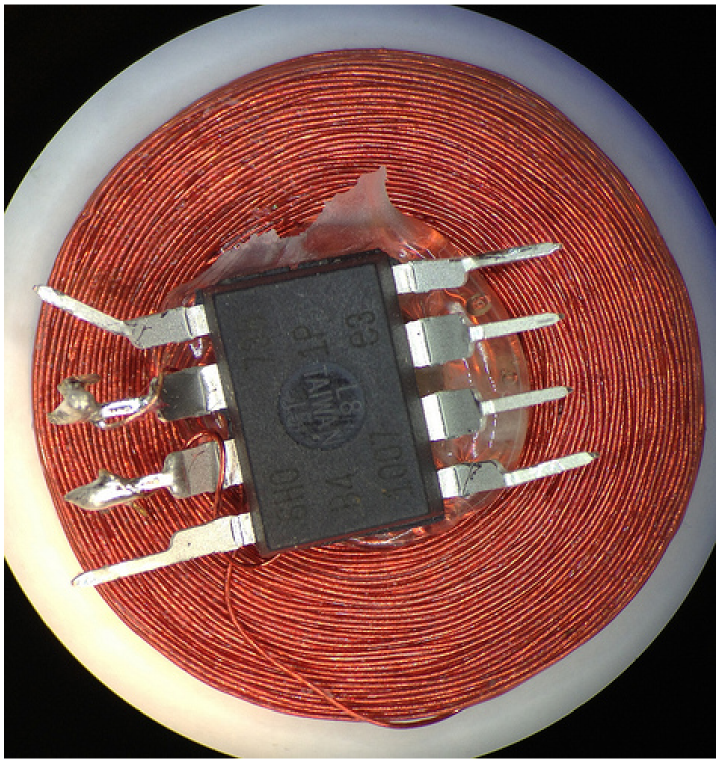
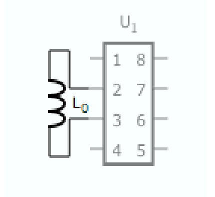
AVR RFID — Trammell Hudson's Projects

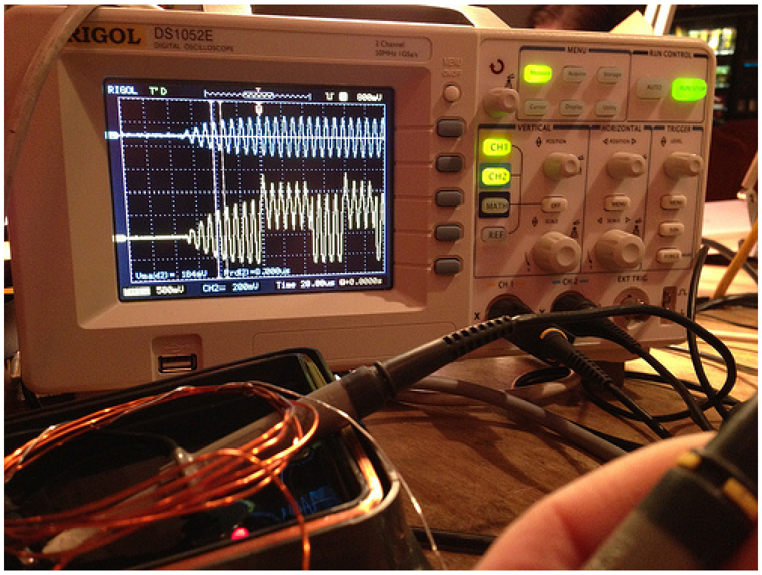
From Trammell Hudson's Projects



I was inspired by Beth’s avrfid.S (http://scanlime.org/2010/06/avrfid-1-1-firmware/) project to try to build a replacement for the multiple HID Prox cards that I carry for work. Her design is simultaneously a technical tour-de-force and an example of how badly we can abuse the Atmel chips. Here is the entire schematic:



There is no connection to power and ground: the chip is powered through leakage current from the input pins. The AC waveform is fed directly into the pins: the internal protection diodes rectify it. During negative parts of the wave the silicon die’s inherent capacitance maintains state. The CPU clock is driven by the AC as well and depends on the ability of the coil to drive more current than the chip when DDRB is configured to pull the pins to the same potential. It's truly amazing that this works at all.



