttons=1
OLD=0101 REG20=0 COL=001 ROW=001 NEW=0010 Buttons=9       WOW!
OLD=0101 REG20=0 COL=101 ROW=001 NEW=0010 Buttons=79      WHAT'S THAT?

OLD=0110 REG20=0 COL=100 ROW=001 NEW=0111 Buttons=7

OLD=0111 REG20=0 COL=100 ROW=010 NEW=1000 Buttons=4

OLD=1000 REG20=0 COL=100 ROW=100 NEW=1001 Buttons=1

OLD=1001 REG20=0 COL=100 ROW=001 NEW=1010 Buttons=7

OLD=1010 REG20=0 COL=100 ROW=100 NEW=1011 Buttons=1

OLD=1011 REG20=0 COL=010 ROW=010 NEW=1100 Buttons=5
OLD=1011 REG20=0 COL=110 ROW=010 NEW=1100 Buttons=45

OLD=1100 REG20=0 COL=100 ROW=100 NEW=1101 Buttons=1

OLD=1101 REG20=0 COL=100 ROW=100 NEW=1110 Buttons=1

OLD=1110 REG20=0 COL=001 ROW=100 NEW=1111 Buttons=3
OLD=1110 REG20=0 COL=101 ROW=100 NEW=1111 Buttons=13
```

So the PIN code is '891781741715113'... oh, and recall that the last digit must be '6'. Also note that there's a surprise - we may repeat first four digits as many times as we want before entering the rest! For example, '8917891789178917817417151136'.

flag<8917817417151136>

Appendix A. The PCAP decoder

pcap-decoder.c:

```
#include "stdio.h"
#include "string.h"
#include "stdlib.h"

typedef struct _mousedata
{
  int btn, dx, dy, x, y;
} _mousedata;

int cntMouse = 0;
struct _mousedata mdata[10000];
char s[1024];

unsigned char prevKeys[8] = { 0, 0, 0, 0, 0, 0, 0, 0 };

void
HexStrToBytes (char *hex, char *outbuf)
{
  char *q;
  while (hex && hex[0] && hex[1])
    {
      char x[3];
      x[0] = hex[0];
      x[1] = hex[1];
      x[2] = 0;
      outbuf[0] = strtoul (x, &q, 16);
      hex += 2;
      outbuf++;
    }
}

void
Keyboard (char *z)
{
  char x[16];
  unsigned char c;
  char *q;
  int i;
  unsigned long int v;
  unsigned char b[8];

  HexStrToBytes (z, &b[0]);

  c = b[2];
  if (b[3])
    c = b[3];

  if (prevKeys[3] == c)
    c = 0;

  if (c)
    {
      if (8 == b[0])
        printf ("{Win}+");
      if (4 == b[0])
        printf ("{Alt}+");

      switch (c)
        {
        case 0x38:
          printf ("-");
          break;
        case 0x32:
          printf ("'");
          break;
        case 0x37:
          printf ((b[0] == 2) ? ":" : ".");
          break;
        case 0x28:
          printf ("{Enter} ");
          break;
        case 0x2B:
          printf ("{Tab} ");
          break;
        case 0x2A:
          printf ("{Backspace} ");
          break;
        case 0x2C:
          printf (" ");
          break;
        case 0x36:
          printf (",");
          break;

        default:
          if (c >= 4 && c <= 29)
            printf ("%c", c + ((2 == b[0]) ? 'A' : 'a') - 4);
          else
            printf ("{%.2X}", c);
          break;
        };
    };

  memcpy (prevKeys, b, 8);
}
```

```
void
Mouse (char *z)
{
  signed char b[4];

  HexStrToBytes (z, &b[0]);

  mdata[cntMouse].btn = b[0];
  mdata[cntMouse].dx = b[1];
  mdata[cntMouse].dy = b[2];
  cntMouse++;
  if (cntMouse >= (sizeof (mdata) / sizeof (mdata[0])))
    {
      exit (1);
    }

}

#define PICWIDTH 1800
#define PICHEIGHT 500

char pic[PICHEIGHT][PICWIDTH];

void
ClearPic (void)
{
  memset (pic, '0', sizeof (pic));
};

void
SetPoint (int x, int y)
{
  pic[y][x] = '1';
};

void
DrawSquare (int x, int y)
{
  int a, b;
  for (a = -3; a <= +3; a++)
    for (b = -3; b <= +3; b++)
      SetPoint (x + a, y + b);

}

void
SavePic (char *fname)
{
  FILE *f;

  f = fopen (fname, "wt");
  if (!f)
    return;

  fprintf (f, "P1\n#\n%d %d\n", PICWIDTH, PICHEIGHT);
  fwrite (&pic[0][0], PICWIDTH, PICHEIGHT, f);
  fclose (f);
}

void
SavePic2 (int i)
{
  char fname[16];
  sprintf (fname, "mouse%.3d.pbm", i);
  SavePic (fname);
}


void
DrawAllVirtualKeys (void)
{
  int i;
  for (i = 544; i <= 4770; i++)
    {
      if (mdata[i].btn)
        DrawSquare (mdata[i].x, mdata[i].y);
    }
}


void
main (void)
{
  FILE *f;
  int i, j, x, y;
  int k;
  char *p, *q;

  static const char *inputFile = "exported.txt";

  f = fopen (inputFile, "rt");
  if (!f)
    {
      printf ("Can't open %s\n", inputFile);
      exit (1);
    }

  while (fgets (s, sizeof (s), f))
    {
      k = strlen (s);
      if (k)
        if (10 == s[k - 1])
          s[--k] = 0;
      if (!k || !strstr (s, "Leftover"))
```

```
        continue;
      p = strstr (s, ": ");
      if (!p)
        continue;
      p += 2;
      k = strlen (p);

      if (k >= 16)
        Keyboard (p);
      else
        Mouse (p);
    }

// draw all mouse movements and clicks in one picture

  x = 1200;
  y = 300;
  ClearPic ();
  for (i = 0; i < cntMouse; i++)
    {
      x += mdata[i].dx;
      y += mdata[i].dy;
      mdata[i].x = x;
      mdata[i].y = y;
      SetPoint (x, y);
      if (mdata[i].btn)
        DrawSquare (x, y);
    }
  SavePic ("mouse-all.pbm");

