Visitekaarduino

Random number generator Ver. 1.00

Robin Pynckels
Filip Pynckels

September 9, 2020

Documents:

http://users.telenet.be/pynckels/2020-3-VisitekaArduino-Random-number-generator_ver_1-0.pdf

http://users.telenet.be/pynckels/2019-4-VisitekaArduino_ver_1-1.pdf

Table of contents

List of figures

Introduction

Some time ago, Filip's business cards were changed to reflect the job he has. The Visitekaarduino (see *Figure 1*) was born. Since people are asking to see some useful applications of this card, we added a proof of concept of a fire-smoke-temperature alarm in the article explaining the Visitekaarduino (Dutch for business card Arduino or something alike, see link on first page).

This document elaborates on a second project based on the Visitekaarduino. A hardware random number generator shield that can be mounted on the Visitekaarduino.

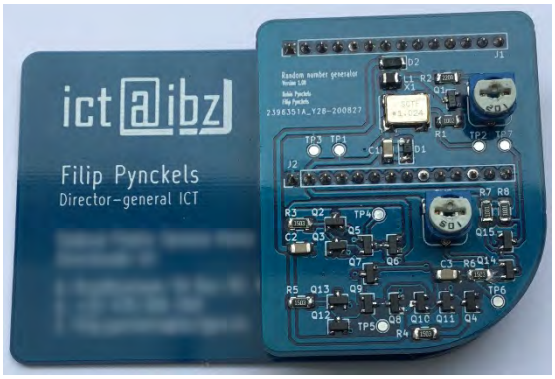


FIGURE 1 - VISITEKAARDUINO RANDOM NUMBER SHIELD

Why, we hear you say, would you do all the effort of making hardware to generate random numbers when so much software random number generators exist. For the fun of course. But there is also a better reason. Lately, more and more applications are network centered. These applications need a good encryption method to communicate data across the Internet. But what is a good encryption method?

There are several good encryption algorithms available. However, they all count on the availability of sequences of random bits. When generating random numbers by use of software algorithms, the generated numbers are pseudo random numbers, and not random numbers. When generating pseudo random numbers, each number depends on the already generated number(s). Also, there is a cycle that repeats itself time after time. This cycle can be large, but it repeats itself anyway. The software generation and the existence of a cycle make it possible to guess what the next number will be when one knows where in the cycle the previous number is generated. And there goes the good encryption…

So, what is needed is the generation of true random bit sequences of a chosen length that are not predictable. And then arise the questions: how to generate true random bit sequences, and how to check if an entropy generator effectively generates true random bit sequences.

An answer to these questions is given in this paper. We hope that you enjoy reading it, and that it can provide you with a reliable method to generate your own random bit sequences. All information in this document is kept under the MIT open software / hardware license. The project is a joined effort of Pynckels & Pynckels, being Robin & Filip.

1. Visitekaarduino

To be as transparent as possible, *Figure 2* repeats the schematics of the Visitekaarduino. Please note that some security provisions are omitted such as an inductor and reverse biased diode on the AREF port and a capacitor on the reset port. This does not make this Arduino version less flexible; it just requires you not to feed it voltage peeks or powering it with fluctuating DC.

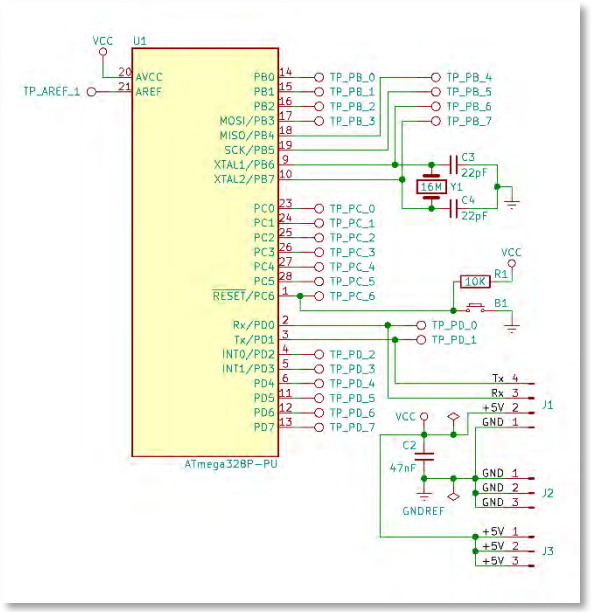


FIGURE 2 - VISITEKAARDUINO SCHEMATICS

The used processor is an ATmega328P, which is about the most used Atmel processor these days. To give our Visitekaarduino a bit of juice, there is place on the PCB to add a 16Mhz crystal with two accompanying capacitors. You can opt not to add this crystal and to use the ports B6 and B7 for other goals. Such a configuration limits the processor to 4Mhz clock speed.