The firmware she wrote in macro assembler is easy to understand and straight-foward, but filled the entire 8 KB flash on the ATTiny85 when compiled for HID Prox cards. Unlike the CW modulated EM41xx cards that just load the coil for thirty RF cycles to send a baseband one and don't load the coil to send a baseband zero, the HID cards use Frequency Shift Keying (FSK) modulation. In FSK a baseband zero is sent by cycling the load on the coil for 50 cycles at a frequency of 4 RF cycles, and a baseband one is sent by cycling the load every 5 RF cycles. Beth's code loads the coil by setting the two bits in DDRB to 1 while holding PORTB at 0, which places a short across the coil by putting both ends at the same potential.

While it turns out that my dream of automatically selecting the right RFID card doesn't work, here are details of how to build your own HID compatible RFID devices and some overview of the hand-tuned assembly necessary to fit the RFID timing.

Contents

- 1 Card format
- 2 Instruction timing
- 3 The state machine
- 4 Flashing the RFID
- 5 Testing the RFID
- 6 Updates

Card format

It turns out that the HID cards always send the same total number of bits regardless of the card format -- the bit stream always starts with seven 0s, a 1, some number of 0s to pad the overall length to 45 bits, and another 1 to indicate the of the header. (Thanks to Technologia Incognita (http://techinc.nl/) for this insight) For a sample 26-bit and 35-bit card, the bits consist of a header, the card type, one or two parity bits, some number of bits for the facility code, finally the bits of the ID number and another parity bit. The card reader doesn't output the first two portions of this, so you need to add the padding yourself.

26-bit Prox card format:

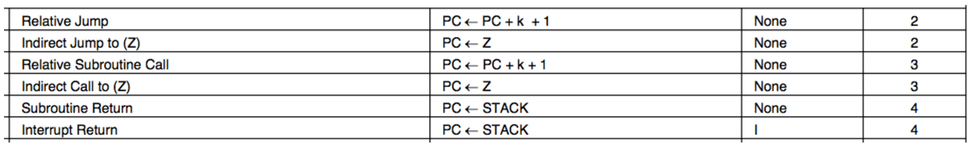
```
00000001-00000000001-0-00101010-0101110110001010-0
|-Head-| |-Padding-| P |Faclty| |--- ID Code --| P
|--- Not output ---| |---- Output by reader -----|
```

35-bit "Corporate 1000" card format:

```
00000001-01-11-100001001000-00011100001100100101-0
|-Head-| LL PP |-Facility-| |--- ID Code ------| P
|No output| |--------- Output by reader ---------|
```

The equivalent lengths of the bitstreams is clear. This likely is done for market segmentation to allow HID to sell the different card lengths at different prices, but save money by only having one actual card format.

Instruction timing



Relative Jump	PC ← PC + k + 1	None	2
Indirect Jump to (Z)	PC ← Z	None	2
Relative Subroutine Call	PC ← PC + k + 1	None	3
Indirect Call to (Z)	PC ← Z	None	3
Subroutine Return	PC ← STACK	None	4
Interrupt Return	PC ← STACK	I	4

One issue with programming HID Prox compatible cards is that the AVR’s `RCALL` and `RET` instructions are quite slow -- 3 and 4 clocks respectively (http://www.atmel.com/images/doc0856.pdf) -- so making a function call and returning from it requires seven clocks and would cause errors in the RF waveform. To get around this, Beth expanded all of the code inline to produce a single function that bit-bangs the coil loading with `NOP`'s between each cycle. The 19-bit manufacturer ID (`0x0801`), 8-bit faciity code, 16-bit unique ID and two parity bits, all Manchester encoded, required 80 instructions per bit for a total of 3700 instructions out of the Tiny85's maximum of 4096. Supporting 34-bit cards would not be possible with this design, much less multiple card IDs!