// draw each mouse movement in separate picture, but with all virtual keypresses
  ClearPic ();
  for (i = 544, j = 0; i <= 4770; i++)
    {
      SetPoint (mdata[i].x, mdata[i].y);

      if (mdata[i].btn)
        {
          DrawAllVirtualKeys ();
          SavePic2 (j);
          j++;
          ClearPic ();
        };
    };
}
```

Appendix B. The password bruteforcer

password.c:

```
#include "stdio.h"
#include "stdlib.h"
#include "string.h"

#define UINT unsigned int

typedef enum CharClass { UPPER = 0, LOWER = 1, DIGIT = 2, MAGIC = 3 } CharClass;

enum { MARK = -1 };

const UINT numCharOfClass[4] = {11, 17, 5, 2 };
#define numUpper numCharOfClass[ UPPER ]
#define numLower numCharOfClass[ LOWER ]
#define numDigits numCharOfClass[ DIGIT ]
#define numMagic numCharOfClass[ MAGIC ]

enum { pwdLen = 35 };

const UINT bitsSetInAllChars[8] = {16,17,19,12,18,24,28,0};
//arrays are indexed from 1 to simplify code
UINT isUniq[1+pwdLen];
char pwd[1+pwdLen+1];
char mask[1+pwdLen][9];
char valid[1+pwdLen][128]; // valid variants for each char
UINT nVar[1+pwdLen];
UINT minBits[1+pwdLen];
UINT maxBits[1+pwdLen];
double total;
UINT figCnt=1;

const char power2[8] = { 1, 2, 4, 8, 16, 32, 64, 128 };

void SetValidChars( UINT posN, char * z ) { strcpy( valid[posN], z ); nVar[posN] = strlen( valid[posN] ); }
UINT BitsSet( char c ) { UINT i, j; for( i=0, j=0; i<8; i++ ) j += !!(c&power2[i]); return j; }
void CharToBits( char c, char * p ) { int i; p[8]=0; for( i=0; i<8; i++ ) p[i] = '0' + !!( c&power2[7-i] ); }

#define IsUpper(x) IsCharOfClass(x,UPPER)
#define IsLower(x) IsCharOfClass(x,LOWER)
#define IsDigit(x) IsCharOfClass(x,DIGIT)

UINT IsCharOfClass( char c, CharClass k )
{
switch(k)
        {
        case UPPER: return (c >= 'A' && c <= 'Z');
        case LOWER: return (c >= 'a' && c <= 'z');
        case DIGIT: return (c >= '0' && c <= '9');
        case MAGIC: return !(IsUpper(c)|IsLower(c)|IsDigit(c));
        }
return 0;
}

UINT PosHasClass( UINT posN, CharClass k )
{
UINT i, j;
if( !nVar[posN] )
        {
        printf( "Error: no valid characters left for position %d\n", posN );
        exit(1);
        }
for( i=0, j=0; i < nVar[posN]; i++ ) j += IsCharOfClass( valid[posN][i], k ); return ( j == nVar[posN] );
}

UINT CountPosOfClass( CharClass k )  { UINT i, j; for( i=1, j=0; i<=pwdLen; i++ ) j += PosHasClass(i,k); return j; }

UINT CheckMask( char c, UINT posN )
{
int i;
char b[9];
CharToBits( c, b );

if( 0 == memcmp( &mask[posN][0], &mask[posN][4], 4 )) // nibbles in the mask are equal
        if( 0 != memcmp( &b[0], &b[4], 4 )) // but in this char, they aren't
                return 0;

for( i=0; i<8; i++ )
        switch( mask[posN][i] ) { case '0': case '1': if( mask[posN][i] != b[i] ) return 0; }

return 1; // ok
}

void AddValidChar( UINT posN, char c )
{
char x[2];
x[0] = c; x[1] = 0;
if( !CheckMask( c, posN )) return;
if( !strstr( valid[posN], x )) { strcat( valid[posN], x ); nVar[posN] = strlen( valid[posN] ); }
}

void RemoveMarked( UINT posN )
{
UINT i, j;
char x[128];
for( i=0, j=0; i < nVar[posN]; i++ ) if( MARK != valid[posN][i] ) x[j++] = valid[posN][i];
x[j] = 0; strcpy( valid[posN], x ); nVar[posN] = j;
}
```

```
void ExcludeCharFromValid( UINT posN, char c )
{
char *p;
p = strchr( valid[posN], c ); if(p) { *p = MARK; RemoveMarked( posN ); }
}

void ListAll( void )
{
UINT i, j;
printf( "\n\nFigure #%d.\n", figCnt ); figCnt++;
for( i=1; i <= pwdLen; i++ )
        {
        printf( "char #%.2d - %s - ", i, mask[i] );
        if( minBits[i] == maxBits[i] )
                printf( "%d bits set     ", minBits[i] );
        else
                printf( "%d to %d bits set", minBits[i], maxBits[i] );
        printf( " - %2d variants: [%s]\n", nVar[i], valid[i] );
        }

printf( "Uppercase: %d from %d found\n", CountPosOfClass(UPPER), numUpper);
printf( "Lowercase: %d from %d found\n", CountPosOfClass(LOWER), numLower);
printf( "Digits: %d from %d found\n", CountPosOfClass(DIGIT), numDigits);
printf( "Magics: %d from %d found\n", CountPosOfClass(MAGIC), numMagic);

total = 1.0; for( i=1; i <= pwdLen; i++ ) total *= (double)nVar[i];
printf("Total: %5.5f passwords to check\n", total );
}

void ReplaceNibbles( char * pattern, char * repl )
{
UINT i;
char *p;
for( i=1;i <= pwdLen;i++)
        {
        p=&mask[i][0]; if( 0 == memcmp( p, pattern, 4 )) memcpy( p, repl, 4 );
        p+=4; if( 0 == memcmp( p, pattern, 4 )) memcpy( p, repl, 4 );
        }
}

void UpdateMask( UINT posN )
{
UINT i, j, k;
char unimask[9];
k=nVar[posN];
if(!k)return;

