Building a Hardware Random Bit Generator for EVPmaker

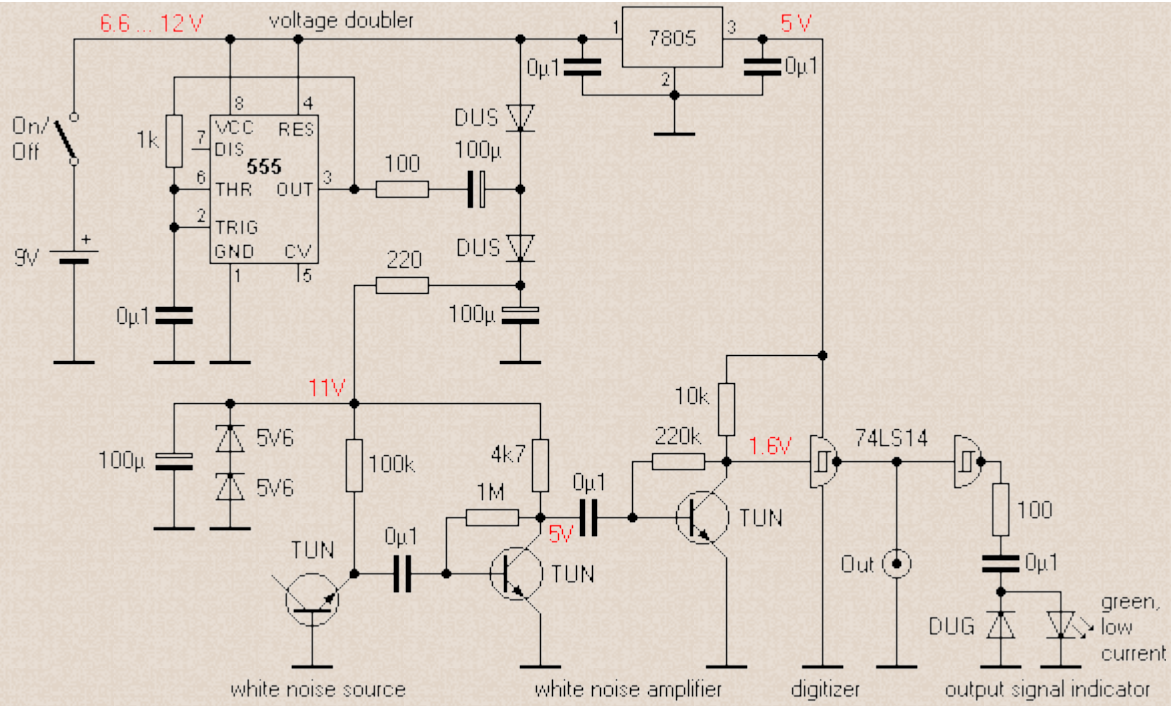
This article shows how you can build a simple hardware random bit generator for EVPmaker.

Although EVPmaker functions well with software-generated pseudo random numbers, it might be advantageous to use "true" random numbers. Perhaps it's possible for the "EVP entities" (which are perhaps our own subconsciousnesses) to influence these true random numbers more easily than the deterministic pseudo random numbers (which in my opinion are actually unchangeable). EVPmaker already offered an option to use the soundcard input as a random source. From version 2.4 on it's also possible to use an external hardware random bit generator connected to the computer's parallel printer port.

Shortcuts

The Schematic Diagram

The noise source is a transistor operated in base-emitter reverse biasing. It needs at least 9 volts to generate noise, so a voltage doubler is used (constructed around the 555 timer IC) to ensure that the voltage is sufficient even with low battery power. The white noise signal is amplified in the two following transistor stages, and finally fed into a shmitt-trigger to get a digital TTL level signal. The second schmitt-trigger drives the LED which indicates the presence of an output signal. The capacitor at the output of the second shmitt-trigger causes the LED to flash only if a high frequency random signal is present at the output. The average frequency of the random signal is about 1 MHz and can be heard due to the harmonics of the square wave signal in any radio band (MW, SW, and even FM) of a nearby radio as a loud hissing sound.