While `RCALL`/`RET` are out of the question, I noticed that `IJMP` is only 2 clocks. This means that the CPU can do an indirect jump to the value in the 16-bit `Z` register in enough time to be ready for the next FSK cycle. If we know where to go, that is... The `LPM` instruction takes three cycles to read a byte from flash into a register, which just barely fits during the idle time during a FSK baseband one. Loading the Z register for `LPM` takes at least two clocks (since it is really the two 8-bit registers `r31:r30`), which means the `pgm_read_word()` macro in `avr/progmem.h` (http://www.nongnu.org/avr-libc/user-manual/group__avr__pgmspace.html) won't work. While the rest of the firmware is in mostly normal C, I resorted to writing assembly to interleave the coil toggling with the operations to determine the next output state and make the appropriate jump. If you want to follow along, the source for the RFID firmware is available in `rfid/avrfid2.c` (https://bitbucket.org/hudson/rfid/src/0ef4c3636f0a/avrfid2.c?at=default).

The state machine

The card IDs are stored in a flash memory character array where the ASCII characters encode the states. `2` is the state that sends the HID header, state `3` is to jump back to the start, and states `0` and `1` send a zero or one. For one of my test cards, the array definition looks like this (with `__used__` to indicate to gcc that this array must be present, even if it does not see any usage of it):

```
static const char hid_bits[]
PROGMEM __attribute__((__used__)) =
{
	HID_HEADER
	"0000000100000000001" // HID code 0x0801 for 26-bit cards
```

```
        "0"             // first party bit
        "00101010"             // Site code 42
        "0101110110001010"     // ID 23946
        "0"                    // second parity bit
        HID_RESET
};
```

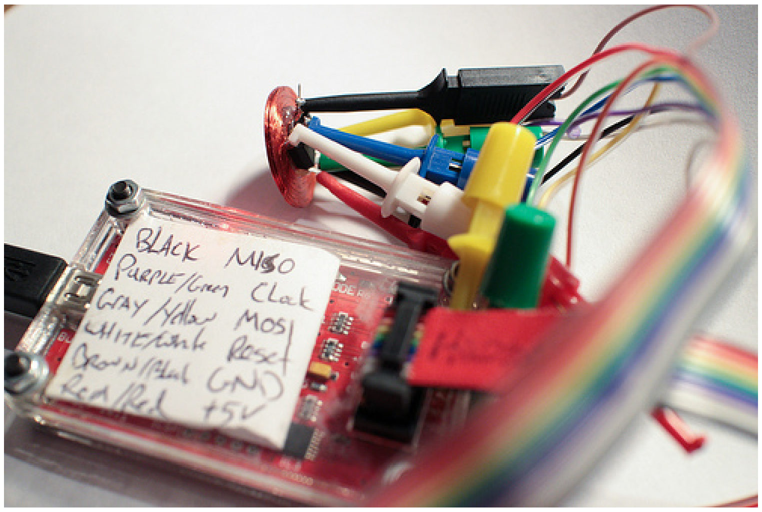
Or for 35-bit Corporate Cards:

```
static const char hid_bits[]
PROGMEM __attribute__((__used__)) =
{
        HID_HEADER
        "0000000101" // 35 bit cards use header 0x05, with 10 bits
// read from card 11-100001001000-00011100001100100101-0
        "11" // 2 parity bits
        "100001001000" // fc=2120, 12 bits
        "00011100001100100101" // id=115493, 20 bits
        "0" // 1 parity bit
        HID_RESET
};
```