// calc combined mask for the position (from all valid chars)
CharToBits( valid[posN][0], unimask );
for( i=1; i < k; i++ )
        {
        char curmask[9];
        CharToBits( valid[posN][i], curmask );

        for( j=0; j < 8; j++ )
                {
                if( unimask[j] == '?' )
                        continue;
                if( unimask[j] != curmask[j] )
                        unimask[j] = '?';
                }
        }

// now replace the old mask with the new
for( i=0; i < 8; i++ ) if( '?' == mask[posN][i] ) mask[posN][i] = unimask[i];
}

void UpdateAll( void )
{
UINT i, j, k, N, err=0;

for( N=0; N < 2; N++ ) {

// check again number of bits
for( i=1; i <= pwdLen; i++ )
        {
        for( j=0; j < nVar[i]; j++ )
                {
                k=BitsSet( valid[i][j] );
                if( !(k >= minBits[i] && k <= maxBits[i]) )
                        {
                        valid[i][j] = MARK;
                        }
                else
                if( 0 == CheckMask( valid[i][j], i ))
                        {
                        valid[i][j] = MARK;
                        }
                }
        RemoveMarked( i );
        }

// if char is known exactly and is unique, it cannot be in other positions
for( i=1; i <= pwdLen; i++ )
        if( (1==nVar[i]) && isUniq[i])
                for( j=1; j < pwdLen; j++ ) if( j != i ) ExcludeCharFromValid( j, valid[i][0] );

{
CharClass x;
for( x=UPPER; x <= MAGIC; x++ )
        {
        i = CountPosOfClass(x);
        if( i > numCharOfClass[x] ) { err = 1; };
```

```
        if( i == numCharOfClass[x] )
                {
                for( i=1; i <= pwdLen; i++ )
                        {
                        if( !PosHasClass(i,x))
                                {
                                for( j=0; j < nVar[i]; j++ ) if( IsCharOfClass( valid[i][j], x )) valid[i][j] = MARK;
                                RemoveMarked( i );
                                }
                        }
                }
        }
}

for( i=1; i <= pwdLen; i++ ) UpdateMask(i);
for( i=1; i <= pwdLen; i++ ) if( 1 == nVar[i] ) minBits[i] = maxBits[i] = BitsSet( valid[i][0] );
};

ListAll();
if(err)
        {
        printf( "Error: too many chars of certain class(es)\n" );
        exit(1);
        }
}



UINT ValidatePassword( const char * const p )
{
UINT i, j, bitK[8], kU, kL, kD, kO;

kU=kL=kD=kO=0;
memset( bitK,0,sizeof(bitK));
for( i=1; i <= pwdLen; i++ )
        {
        char c=p[i];
        if( IsUpper(c)) kU++;
        else if( IsLower(c)) kL++;
        else if( IsDigit(c)) kD++;
        else kO++;
        for( j=0; j < 8; j++ )
                bitK[j] += !!(c&power2[j]);
        }
if( !(kU == numUpper && kL == numLower && kD == numDigits && kO == numMagic) ) return 0; // check number of letters/digits/etc.
for( i=0; i < 8; i++ ) if( bitK[i] != bitsSetInAllChars[i] ) return 0; // check number of bit set in all chars

// check equality
if( ( p[9]  != p[19] )||( p[13] != p[25] )||( p[29] != p[32] )||( p[8]  != p[28] )||
    ( p[3]  != p[15] )||( p[3]  != p[27] )||( p[16] != p[30] )
) return 0;

// check unique chars
{
char k[256];
memset(k,0,sizeof(k));
for( i=1;i <= pwdLen;i++ )  k[ p[i] ]++;
for( i=1;i <= pwdLen;i++ ) if( isUniq[i] && (k[ p[i] ] != 1)) return 0;
}

// validate SHA256

fprintf( stderr, "." );
{
FILE *f;
char hash[1+64+1];

f=fopen( "tmp", "wb" );
if(!f)return 0;
fwrite(&p[1], 1, pwdLen, f );
fclose(f);
system( "sha256sum tmp > hash" );

f=fopen( "hash", "rb" );
if(!f)return 0;
fread( &hash[1], 1, 64, f );
fclose(f);

if( p[1]    != hash[1]  ) return 0;
if( hash[2] != hash[63] ) return 0;
if( hash[3] != hash[64] ) return 0;
}

printf( "\nVALID PASSWORD: %s\n", &p[1]);
return 1;
}

char currpwd[1+pwdLen+1];
int cnt=0;
int cntValidPwds=0;

void Gen( int pos )
{
UINT i, j, k;
if( pos > pwdLen )
        {
        currpwd[pos]=0;
        if( ValidatePassword(&currpwd[0]))
                {cntValidPwds++; printf( "%.3d" "%c" ": %s\n", (int)(100.0*(double)cnt/total), 0x25, &currpwd[1] );}
        cnt++;
        return;
        }

for( i=0; i < nVar[pos]; i++ )
```

```
        {
        currpwd[pos] = valid[pos][i];
        Gen(pos+1);
        }
}


void H(int i, char * p) { memcpy( &mask[i][0], p, 4 ); }
void L(int i, char * p) { memcpy( &mask[i][4], p, 4 ); }

void Init( void )
{
UINT i, j, k;
char *p;

memset(pwd,0,sizeof(pwd));
memset(valid,0,sizeof(valid));

// mark unique chars
{
const int uniqChars[]={ 1, 2, 4, 5, 6, 7, 10, 11, 12, 14, 17, 18, 20, 21, 22, 23, 24, 26, 31, 33, 34, 35 };
for( i=0; i < (sizeof(uniqChars)/sizeof(uniqChars[0])); i++ )
        isUniq[ uniqChars[i] ]=1;
};

// initialize less/more statements
for( i=1; i <= pwdLen; i++ )
        {
        switch( "?MMLMMMMM.MMLMLLML!.LLMMLMMLM.MM.!LM"[i] )
                {
                case '.': minBits[i] = maxBits[i] = 2; break;
                case 'L': minBits[i] = maxBits[i] = 3; break;
                case 'M': minBits[i] = 4; maxBits[i] = 5; break;
                case '!': minBits[i] = maxBits[i] = 6; break;
                }
        }