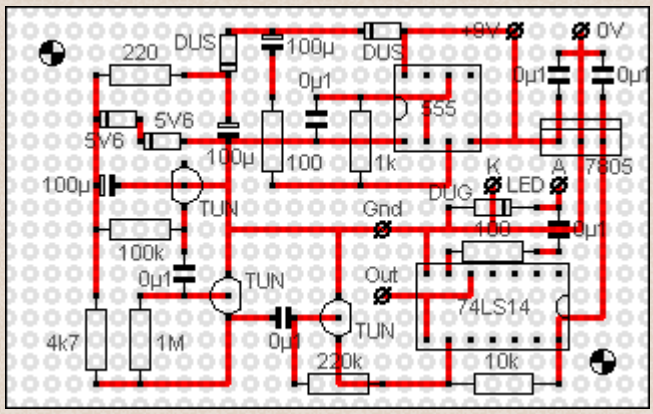
The abbreviations and their meanings:

- DUS = "Diode Universal Silicon" (e.g. 1N914, 1N4148, etc.)
- DUG = "Diode Universal Germanium" (e.g. 1N34A, AA112, etc.)
- TUN = "Transistor Universal NPN" (e.g. 2N3904, BC547, etc.)

The voltage regulator IC "7805" can be the 0.1A type "78L05"; I used the 2A type "78S05" only because I had it in my box. :-)

The Circuit Board

I have built the circuit on a piece of perforated circuit board. Here's the layout:

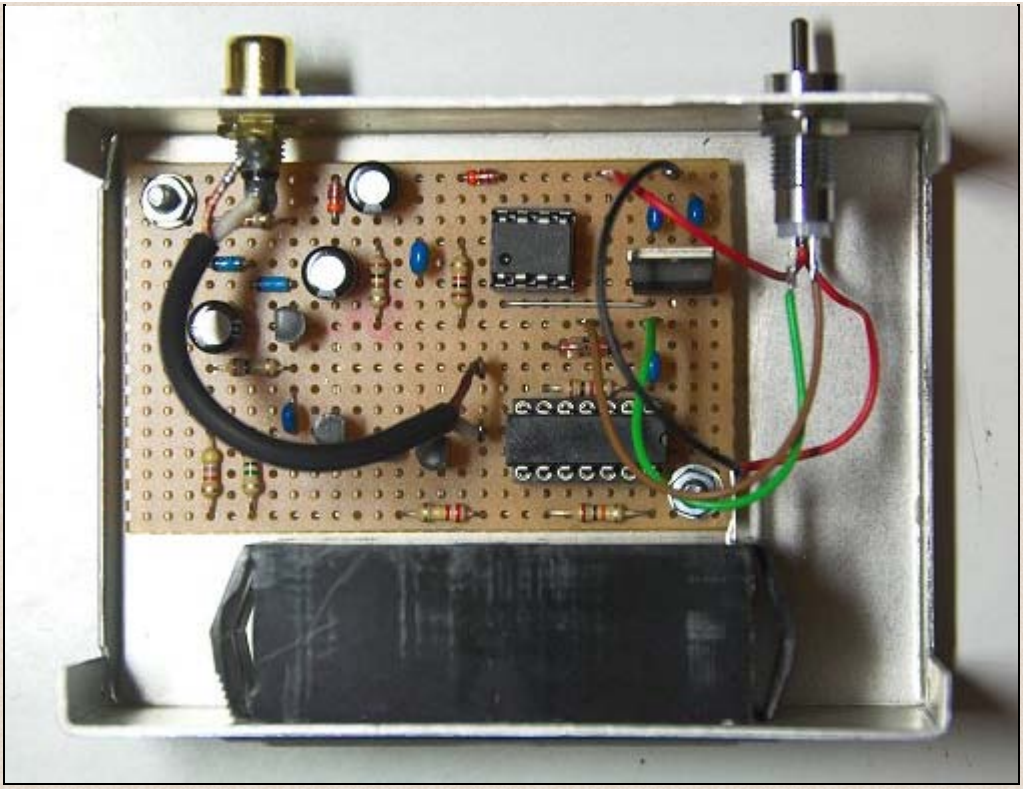


Top view of the perforated circuit board

The red lines are the soldered connections between the components on the copper side of the board.