The code to send a baseband one looks roughly like this, with the FSK generation interleaved with reading the next state from the `hid_bits[]` array and then looking up the function to call from the `state_handlers[]`. At the end of the function, the `Z` register holds the function pointer to be called next. The `toggle` macro takes two clocks and turns the load on the coil if it is currently unloaded, or turns it off if it is currently on. This leaves three clocks to do before the next toggle. Most of the instructions are single cycle, except for `LPM` which is three clocks, and `RJMP .+0` which is a two clock NOP.

```
baseband1:
        toggle /* 5 */
                                ldi r30, lo8(hid_bits)
                                ldi r31, hi8(hid_bits)
                                add r30, r15 // bit_num
        toggle /* 10 */
                                lpm r24, Z // next_bit = lpm(hid_bits[bitnum])
        toggle /* 15 */
                                ldi r30, lo8(state_handlers)
                                ldi r31, hi8(state_handlers)
                                nop
        toggle /* 20 */
                                subi r24, '0'
                                lsl r24
                                add r30, r24 // z = &state_handlers[next_bit - '0']
        toggle /* 25 */
                                lpm r24, Z+
        toggle /* 30 */
                                lpm r31, Z
        toggle /* 35 */
                                mov r30, r24 // z = lpm(z);
                                rjmp .+0
        toggle /* 40 */
                                inc r15 // bit_num++
                                rjmp .+0
        toggle /* 45 */
                                nop // Nothing to do!
                                rjmp .+0
        toggle /* 50 */
                                /* Leave last delay slot free */
```

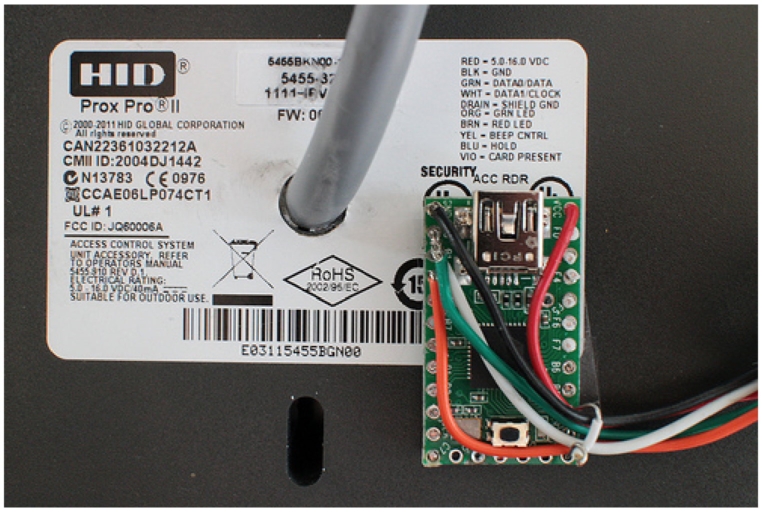
Flashing the RFID



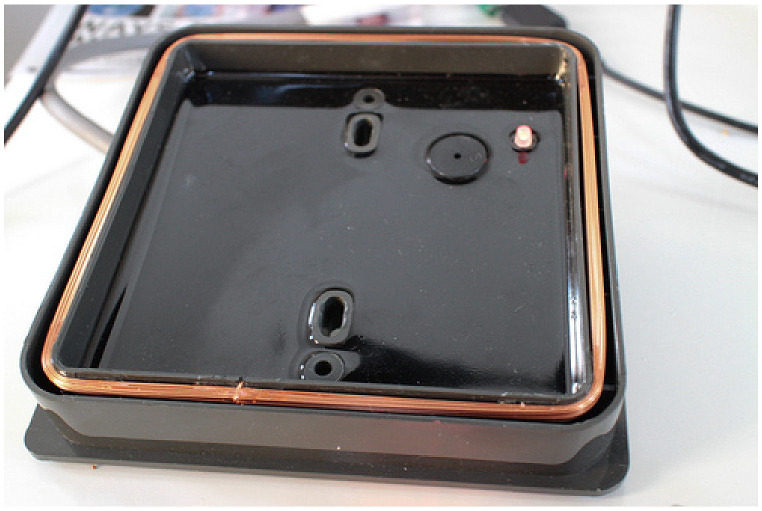
Once the fuse bits have been configured to use the RF waveform as the clock source the chip will no longer be programable with a normal AVR ISP. One option is to use Dangerous Prototypes' buspirate (http://dangerousprototypes.com/docs/Bus_Pirate), which can provide a "recovery" clock during programming. Unfortunately it didn't work for me with the current release of avrdude; I had to make the following patches to the avrdude/buspirate.c (http://dangerousprototypes.com/forum/viewtopic.php?f=41&t=4922) source to get it to work. The pinout to connect the Tiny85 to the buspirate:

```
                        +-------+
 White/white    Reset  |1   v  8| Vcc    Red
 Blue/Blue      Xtal1  |2      7| SCK    Purple/green
                Xtal2  |3      6| MISO   Black/Black
 Black          Gnd    |4      5| MOSI   Gray/Yellow
                        +-------+
```