// set masks
for( i=1; i <= pwdLen; i++ ) strcpy( mask[i], "0???????" );

// apply nibble/char equality rules
p="0aaa"; H(6,p); L(23,p); H(10,p); L(6,p); H(20,p); H(26,p);
p="0bbb"; H(8,p); L(11,p); L(17,p); L(21,p); L(29,p); L(32,p); H(24,p); H(28,p); H(29,p); H(31,p); H(32,p); H(34,p);
p="0ccc"; H(9,p); L(13,p); L(16,p); L(25,p); L(30,p); H(13,p); H(17,p); H(19,p); H(25,p);
p="0ddd"; H(11,p); L(10,p); L(18,p); H(14,p); H(16,p); H(18,p); H(22,p); H(23,p); H(30,p);
p="eeee"; L(9,p); L(19,p); // no high nibbles, so high bit is unknown
p="ffff"; L(8,p); L(28,p); // the same
p="0ggg"; H(3,p); H(15,p); H(27,p);
p="hhhh"; L(3,p); L(15,p); L(27,p); // the same

SetValidChars( 1, "0123456789ABCDEFabcdef" ); // char #1 is hex

for( i=2;i <= pwdLen; i++ )
        {
        for(j=32; j < 128; j++ ) // only printable ASCII chars
                {
                k=BitsSet(j);
                if( k >= minBits[i] && k <= maxBits[i] )
                        {
                        if( IsCharOfClass(j,MAGIC))
                                {
                                if (j < 0x40)
                                        AddValidChar(i,j);
                                }
                        else
                                AddValidChar(i,j);
                        }
                }
        }
}


#define Say(x) printf( "\n" x "\n" )

void main()
{
UINT i, j, k;

Say("The beginning.");
Init();
ListAll();

Say("Set 0ddd to 0111 (i.e. 7)");
ReplaceNibbles( "0ddd", "0111" ); UpdateAll();

Say("Set 0bbb to 0100 (i.e. 4)");
ReplaceNibbles( "0bbb", "0100" ); UpdateAll();

Say("Set 0ccc to 0cc1 (i.e. 3 or 5)");
Say("Set eeee to 0000" );
ReplaceNibbles( "0ccc", "0cc1" ); ReplaceNibbles( "eeee", "0000" ); UpdateAll();

Say("0aaa is either 3, 5 or 6");
for( i=1; i <= pwdLen; i++ )
        {
        if( 0 == memcmp( mask[i], "0aaa", 4 ))
                {
                for( j=0; j < nVar[i]; j++ )
                        {
                        k=valid[i][j] / 16;
                        switch(k){ case 3: case 5: case 6: break; default: valid[i][j] = MARK; };
                        }
                RemoveMarked( i );
                }
```

```
        if( 0 == memcmp( &mask[i][4], "0aaa", 4 ))
                {
                for( j=0; j < nVar[i]; j++ )
                        {
                        k=valid[i][j]&15;
                        switch(k){ case 3: case 5: case 6: break; default: valid[i][j] = MARK; };
                        }
                RemoveMarked( i );
                }
        }
UpdateAll();

// suppose 0aaa = 3, 0cc1 = 3 -- no valid chars left for #13
// ReplaceNibbles( "0cc1", "0011" ); ReplaceNibbles( "0aaa", "0011" ); UpdateAll();

// suppose 0aaa = 3, 0cc1 = 5 -- too many uppercase chars (12)
// ReplaceNibbles( "0cc1", "0101" ); ReplaceNibbles( "0aaa", "0011" ); UpdateAll();

// suppose 0aaa = 5, 0cc1 = 5 -- no valid chars left for #13
// ReplaceNibbles( "0aaa", "0101" ); ReplaceNibbles( "0cc1", "0101" ); UpdateAll();

// suppose 0aaa = 6, 0cc1 = 5 -- too many uppercase chars (12)
// ReplaceNibbles( "0aaa", "0110" ); ReplaceNibbles( "0cc1", "0101" ); UpdateAll();

// so 0cc1 = 3, 0aaa = 5 or 6 (i.e. 01aa)
Say("Set 0cc1 to 3 and 0aaa to 01aa");
ReplaceNibbles( "0cc1", "0011" ); ReplaceNibbles( "0aaa", "01aa" ); UpdateAll();

// now all digits have been found!
Say("Set the beginning to 'flag<' and last char to '>'");
Say("Also set 0aaa, hhhh and 0ggg");

SetValidChars( 1,  "f" );
SetValidChars( 2,  "l" );
SetValidChars( 3,  "a" ); SetValidChars( 15, "a" ); SetValidChars( 27, "a" );
SetValidChars( 4,  "g" );
SetValidChars( 5,  "<" );
SetValidChars( 35, ">" );

ReplaceNibbles( "01aa", "0101" ); ReplaceNibbles( "hhhh", "0001" );
ReplaceNibbles( "0ggg", "0110" ); UpdateAll();

// now bruteforce password
Say("Bruteforcing password");
Gen(1);
printf( "\nFound %d valid password(s)\n", cntValidPwds );
Say("The end.");
}
```

Appendix C. Code for the drivers survey task

drivers.c:

```
#include "stdio.h"
#include "stdlib.h"
#include "mem.h"
#include "string.h"

typedef struct _ride
{
unsigned int driver;
double x1, y1, x2, y2;
}
_ride;

unsigned int cntRides = 0;
struct _ride ride[10000];

void LoadDrivers( char * fname )
{
FILE *f;
unsigned int i;
char s[128];

f=fopen( fname, "rt" );
if(!f)
        {
        printf( "Can't open %s\n", fname );
        exit(1);
        }

i=0;
while(fgets(s,sizeof(s),f))
        {
        if( strstr( s, "driver" ))
                continue;

        sscanf( s, "%d\t%lf,%lf\t%lf,%lf", &ride[i].driver, &ride[i].x1, &ride[i].y1, &ride[i].x2, &ride[i].y2 );
        i++;

        if( i >= (sizeof(ride)/sizeof(ride[0])) )
                {
                printf( "Too much data in %s\n", fname );
                exit(1);
                }
        }
cntRides=i;
};


char pic[1800][4300];