The Mechanical Construction

Since the random bit generator is very sensitive due to its high-impedance input and its high gain, and since it also radiates radio waves, I have built it into an aluminum case:



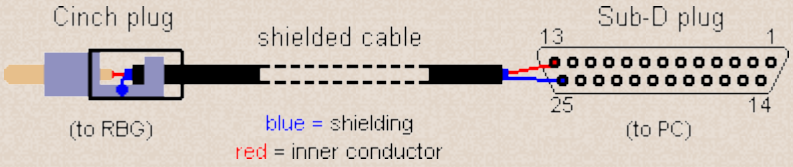
The finished device with removed cover

At the front side of the case (top of the image) you can see the Cinch jack for the signal output, the power switch, and the LED holder. The black box at the opposite side is the battery holder. The circuit board is fixed to the case bottom using spacer sleeves and M3 screws.

The connected device in action

To connect the random bit generator to the computer's parallel printer port, I have made a cable with a Cinch plug at one end, and a male 25-pin Sub-D plug at the other end.

The shielding of the cable is soldered to pin 25 (Ground) of the Sub-D plug, the inner conductor to pin 13 ("Select" input):



Using the RBG with software applications

To request random bits from the generator, the Select line of the printer port is queried. To find out the 2-byte port adresses of the installed parallel ports LPT 1 to 3, the memory addresses 0x408, 0x40A, and 0x40C can be read. Usually, the first LPT port, LPT1, is at port address 0x378. Each LPT port has 3 registers of 8 bits each: The data register at the base address, the status register at base address + 1, and the control register at base address + 2. The status register has 5 input lines that can be used to connect the RBG: "Acknowledge" at pin 10 of the 25-pin connector (bit 6 of the status register), "Busy" at pin 11 (bit 7), "Paper Out" at pin 12 (bit 5), "Select" at pin 13 (bit 4), and "Error" at pin 15 (bit 3).

Here's some sample C++ code that shows how to read the status register of LPT1 and query the Select line for a single random bit:

```
int nBaseAddressLPT1 = ReadMemShort(0x408);
int nStatusRegister = Inp32((short)(nBaseAddressLPT1 + 1));
int nRandomBit = (nStatusRegister >> 4) & 1;
```

To get a 32-bit random number, the port has to be queried 32 times, while each retrieved bit is shifted into a 32-bit integer variable:

```
unsigned int nRandomNumber = 0;
for(int nBit = 0; nBit < 32; nBit++)
{
        int nStatusRegister = Inp32((short)(nBaseAddressLPT1 + 1));
        int nRandomBit = (nStatusRegister >> 4) & 1;
        nRandomNumber <<= 1;
        nRandomNumber |= nRandomBit;
        // (Wait)
}
```