Testing the RFID

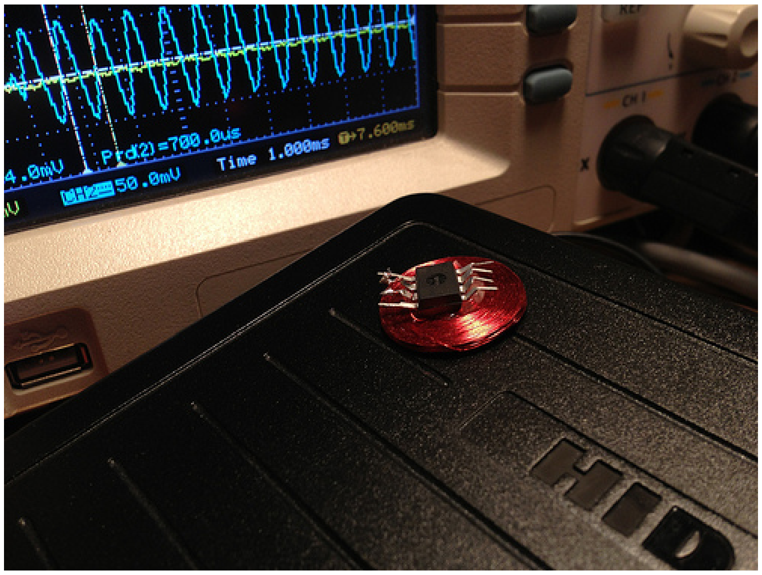


To better test my design, I needed a way to read the HID cards. Unfortunately the commonly available RFID readers from sparkfun (https://www.sparkfun.com/categories/144) and adafruit (http://www.adafruit.com/category/55) only read EM4xxx cards, not the HID Proxcard modulation. So I acquired a surplus HID ProxPro II and built a small adapter with a Teensy 2.0. The output of the device is an odd format -- Wiegand transmits zeros on one wire and ones on the other -- so it required adapting to connect to a normal computer. My `rfid/hid-rfid-reader.c` (https://bitbucket.org/hudson/rfid/src/tip/hid-rfid-reader.c?at=default) program translates this into an easy to parse serial output. Each output line is a complete read of a card in ASCII formatted binary.



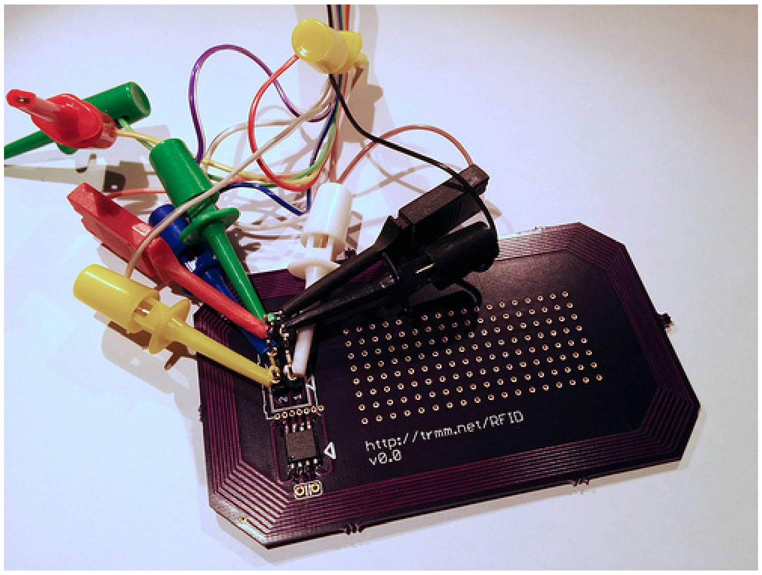
If you wanted to try to modify the firmware of the reader, you're out of luck. There are no user servicable parts inside -- the entirety of the body is filled with potting compound epoxy. If you're interested in lower level details and compatibility with multiple card formats, the proxmark3 (http://code.google.com/p/proxmark3/wiki/HomePage) is a better device for further hacking.