void ClearPic( void )
{
memset( pic, '0', sizeof(pic));
}

void SetPoint( double x, double y )
{
int line, col;
int i, j;
line = 1800 - ( (int)( x * 100.0 ) - 3000);
col = 4300 - ((int)(y * -100.0) - 8000);
for( i=-2; i < 3; i++ ) for( j=-2; j < 3; j++ ) pic[line+i][col+j] = '1';
}

void SavePic( char * fname )
{
FILE *f;
int line, col;

f=fopen( fname, "wt" );
if(!f)
        {
        fprintf( stderr, "Can't write %s\n", fname );
        return;
        }
fprintf( stderr, "Saving %s\n", fname );
fprintf( f, "P1\n#\n4300 1800\n" );
fwrite( &pic[0][0], 4300, 1800, f );
fclose(f);
}

int X( double x )
{
return 40 - ((int)(x*100.0) - 3700);
}

int Y( double y )
{
return (12230+(int)(y*100.0));
}

void GenSvg88( char * fname )
{
FILE *f;
unsigned int i;

f=fopen( fname, "wt" );
```

```
if(!f)
        {
        printf( "Can't create %s\n", fname );
        return;
        }

fprintf( f, "\x3C" "?xml version=\"1.0\"?" "\x3E" );
fprintf( f, "\x3C" "svg width=\"840\" height=\"30\" viewBox=\"0 0 840 30\" xmlns=\"http://www.w3.org/2000/svg\"" "\x3E" "\n" );

for( i=0; i < cntRides; i++ )
        {
        if( 88 == ride[i].driver )
                {
fprintf( f, "\x3c" "line x1=\"%d\" y1=\"%d\" x2=\"%d\" y2=\"%d\" stroke=\"black\" stroke-width=\"2\"/" "\x3e" "\n",
Y(ride[i].y1), X(ride[i].x1), Y(ride[i].y2), X(ride[i].x2) );
                }
        };

fprintf( f, "\x3c" "/svg" "\x3e" );
fclose(f);
}

void main()
{
unsigned int i, j, drv;

LoadDrivers( "mail0009-attachment.bin" );

// create picture with all starting and ending points

ClearPic();

for( i=0; i < cntRides; i++ )
        {
        SetPoint( ride[i].x1, ride[i].y1 );
        SetPoint( ride[i].x2, ride[i].y2 );
        };

SavePic( "drv-all.pbm" );

// now create per-driver pictures
for( drv=1; drv <= 123; drv++ )
        {
        char s[16];
        ClearPic();

        for( i=0; i < cntRides; i++ )
                {
                if( ride[i].driver == drv )
                        {
                        SetPoint( ride[i].x1, ride[i].y1 );
                        SetPoint( ride[i].x2, ride[i].y2 );
                        }
                };

        sprintf( s, "drv%.3d.pbm", drv );
        SavePic(s);
        };

// now generate the flag picture
GenSvg88( "flag.svg" );
}
```

Appendix D. The audio decoder

audio.c:

```
#include "stdio.h"
#include "stdlib.h"
#include "mem.h"

#define NUMSAMPLES 2082884

unsigned char m[44 + 2 * NUMSAMPLES];   // 44 bytes for RIFF WAVE header
unsigned char n[NUMSAMPLES];
unsigned long int nsize;
unsigned long int fsize;
unsigned long int begoff;
char cmdbuf[128];
unsigned long int cntCmd = 0;
int picNum = 0;
int savedPos[300];
unsigned long int cntSaved = 0;

#define PICWIDTH 2000           // about 27.7 microseconds - that enough to display one packet
#define PICHEIGHT 256

#define CUTOFF_VALUE 25

unsigned char pic[PICHEIGHT][PICWIDTH];

void
CleanPic (void)
{
  memset (pic, '0', sizeof (pic));
}

void
SetPoint (int x, int y)
{
  if (x >= 0 && x <= (PICWIDTH - 1) && y >= 0 && y <= (PICHEIGHT - 1))
    {
      pic[(PICHEIGHT - 1) - y][x] = '1';
    }
}

void
SaveNamedPic (char *fname)
{
  FILE *f;
  int i, j;

  fprintf (stderr, "saving %s\n", fname);
  f = fopen (fname, "wt");
  if (!f)
    return;
  fprintf (f, "P1\n#\n%d %d\n", PICWIDTH, PICHEIGHT);
  for (i = 0; i < PICHEIGHT; i++)
    fwrite (&pic[i][0], 1, PICWIDTH, f);
  fclose (f);
}

void
SavePic (int num)
{
  FILE *f;
  int i, j;
  char fname[32];
  sprintf (fname, "packet%.3d.pbm", num);
  SaveNamedPic (fname);
}

void
CreatePacketPic (int mpos, int length)
{
  int i, j;

  fprintf (stdout, "pic %.4d width = %.5d\n", picNum, length);
  fflush (stdout);

  CleanPic ();

  for (i = 0; i < length; i++)
    {
      SetPoint (i, n[mpos + i] ? 128 : 0);
    }

/*
for( i=(100-82); i < length; i+=128 )
        {
        for( j=0; j < 255; j++ )
                SetPoint( i, j );
        }
*/

  SavePic (picNum);
  picNum++;
}

void
DecodeManchester (int mpos)     // int length )
{
  int i, j, k;
```

```
  char bits[14 + 1];
  char *q;

  if (!cntCmd)
    memset (cmdbuf, 0, sizeof (cmdbuf));