Pin C6 is pulled up to Vcc to prevent the processor from going into reset mode due to a floating pin. To permit you to reset the processor, a reset button can be added. Pulling pin C6 to Gnd resets the processor.

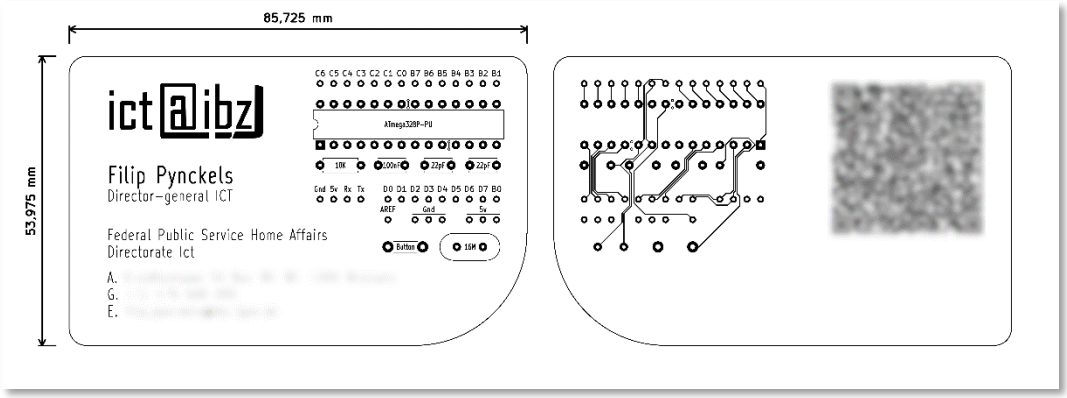


FIGURE 3 - VISITEKAARDUINO PRINTED CIRCUIT BOARD (PCB)

4 of the 6 FTDI pins are broken out, so to permit this little Arduino to communicate with the outside world (Rx on port D0 and Tx on port D1), and to get its power through a variety of power supplies (+5V and Gnd). One option is an USB to TTL cable, another option is a simple 5V transformer. The configuration as given in figure 2 uses less than 100mA. However, if you decide to connect some peripheric devices, this can go up to 500mA (before you get magic smoke). So, caution how much current you source or sink per ATmega328 pin and in total (20mA per pin, 200mA in total) is more important than a powerful transformer. If you wish to source or sink more, some plain vanilla transistors such as the 2N3904 (NPN) or the 2N3906 (PNP) can be of use.

Finally, we have broken out the Vcc (*Figure 2*, J2) and the Gnd (*Figure 2*, J3) power levels to facilitate the connection of the Visitekaarduino to external electronic components.

The printed circuit board design is shown in *Figure 3*. On the left, the frontside of the board is shown, on the right one finds the backside of the board. The populated Visitekaarduino is shown in *Figure 4*.

The size of the PCB is conforming to the CR80 standard: 85.60mm × 53.98mm with a corner radius of 3.175mm. Note that the measurements in *Figure 3* are inclusive the edge cut area. The usual thickness of a PCB is 1.6mm but we will opt for a card thickness of 0.4mm to make it more manageable as a business card. The sturdiness of the board will be maintained mostly by the soldered components, but the board must nevertheless be handled with care.

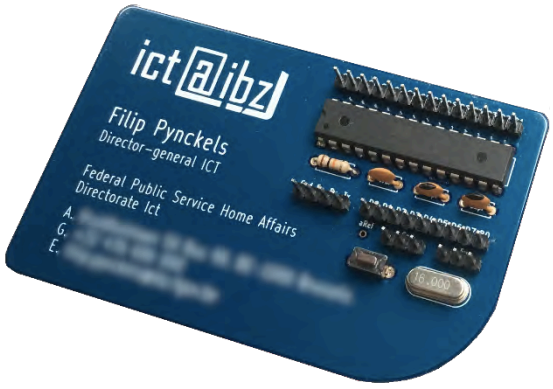


FIGURE 4 - POPULATED VISITEKAARDUINO

2. Entropy generation

When we consider random bits as the digitization of random analog signals, the complexity of generating random bits is reduced to the generation of entropy that can be translated into electric signals.

There are different methods to generate entropy. One way is to capture the nuclear background radiation with a Geiger-Müller tube and to digitize the time between 'ticks' to a random bit stream. Albeit random, this is not the method one wishes to use to make a cheap but still effective entropy generator.

Other methods exist, that are not only correct but also very artistic. One example is the LavaRand project that uses consecutive pictures of an array of lava lamps. Again, true random, but not immediately the kind of solution one wishes to use at home, or in one's data center.