Unfortunately it turns out that my dream of cycling through multiple IDs won't work with the way HID Prox readers report their results. They seem to query the card continuously until they receive two identical reads of the ID, then they stop probing until the reader detects that the card has left the field. This means that cycling continuously between IDs will cause it to never read any of them, and cycling between two repeats of each cards will cause it to stop reading after the first duplicate read succeeds. There are enough pins on the Tiny85 (or even the microscopic Tiny10) and plenty of space left if I wanted to add a switch to select between different IDs, but this would require me to remove the device from my pocket, which would somewhat defeat the purpose of the multipass.



Oh well... It was a mostly successful and very fun project, and an exciting challenge to fit everything in such a small device. Thanks so much to Beth for the source code and documentation on the various protocols.

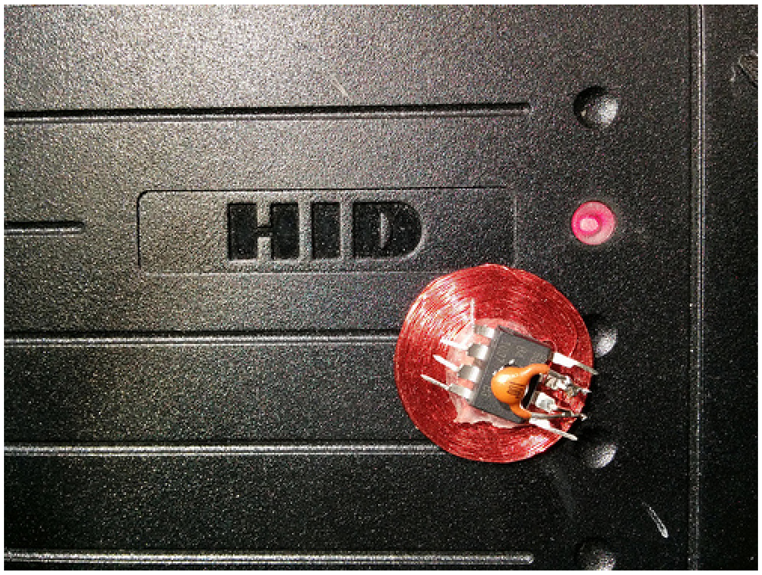
Updates



I've tried to make a PCB mount version that uses a long trace as the coil. It generates plenty of voltage, but for some reason that AVR isn't able to signal back to the reader. I tried to make the avr fit inside the board, but messed up the instructions to OSHpark so it requires some cable hacking to program.

The C version original described only supported classic 26-bit cards, but I recently needed to support the "secure" HID Corporate 1000 35-bit format. The above text has been updated to reflect these changes.

Based on Daniel Smith's writeup on the format (http://www.pagemac.com/azure/data_formats.php) and some digging around, I figured out that the `MFG_CODE` for this format is 10-bits long with the value `0x005`. He also pointed out that the 26-bit firmware had the wrong code -- it is not the 20-bit code `0x01002`, but is instead the 19-bit code `0x0801` and the bottom bit is part of the parity computation for the card id. If you're using a HID branded Proxcard reader, the value that it outputs is the entire data portion, including all of the parity bits, but does not include the `MFC_CODE` part.



I've updated my firmware (https://bitbucket.org/hudson/rfid/commits/1636e2adcbc5ef663dc7ae3 with these changes and it works great. Emulating a 35-bit card takes 846 bytes of flash (nine more than the 26-bit cards since the state machine stores one bit per byte), so it might be possible to port this to the attiny10. I've also found that the tags work much better with a small capacitor across the two clock pins, as shown in the above photo.

(Originally posted to the NYCR Blog, "AVRFID Mutlipass" (http://www.nycresistor.com/2012/12/27/rfid-multipass/)) and "extending the firmware to support 35-bit Corporate 1000 cards" (http://www.nycresistor.com/2013/11/18/avrfid-35-bit-support/)

Categories: Repost | Hacks | AVR | RFID | 2012