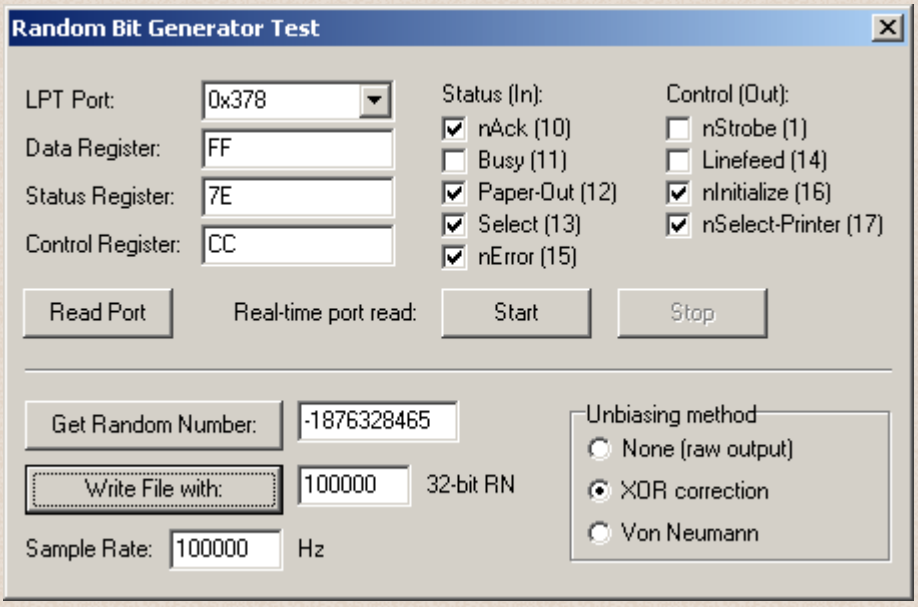
At the location denoted with "(Wait)", the program should wait for a short time to give the RBG time to output a new random bit which is not correlated to the previous one. The amount of time depends on the sample rate. For 100 kHz, the time should be 10 µs. Since the Sleep() function has a resolution of maximum 1 ms, use QueryPerformanceFrequency() and QueryPerformanceCounter() instead. Please refer to your developer manual for more information on that.

While reading memory addresses and ports was an easy task in the good old MS-DOS days, a device driver has to be used under Windows. Unfortunately, there's no such device driver by default that applications can use to read memory or port addresses, so some nice people have written such a device driver, for instance **www.logix4u.net**. Their hwinterface32.dll DLL enables programs to access (read and write) any memory location or I/O port of the computer.

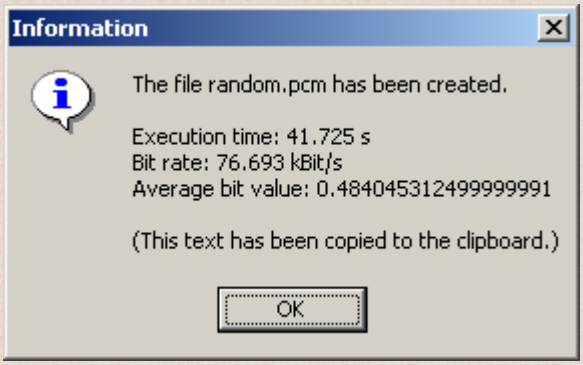
You'll find a working example (including the source code for Borland C++ Builder 4.0) in the "Random Bit Generator Test" program described below.