  // printf ("Decoded: ");
  for (k = 0; k < 14; k++)
    {
      int sumA, sumB;
      sumA = 0;
      for (i = 0; i < 64; i++)
        {
          if (n[mpos + 18 + k * 128 + i])
            sumA++;
        }
      sumB = 0;
      for (i = 0; i < 64; i++)
        {
          if (n[mpos + 18 + 64 + k * 128 + i])
            sumB++;
        }
      bits[k] = (sumA < sumB) ? '1' : '0';
    }
  bits[14] = 0;
  // printf ("S=%.2s T=%.1s A=%.5s CMD=%.6s\n", &bits[0], &bits[2], &bits[3], &bits[8]);
  i = strtoul (bits, &q, 2);
  {
    int t, addr, cmd;
    cmd = i & 63;
    i /= 64;
    addr = i & 31;
    i /= 32;
    t = i & 1;
    printf ("Decoded: T=%d A=%d CMD=%d\n", t, addr, cmd);
    cmdbuf[cntCmd] = cmd;
    cntCmd++;
  };
}

signed short int
GetWord (unsigned char *x)
{
  signed short int r;
  r = x[1];
  r *= 256;
  r |= x[0];
  return r;
}

void
LoadWAV (void)
{
  FILE *f;

  static const char inputFile[] = "mail0004-attachment.bin";

  f = fopen (inputFile, "rb");
  if (!f)
    {
      printf ("can't open %s\n", inputFile);
      exit (1);
    }

  fseek (f, 0, SEEK_END);
  fsize = ftell (f);
  fseek (f, 0, SEEK_SET);

  if (fsize > sizeof (m))
    {
      printf ("file is too big\n");
      exit (1);
    }

  memset (n, 0, sizeof (n));

  fread (m, 1, fsize, f);
  fclose (f);

  begoff = 0;
  if (0 == memcmp (m, "RIFF", 4))
    begoff = 0x2C;
}

void
RemoveStrayDots (void)
{
  unsigned long int i, j;

// pass 1
  for (i = 0; i < nsize; i++)
    {
      int sum;
      // window 50 pixels
      sum = 0;
      for (j = 0; j < 50; j++)
        sum += n[i + j];
      if (sum < 20)
        n[i] = 0;
    }

// pass 2
  for (i = 5; i < nsize; i++)
    {
```

```
      int sum;
      sum = 0;
      for (j = 0; j <= 10; j++)
        {
          sum = sum + n[(i - 5) + j];
        };

      if (n[i] && sum < 4)
        n[i] = 0;
      else if (!n[i] && sum > 6)
        n[i] = 1;
    }

}

void
FindAllPackets (void)
{
  unsigned long int i;
  int state = 0, sum = 0;
  for (i = 0; i < nsize; i++)
    {
      if (n[i] == state)
        {
          sum++;
        }
      else
        {
          if (sum > 4000)
            {
              printf ("@%.8lX: sum=%d\n", i, sum);
              savedPos[cntSaved] = i;
              cntSaved++;
            }
          state = !state;
          sum = 1;
          //markPos = i;
        }
    }

  if (sum > 4000)
    {
      printf ("@%.8lX: sum=%d\n", i, sum);
      savedPos[cntSaved] = i;
      cntSaved++;
    }
}

void
main (void)
{
  FILE *f;
  unsigned long int i, j;
  signed short int t;

  LoadWAV ();

  for (i = 1, j = 0; i < NUMSAMPLES; i++)
    {
      signed long xx, prev;
      xx = GetWord (&m[begoff + i * 2]) + 32768;
      prev = GetWord (&m[begoff + (i - 1) * 2]) + 32768;
      n[j] = abs ((xx - prev) / 256) > CUTOFF_VALUE ? 1 : 0;
      j++;
    }

  nsize = j;

  RemoveStrayDots ();
  FindAllPackets ();

  picNum = 0;
  for (i = 1; i < cntSaved; i++)
    {
      CreatePacketPic (savedPos[i - 1] - 100, savedPos[i] - savedPos[i - 1]);
      if (1 == (i % 3))         // we only need to decode each third packet
        DecodeManchester (savedPos[i - 1] - 100);       //, savedPos[i]-savedPos[i-1] );
    }

  printf ("The commands: ");
  for (i = 0; i < cntCmd; i++)
    printf ("%c", cmdbuf[i] + '0');
  printf ("\n");

  for (i = 0; i < cntCmd; i += 2)
    printf ("%c", cmdbuf[i] * 10 + cmdbuf[i + 1]);

}
```

Appendix E. The taximeter flag calculator

taxiflag.c:

```
#include "stdio.h"

unsigned char m[16384];

void
main (void)
{
  FILE *f;
  unsigned short int v[4], i, j;

  static const char inputFile[] = "j1.bin";

  f = fopen (inputFile, "rb");
  if (!f)
    {
      printf ("can't open %s\n", inputFile);
      return;
    }

  fread (&m[0], 1, sizeof (m), f);
  fclose (f);

  for (i = 0; i < 4; i++)
    v[i] = 31337;               // :)

  for (i = 0x800; i < 0x1c00; i += 2)
    {
      j = m[i];
      j *= 256;
      j |= m[i + 1];

      v[0] += j;
      v[1] += v[0];
      v[2] += v[1];
      v[3] += v[2];
    }

  printf ("flag" "\x3c");
  for (i = 0; i < 4; i++)
    {
      char digit[4];
      for (j = 0; j < 4; j++)
        {
          digit[j] = v[i] & 15;
          v[i] /= 16;
        }
      for (j = 0; j < 4; j++)
        printf ("%X", digit[j]);
    }
  printf ("\x3e" "\n");
}
```

Appendix F. The gate calculator

gatecalc.c:

```
#include "stdio.h"
#include "stdlib.h"
#include "string.h"

enum
{
  OPEN = 243, ROW0 = 244, ROW1 = 245, ROW2 = 246, ROW3 = 247,   // outputs
  CLK = 248, RESET = 249, COL0 = 250, COL1 = 251, COL2 = 252    // inputs
};


typedef struct _gate
{
  int dst, srcA, srcB;
} _gate;

#define NUMGATES 248
#define NUMWIRES 253

_gate gate[NUMGATES];