Yet another method is to use the discretized trajectory of a double pendulum. Although the movements fade out after a relatively short time, there is a possibility to use the middle part of the trajectory as a true random generator. Unfortunately, this is not the solution one wants to implement when one needs a lot of random bit sequences during a longer period.

When looking at more 'electricity' oriented methods, there is the Chua circuit. This is one of the simplest electronic circuits, exhibiting not only chaos, but also bifurcation phenomena. Since this circuit is dependent on specific resistor values that are not part of the E24 series, it would be difficult for readers to reproduce the circuit in question. So, although a good method, it is not suitable for an effective open hardware/software project.

The method that is used in this paper is axed on the avalanche noise in a reverse-biased p-n junction. This can be a diode junction, or one of the junctions of a BJT (bipolar junction transistor).

2.1. Avalanche noise generation

The simplest way to get a grasp on avalanche noise is to take a diode, and to connect it the wrong way. Normally (using the conventional current definition) a diode lets current pass from the 'high voltage lead' to the 'low voltage lead'. Real electrons go the other way, but never mind that for the moment. When we place the diode upside down, there will be no substantial current flow up until a certain voltage difference between the two leads.

When we adjust the voltage difference to be larger and larger (without creating magic smoke), at a certain moment, some electrons will jump over anyway. At first not a whole lot, but the higher the voltage difference, the more electrons will pass. This value of voltage difference is called the breakdown voltage of the p-n junction.

This phenomenon is not a linear one. Meaning that using the same voltage difference, there will not always be a constant number of electrons jumping through the p-n junction. Some moments, there will be more, other moments, there will be less. And since electrons moving around the place are defined as 'current', there will be a random current flowing through the p-n junction. This randomness is dependent on the quantum-mechanical properties of p-n junctions, the atoms they are made off, and how heavy the doping is. The reality is a bit more complex, but

not particularly useful to know within the context of this paper. If, however, you like to know more about this part of our project, do not hesitate to contact one of the authors of this paper.

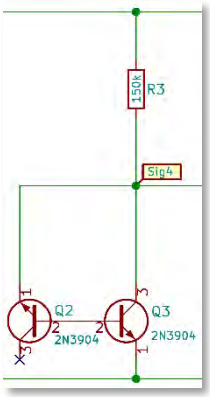


FIGURE 5 - ENTROPY GENERATOR

In this project, we will use one of the p-n junctions of 2N3904 BJT transistors, leaving the collector unconnected (see *Figure 5*, transistor Q2). These are transistors that generate more noise than an average transistor, but the disadvantage they have is that the breakdown voltage is relatively high compared to the breakdown voltage of other types of transistors.

2.2. Voltage boosting

The next problem to tackle is generating the necessary voltage to induce avalanche noise over a transistor's p-n junction. The Visitekaarduino is fed with a voltage of +5V that is clearly not enough to seduce electrons to jump through a p-n junction.

One of the easiest ways to generate a higher DC voltage from a lower DC voltage is a voltage multiplier. However, this method can generate so little current for a 'client' circuit (load), that the generated voltage drops amazingly fast when more current is taken by that load.

Using a boost converter seems a better alternative. This kind of circuits uses a pulse generator to push current pulses through a diode. The current is captured by a capacitor, that releases the current to a load circuit bit by bit. When controlling the frequency and the duty-cycle of the pule wave, the voltage boost can be made bigger or smaller. This method of boosting voltage uses relatively few components and is reliable for the kind of project we have at hand.

3. Random number generator 1.0

After creating some breadboard versions, a first working version of a random number generator saw the light (see *Figure 6* and *Figure 7*). The entire schematics together with the intermediate and final wave forms can be found in *Appendix 1*.

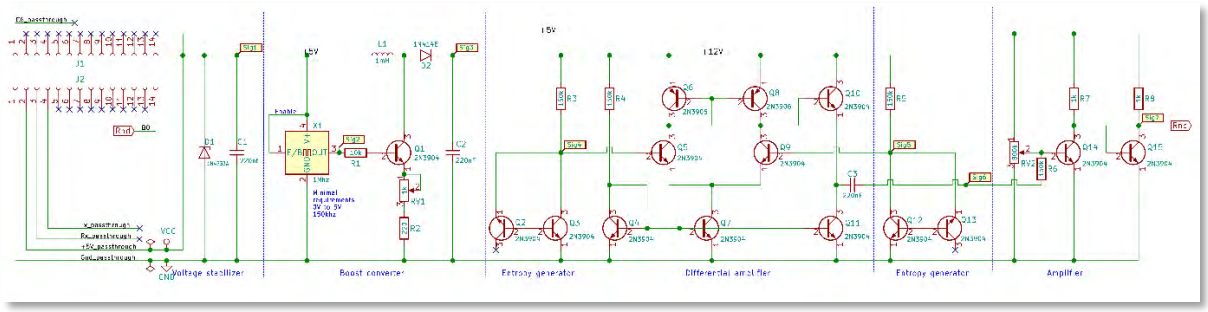


FIGURE 6 - SCHEMATICS

Some elements to notice are:

- The boost converter section uses an oscillator with a fixed frequency of ±1Mhz and a fixed duty cycle of ±50%. The generated voltage can be adjusted with potentiometer RV1. Note that the generated voltage depends on the load of this part of the circuit (hence the need for adjustability).
- The entropy generation (avalanche noise generation) is done by transistors Q2 and Q13. A double entropy generator is used to feed the two entrances of a differential amplifier.
- A differential amplifier is used to subtract the two entropy signals. Since temperature and inherent circuit noise will be available in both signals, subtracting the signals will reduce these effects substantially. What rests is the consequence of the avalanche noise generation, and only the consequence of the avalanche noise generation.
- The differential amplifier is fed with +12V, as are the entropy generators. This way, there is no need for a biasing stage between the entropy generators and the differential amplifier.
- The used differential amplifier is a minimal circuit. As a result, the output signal of the differential amplifier must be amplified further by a double amplifier stage. The latter rails the signal to make it binary. This is not ideal, but it does the job.

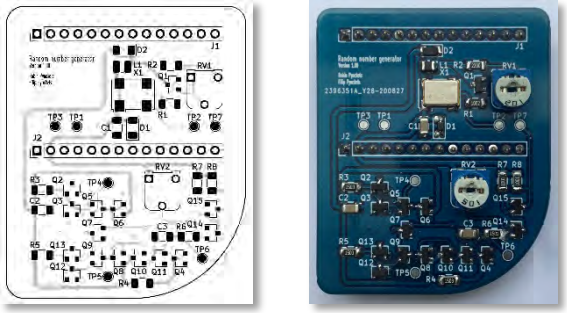


FIGURE 7 - PCB DESIGN

The resulting signal can also be found in *Appendix 1* (bottom right image, Signal 7). This signal will be read by the Visitekaarduino and transferred to a PC with the program that can be found in *Appendix 2*. Once loaded, the program makes the Visitekaarduino wait for the serial input of two numbers. The first number is the number of key sequences to generate, the second number is the number of bytes per key. The quality of the generated bit stream or of the generated random keys will be assessed on a PC using reliable open-source software test suites.

Figure 1 and *Figure 10* show the device mounted as a shield on the Visitekaarduino. This is also the test setup used to perform the quality tests described below.

WHEN ASSEMBLING THE RANDOM NUMBER GENERATOR 1.0 SHIELD, MOUNT THE TRANSISTORS UPSIDE DOWN. The reason is an problem with the default SOT-23 footprint of KiCad where the pins 1 and 2 are reversed with as a consequence that mounting the transistors upside-up results in connecting their base pin to their emitter pin and vice-versa.

4. Random number generator 1.0 - Quality test

Since there is a lot of mathematics involved, we will make abstraction of the theory and go straight to the tests. To our knowledge, there are a couple of good open source test suites for random number and pseudo random number generators.

The first test suite is composed of the statistical tests of the National Institute of Standards and Technology (NIST) in the USA. The test programs for random bit streams can be downloaded from the NIST website. The Random number generator 1.0 generates a bit stream that qualifies for these tests.

A second test suite is the GNU rng-tools. Programs like dieharder, ent and rngtest offer a lot of options to test random bit streams. The Random number generator 1.0 generated bit streams seem to have no problem with these test programs.

We can conclude that the bit stream that the Random number generator 1.0 generates is a high-quality random bit stream.

DO NOT FORGET, HOWEVER, TO CALIBRATE THE DEVICE USING THE POTENTIOMETERS RV1 AND RV2 SHOWN IN FIGURE 7 AND IN THE SCHEMATICS IN APPENDIX 1.

The calibrated device should generate the signals Sig1, Sig2, … Sig7 shown in *Appendix 1* on the respective test points TP1, TP2, … TP7. Note that signal Sig7 is a railed version of signal Sig6.

THE DEVICE MUST BE TURNED ON FOR A COUPLE OF MINUTES BEFORE THE CALIBRATION STARTS TO PERMIT THE COMPONENTS TO REACH THE AVERAGE OPERATIONAL TEMPERATURE.

Appendix 2 shows the source code for the Visitekaarduino. Once loaded, the code makes the Visitekaarduino wait for the serial input of two numbers. The first number is the number of key sequences to generate, the second number is the number of bytes per key. However, if both numbers are 0, the Visitekaarduino shows the statistics of the bitstream. These statistics can help to bias the railing amplifier with potentiometer RV2 (with other words: to assure that there are on average as many 0's as 1's generated in each batch of 250000 bits).

5. Future optimizations

Experience with the Random number generator 1.0 teaches us some potential optimization options.

5.1. Boost convertor

The actual boost convertor circuit forces us to regularly calibrate the device using potentiometer RV1. Calibrating this part of the circuit guarantees that the generated voltage is correct (between 12V5 and 13V5). More specific, that the voltage applied to the two entropy generators induces the correct amount of avalanche noise without blowing up the transistors.

The fact that calibration must be done regularly is cumbersome. The fact that there is no full-time monitoring and adjustment of the boosted voltage makes that we can never be sure to get a 100% random bit stream. On average, calibration each 10 to 20 days seems more than sufficient. But even that is tedious.

A solution to this problem is a negative feedback loop. For instance the use of a controller that reads the input voltage and the boosted voltage, and that generates a square wave with a certain frequency and a certain duty cycle. We want the boost convertor to run in continuous current mode (CCM) and to provide up to 10mA to the entropy generators.

When starting this project, we wanted to use only discrete components that are hard to influence. However, when using a controller that can be reprogrammed by the user before the first use, or whenever the user wishes to, also guarantees that fiddling with the device to induce a predictable bit stream is useless and can be overridden by the user at any time. The rest of the boost convertor stage is composed of discrete components.

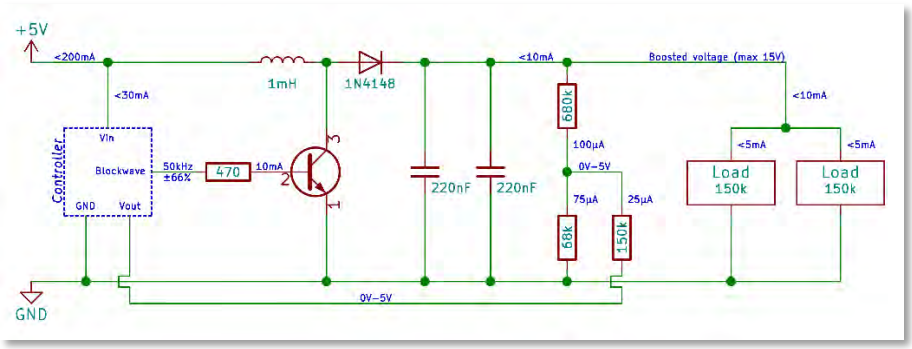


FIGURE 8 - DYNAMIC BOOST GENERATOR

An inductor of 1mH is more than large enough to be bigger than the critical inductance to achieve CCM. When using a total of 440nF capacitance and a square wave with a frequency of 50kHz, we can be sure to get an output ripple of at most 10mV, which is acceptable for avalanche noise generation. Moreover, since both entropy generators get the same voltage input, they also get the same ripple. This ripple will pass to their respective outputs that are subtracted from one another by the differential amplifier. So even if the ripple would be bigger, we could still get away with it.

The controller's job will thus be limited to adjusting the duty cycle of the square wave it generates to guarantee a correct boosted voltage with at most ±250mV deviation.

5.2. Differential amplifier

As already stated above, the differential amplifier that is used in the Random number generator 1.0 is a minimal circuit with a high output impedance, a low input impedance and hence a volatile result when too much current is drawn from its output stage. The result is that an extra amplifier stage is needed that must be calibrated from time to times (comparable to the boost convertor).