#define NUMKEYWORDS 11

const char keyword[NUMKEYWORDS][8] = {
  "NAND", "OPEN", "ROW0", "ROW1", "ROW2", "ROW3", "CLK", "RESET", "COL0",
  "COL1", "COL2"
};

void
LoadNetlist (void)
{
  FILE *f;
  unsigned int i, j, gateIdx;
  char *p;
  int state;
  char s[50];

  static const char inputFile[] = "NETLIST.NAND";

  memset (gate, 0, sizeof (gate));

  f = fopen (inputFile, "rt");
  if (!f)
    {
      printf ("Can't open %s\n", inputFile);
      exit (1);
    }

  while (fgets (s, sizeof (s), f))
    {
      if (strstr (s, "#"))
        continue;
      p = s;

      state = 0;
      while (p)
        {
          j = 0xFF;
          for (i = 0; i < NUMKEYWORDS; i++)
            {
              if (0 == strncmp (p, keyword[i], strlen (keyword[i])))
                {
                  j = i;
                  break;
                }
            }

          if (j != 0)           // ignore keyword 0 (NAND)
            {

              j = (0xFF == j) ? atoi (p) : j + 242;

              if ((j >= NUMWIRES) || ((0 == state) && (j >= NUMGATES)))
                {
                  printf ("Error in netlist\n");
                  exit (1);
                }

              switch (state)
                {
                case 0:
                  gateIdx = j;
                  gate[gateIdx].dst = j;
                  state++;
                  break;
                case 1:
                  gate[gateIdx].srcA = j;
                  state++;
                  break;
                case 2:
                  gate[gateIdx].srcB = j;
                  state++;
                  break;
                }

              if (state > 2)
```

```
              break;

          };

         p = strstr (p, " ");
         if (p)
           p++;
       };

    }
  fclose (f);
}


char retbuf[2048];

int level = 0;

char *
Calc (int gateNum)
{
  char tmp[2048];
  char tmp2[2048];

  level++;

  if (gateNum >= 0xBB && gateNum < OPEN)        // these gates are part of flip-flops
    {
      // calculating flip-flops using recursion is not a good idea :)
      printf ("%d? No way!\n", gateNum);
      exit (1);
    }

  do
    {

      if (gateNum >= CLK)
        {
          // do not calc inputs
          strcpy (retbuf, keyword[gateNum - 242]);
          break;
        };

      switch (gateNum)
        {
          // registers
        case 3:
        case 4:
        case 0x14:
        case 0x18:
        case 0x19:
        case 0x20:
        case 0x21:
        case OPEN:
          sprintf (retbuf, "REG%d", gateNum);
          continue;             // i.e. break
        case ROW0:
        case ROW1:
        case ROW2:
        case ROW3:
          if (level > 1)
            {
              sprintf (retbuf, "ROW%d", gateNum - ROW0);
              continue;
            }
          break;
        };

      if (gate[gateNum].srcA == gate[gateNum].srcB)
        {
          sprintf (tmp, "!(%s)", Calc (gate[gateNum].srcA));
          strcpy (retbuf, tmp);
        }
      else
        {
          strcpy (tmp, Calc (gate[gateNum].srcA));
          strcpy (tmp2, Calc (gate[gateNum].srcB));
          sprintf (retbuf, "!(%s & %s)", tmp, tmp2);
        };

    }
  while (0);

  level--;

  return retbuf;
}



void
main (int argc, char **argv)
{
  int i;
  if (argc < 2)
    {
      printf ("\nusage: gatecalc GATE# (e.g.  gatecalc 25  or  gatecalc OPEN)\n\n");
      return;
    }

  LoadNetlist ();

  for (i = 1; i < (sizeof (keyword) / sizeof (keyword[0])); i++)
    {
```

```
      if (0 == strcmp (&argv[1][0], &keyword[i][0]))
        {
          printf ("%s = %s\n", keyword[i], Calc (i + 242));
          return;
        }
    }

  i = atoi (argv[1]);
  printf ("GATE%d = %s\n", i, Calc (i));
}
```

Appendix G. Truth table generators

truth-open.c:

```
#include "stdio.h"

#define COL0 (!!(i&1))
#define COL1 (!!(i&2))
#define COL2 (!!(i&4))
#define RESET 0
#define REG20 (!!(i&8))
#define ROW1 (!!(i&16))
#define REG24 (!!(i&32))
#define REG25 (!!(i&64))
#define REG32 (!!(i&128))
#define REG33 (!!(i&256))


void
main (void)
{
  int i, GATE42;

  for (i = 0; i < 512; i++)
    {
      GATE42 = !(!(!(!(!(!(!(!(!(!(REG32 & REG33)) & ROW1)) &
                          !(RESET))) & !(!(!(!(COL2 & !(COL1))) &
                                          !(!(REG24 & REG25)))))) &
                    !(!(!(!(!(!(COL2) & !(COL1))) & !(COL0)) & !(REG20)))));

      printf
        ("COL=%d%d%d ROW1=%d REG20=%d REG24_REG25_REG32_REG33=%d%d%d%d res=%d\n",
         COL0, COL1, COL2, ROW1, REG20, REG24, REG25, REG32, REG33, GATE42);
    };
}
```

truth-4regs.c:

```
#include "stdio.h"

#define REG24 (!!(i&1))
#define REG25 (!!(i&2))
#define REG33 (!!(i&4))
#define REG32 (!!(i&8))
#define RESET 0
#define COL0 (!!(i&16))
#define COL1 (!!(i&32))
#define COL2 (!!(i&64))
#define REG20 (!!(i&128))
#define ROW0 (!!(i&256))
#define ROW1 (!!(i&512))
#define ROW2 (!!(i&1024))

void
main (void)
{
  int newREG24, newREG25, newREG33, newREG32;
  int i;