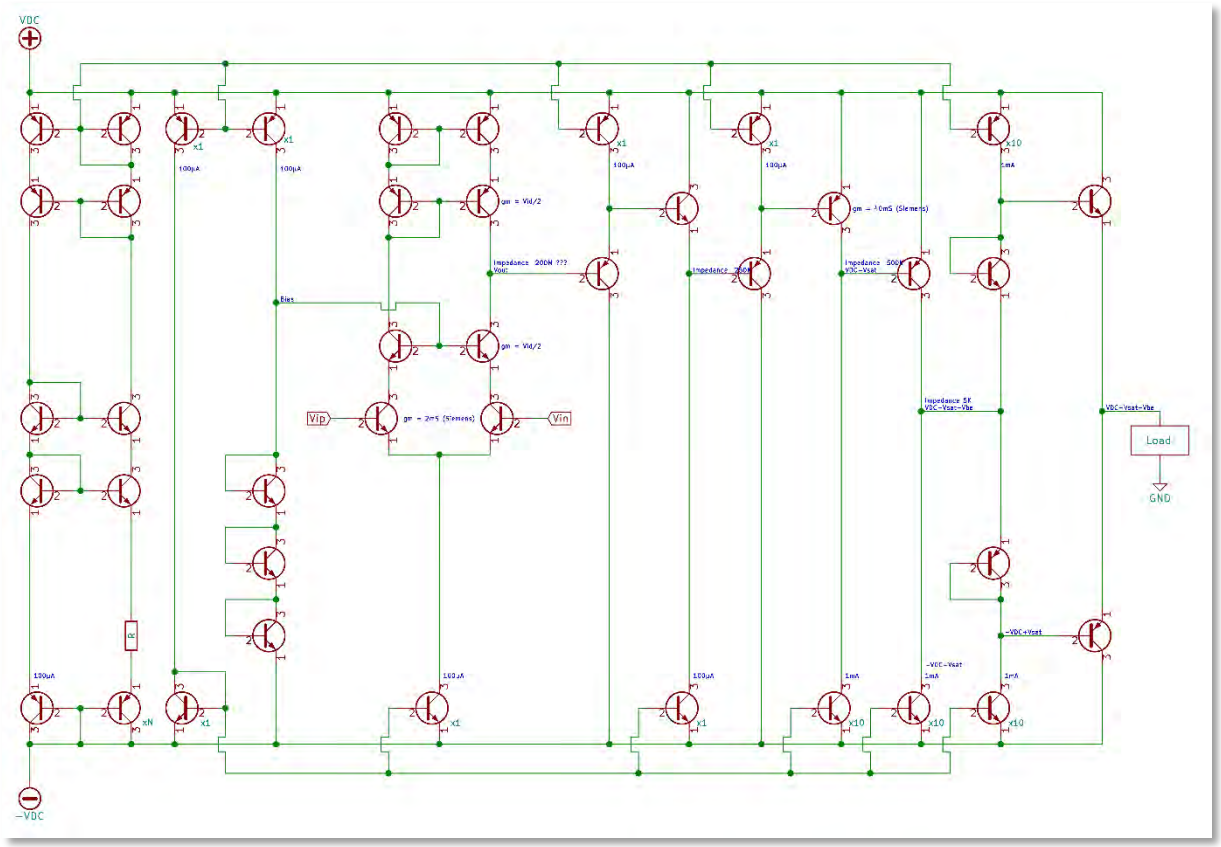


FIGURE 9 - OPTIMIZED DIFFERENTIAL AMPLIFIER

Again, we will not explain the entire development reasoning behind the optimized differential amplifier. The requirements that make it more independent of the other stages of the device are:

- Amplification factor of minimum 10^6
- Input impedance of at least 50kΩ
- Can drive a load of 50Ω or more
- As few gain stages as possible
- Uses only discrete components
- Uses only transistors with amplification β=±100

This result is the design shown in *Figure 9* and more readable in *Appendix 3*. Unfortunately, the optimized design is already a good (but not great) operational amplifier. Implementing this design with discrete components would take a lot of PCB real estate. So, the question arises if using an operational amplifier on a chip would not be a better choice. Every 'opamp' IC can be tested before use to assess correct functioning.

Supplementary advantages of an 'opamp' on a chip is the fact that it is more temperature resilient, that it consumes less current and that the transistors are much smaller and closer to each other which guarantees less electro-magnetic disturbance.

6. Conclusion

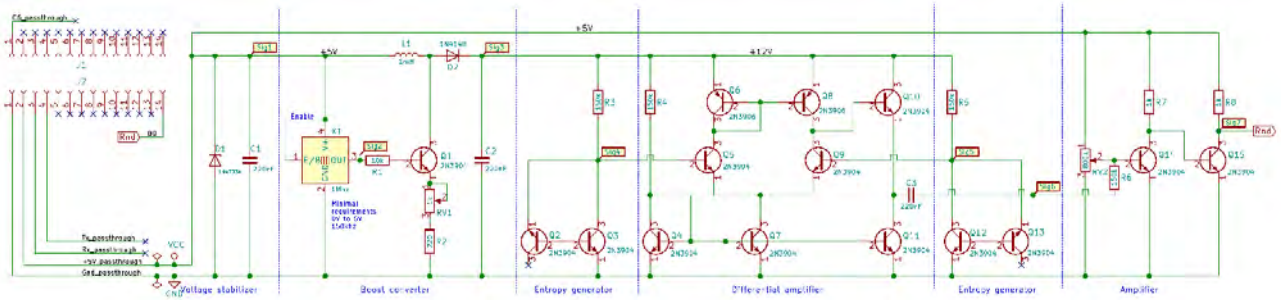
Figure 10 shows the Random number generator mounted on the Visitekaarduino. This design, although not yet independent enough (calibration aspect) for long-term high-performance professional use in a datacenter, is more than sufficient to be used by an individual who wishes to generate random security keys instead of pseudo random security keys, random bit streams for scientific research, etc. The generated bit streams (after calibration of the device) are compliant with the NIST tests and with the tests of the GNU rng-tools.

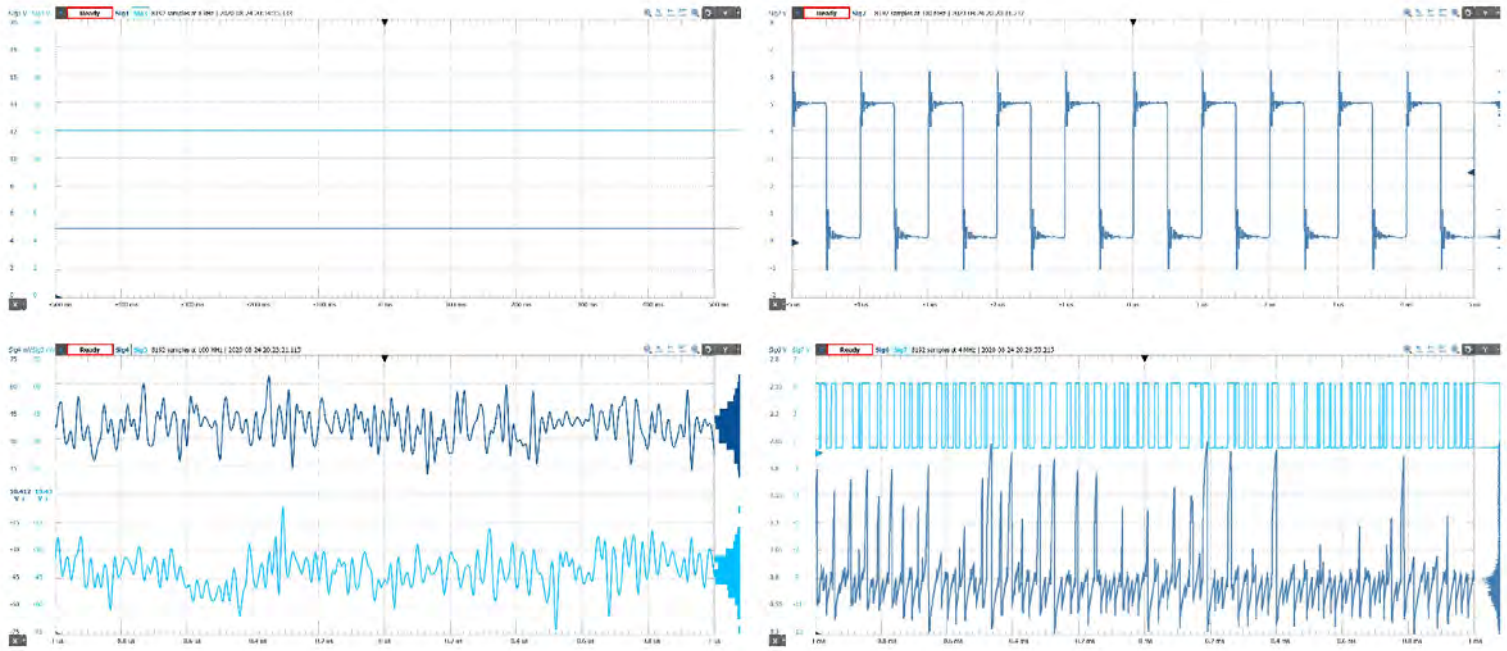
FIGURE 10 - FUNCTIONING DEVICE

The next steps of this project will build further on the experience we have with this first version. Whether the next versions will still be a Visitekaarduino shield, or a stand-alone design is a choice that will be made later.

It is clear, however, that to obtain a long-term high-performance usability without manual interventions, some specifications will have to change. Using less discrete components, letting the device calibrate itself full-time and augmenting the number of keys that can be generated per minute are only some of them.