  for (i = 0; i < 2048; i++)
    {

      if (ROW0 + ROW1 + ROW2 > 1) continue; // impossible - skip

        newREG24 = !(!(!(!(!(!(!(!(!(!(!(!(!(REG24) & REG25)) & !(!(!(COL2) & !(COL1))))) &
!(!(!(!(REG32 & REG33)) & ROW1))) & !(!(!(!(!(!(COL2) & COL1)) & !(!(REG24 & !(REG25))))) & !(!(!(!(REG32 & REG33)) &
ROW1))))) & !(!(!(!(!(!(!(REG32 & !(REG33))) & ROW0)) & !(!(!(!(REG24 & REG25)) & !(!(!(COL2) & !(COL1)))))) &
!(!(!(!(!(REG32 & !(REG33))) & ROW2)) & !(!(!(!(REG24 & !(REG25))) & !(!(!(COL2) & !(COL1)))))))))) & !(!(!(!(!(!(!(REG24 &
!(REG25))) & !(!(!(!(!(COL2) & !(COL1))) & ROW0)))) & !(REG32)) & !(!(!(!(!(!(!(!(REG32) & REG33)) & ROW0)) & !(!(!(!(COL2 &
!(COL1))) & !(!(REG24 & REG25))))) & !(!(!(!(!(!(REG32) & !(REG33))) & !(!(REG24 & REG25)))) & !(!(!(!(!(COL2) &
!(COL1))) & ROW0)))))))) & !(!(!(!(!(!(COL2) & !(COL1))) & !(COL0)) & !(REG20)))) & !(!(!(!(!(!(COL2) & !(COL1))) &
!(COL0)) & !(REG20)) & REG24)) & !(RESET)));
        newREG25 = !(!(!(!(!(!(!(!(!(!(!(!(REG24) & REG25)) & !(!(!(REG32) & !(REG33))))) &
!(!(!(!(!(COL2) & COL1)) & ROW2))) & !(!(!(!(!(!(REG32) & REG33)) & ROW2)) & !(!(!(!(!(REG24) & REG25)) &
!(!(!(COL2) & !(COL1))))))) & !(!(!(!(!(!(REG24) & !(REG25))) & !(!(REG32 & REG33)))) & !(!(!(!(!(COL2) &
!(COL1))) & ROW2))))) & !(!(!(!(!(!(!(!(!(!(COL2) & COL1)) & !(!(REG24 & !(REG25))))) & !(!(!(!(REG32 & REG33)) &
ROW1))) & !(!(!(!(!(REG32 & !(REG33))) & ROW0)) & !(!(!(!(!(COL2) & !(COL1))) & REG25))))) & !(!(!(!(!(!(!(REG32) &
REG33)) & ROW0)) & !(!(!(!(COL2 & !(COL1))) & !(!(REG24 & REG25))))) & !(!(!(!(!(!(REG32) & !(REG33))) & !(!(REG24 &
REG25)))) & !(!(!(!(!(COL2) & !(COL1))) & ROW0)))))))) & !(!(!(!(!(!(COL2) & !(COL1))) & !(COL0)) & !(REG20)))) &
!(!(!(!(!(!(COL2) & !(COL1))) & !(COL0)) & !(REG20)) & REG25)) & !(RESET)));
        newREG33 = !(!(!(!(!(!(!(!(!(!(!(!(COL2 & !(COL1))) & !(REG24))) & !(!(!(!(REG32 & !(REG33))) &
ROW2))) & !(!(!(!(!(REG32 & !(REG33))) & ROW2)) & !(!(!(!(REG24 & !(REG25))) & !(!(!(COL2) & !(COL1)))))))) &
!(!(!(!(!(!(!(!(REG24) & !(REG25))) & !(!(!(REG32) & REG33)))) & !(!(!(!(!(COL2) & !(COL1))) & ROW0))) &
!(!(!(!(!(!(!(REG32) & REG33)) & ROW2)) & !(!(!(!(!(REG24) & REG25)) & !(!(!(COL2) & !(COL1)))))))))) &
!(!(!(!(!(!(!(!(!(!(REG24 & !(REG25))) & !(!(!(REG32) & REG33)))) & !(!(!(!(!(COL2) & !(COL1))) & ROW0))) &
!(!(!(!(!(!(REG32) & REG33)) & ROW0)) & !(!(!(!(!(COL2 & !(COL1))) & !(!(REG24 & REG25)))))))) & !(!(!(!(!(REG32 &
!(REG33))) & ROW0)) & !(!(!(!(!(COL2) & !(COL1))) & REG25)))))) & !(!(!(!(!(!(COL2) & !(COL1))) & !(COL0)) &
!(REG20)))) & !(!(!(!(!(!(COL2) & !(COL1))) & !(COL0)) & !(REG20)) & REG33)) & !(RESET)));
        newREG32 = !(!(!(!(!(!(!(!(!(!(!(!(!(REG24) & !(REG25))) & !(!(!(REG32) & REG33)))) &
!(!(!(!(!(COL2) & !(COL1))) & ROW0))) & !(!(!(!(!(!(REG32) & REG33)) & ROW2)) & !(!(!(!(!(REG24) & REG25))
& !(!(!(COL2) & !(COL1))))))) & !(!(!(!(!(!(!(REG24) & !(REG25))) & !(!(!(REG32) & !(REG33))))) &
!(!(!(!(!(COL2) & COL1)) & ROW2))) & !(!(!(!(!(!(REG24) & REG25)) & !(!(!(REG32) & !(REG33))))) &
!(!(!(!(!(COL2) & COL1)) & ROW2))))))) & !(!(!(!(!(!(!(REG24 & !(REG25))) & !(!(!(!(!(COL2) &
!(COL1))) & ROW0)))) & !(REG32)) & !(!(!(!(!(!(!(!(REG32) & REG33)) & ROW0)) & !(!(!(!(COL2 & !(COL1))) &
!(!(REG24 & REG25))))) & !(!(!(!(!(!(REG32) & !(REG33))) & !(!(REG24 & REG25)))) & !(!(!(!(!(COL2) &
!(COL1))) & ROW0)))))))) & !(!(!(!(!(!(COL2) & !(COL1))) & !(COL0)) & !(REG20)))) &
!(!(!(!(!(!(COL2) & !(COL1))) & !(COL0)) & !(REG20)) & REG32)) & !(RESET)));

      printf
```

```
        ("OLD=%d%d%d%d REG20=%d COL=%d%d%d ROW=%d%d%d NEW=%d%d%d%d Buttons=",
         REG24, REG25, REG33, REG32, REG20, COL0, COL1, COL2, ROW0, ROW1,
         ROW2, newREG24, newREG25, newREG33, newREG32);

      if (ROW0 && COL0) printf ("1");
      if (ROW0 && COL1) printf ("2");
      if (ROW0 && COL2) printf ("3");
      if (ROW1 && COL0) printf ("4");
      if (ROW1 && COL1) printf ("5");
      if (ROW1 && COL2) printf ("6");
      if (ROW2 && COL0) printf ("7");
      if (ROW2 && COL1) printf ("8");
      if (ROW2 && COL2) printf ("9");

      printf ("\n");

    };

}
```