Appendix 1 - Random number generator 1.0 - Schematics





Appendix 2 - Visitekaarduino - Program

```
1.  //=============================================================================
2.  //
3.  // Project: entropy
4.  // File:    source_1_0.ino
5.  // Version: 1.00
6.  //
7.  // Author : Filip Pynckels
8.  //
9.  // Changed: 2020 07 27
10. //
11. // Tools:   Arduino IDE
12. //          Visitekaarduino
13. //          Hardware entropy generator
14. //
15. //=============================================================================
16. //
17. // This program is free software. You can redistribute it and/or modify it under
18. // the terms of the MIT Public License.
19. //
20. // This software is distributed in the hope that it will be useful, but without
21. // any warranty. Without even the implied warranty of merchantability or fitness
22. // for a particular purpose.
23. //
24. //=============================================================================
25.
26. // ======================================================
27. // = Constants                                          =
28. // ======================================================
29.
30. #define SERIAL_SPEED        1000000                     // Serial port settings
31. #define SERIAL_CONFIG       SERIAL_8N1
32. #define PIN_IN              8                           // Data input pin (PB0)
33. #define SAMPLES_PER_STAT    250000
```

```
34.
35. // ======================================================
36. // = Show bit statistics (calibration data)             =
37. // ======================================================
38.
39. void showStatistics()
40. {
41.    long unsigned nrZeros = 0;
42.    long unsigned nrOnes  = 0;
43.    float pctZeros;
44.    float pctOnes;
45.
46.    long unsigned statCounter = 0;
47.
48.    while (true) {
49.        byte bit=digitalRead(PIN_IN) & 0x01;
50.
51.        switch (bit) {
52.            case 0:
53.                nrZeros++;
54.                break;
55.
56.            case 1:
57.                nrOnes++;
58.                break;
59.        }
60.
61.        statCounter++;
62.
63.        if (statCounter == SAMPLES_PER_STAT) {
64.            statCounter = 0;
65.
66.            pctZeros  = nrZeros;
67.            pctZeros /= (nrZeros+nrOnes);
68.            pctZeros *= 100;
69.            pctOnes   = nrOnes;
70.            pctOnes  /= (nrZeros+nrOnes);
71.            pctOnes *= 100;
```

```
72.
73.             Serial.print("  0: ");
74.             Serial.print(nrZeros);
75.             Serial.print(" (");
76.             Serial.print(pctZeros);
77.             Serial.print("%)\t1: ");
78.             Serial.print(nrOnes);
79.             Serial.print(" (");
80.             Serial.print(pctOnes);
81.             Serial.print("%)");
82.             Serial.println();
83.
84.             nrZeros = 0;
85.             nrOnes  = 0;
86.         }
87.     }
88. }
89.
90. // =======================================================
91. // = Execute an order for keys with a specified length =
92. // =======================================================
93.
94. void executeOrder(unsigned long numberKeys, unsigned long bytesPerKey)
95. {
96.     if ((numberKeys == 0) && (bytesPerKey == 0)) showStatistics();
97.
98.     if (numberKeys  == 0) return;                       // Parameter checking
99.     if (bytesPerKey == 0) return;
100.
101.    long unsigned keyCount = 0;                         // Iterate over numberKeys
102.    while (keyCount < numberKeys) {
103.
104.        long unsigned byteCount = 0;                    // Iterate over bytesPerKey
105.        while (byteCount < bytesPerKey) {
106.
107.            byte bitCount  = 0;                         // Iterate over 8 bits per byte
108.            byte byteValue = 0;
109.            while (bitCount < 8) {
```

```
110.                byteValue = (byteValue << 1) | digitalRead(PIN_IN) & 0x01;
111.                bitCount++;                                 // Another bit of the byte received
112.            }
113.
114.            char hexBuffer[3];                              // Print byte in format %02X
115.            sprintf(hexBuffer, "%02X", byteValue);
116.            Serial.print(hexBuffer);
117.            byteCount++;                                    // Another byte bites the dust
118.        }
119.
120.        Serial.println();                                   // Add CR/LF at end of key
121.        keyCount++;                                         // Another key has sailed
122.    }
123. }
124.
125. // =======================================================
126. // = Setup the Visitekaarduino                           =
127. // =======================================================
128.
129. void setup()
130. {
131.    Serial.begin(SERIAL_SPEED, SERIAL_CONFIG);           // Initialize UART ('Serial' for the friends)
132.    while (!Serial);                                     // Wait until serial port is available
133.    while (Serial.available()) Serial.read();            // Empty Rx buffer should clutter be available
134.    pinMode(PIN_IN, INPUT);                              // Prepare input pin
135. }
136.
137. // =======================================================
138. // = Go into eternal program loop                        =
139. // =======================================================
140.
141. void loop()
142. {
143.    long unsigned bytesPerKey;                           // Parameters for key generation
144.    long unsigned numberKeys;
145.
146.    while (!Serial.available());                         // Wait for the beginning of a command
147.    String      cmd  = Serial.readStringUntil('\n');     // Wait for the command to finish (eol)
```

```
148.     const char* ccmd = cmd.c_str();                    // Prepare sscanf()
149.     int         rc   = sscanf(ccmd, "%lu %lu", &numberKeys, &bytesPerKey);
150.     if (rc == 2) executeOrder(numberKeys, bytesPerKey); // Execute the command if it is kinda valid
151. }
152.
```

Appendix 3 - Optimized differential amplifier