Testing the functionality of the RBG

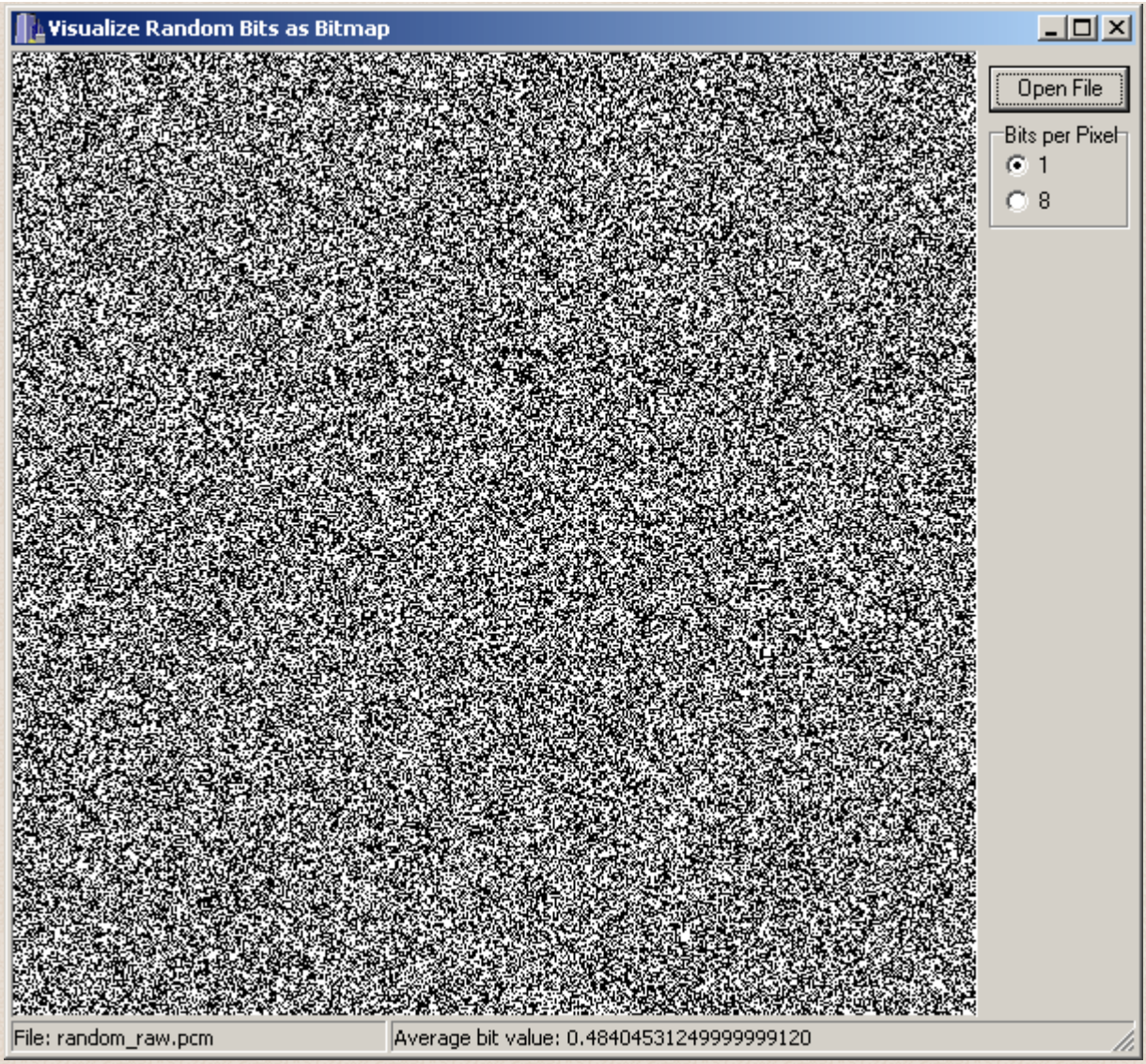
For a rough test of the random bit generator, I have written two simple programs which are available for download (**see below**). The first one, called "**Random Bit Generator Test**", shows the values of the three registers of the parallel printer port in real-time, and it lets you generate a file with random bits. This file is always called "random.pcm" and can be analyzed by any other program.



You can specify the number of 32-bit random numbers to be generated, the sample rate, and the **unbiasing method**. After the file is written, a message box displays some statistical data:



The second program, "**RandomBitmap**", displays the bits of a file as pixels. By default, each "1" bit is shown as a black pixel, and each "0" bit as a white pixel:



This way you can see if there are any patterns in the signal which may be the case for some pseudo random generators but should not be the case for true random numbers.

By default, 512 x 512 pixels (= the first 262,144 bits from the file) are displayed, but you can maximize the window to display more pixels/bits.

If you alter the setting "Bits per Pixel" to 8, eight bits from the file are used to determine the brightness of one pixel, so a pixel can have 256 (2^8) different brightness values.

The "Average bit value" in the status line at the bottom of the above screen displays the ratio of the "1" bits to the total number of bits in the file. Ideally, this should be exactly 0.5, but as you can see, there's a slight bias in the outputted bits: The average bit value is about 0.484, or in other words, only 48.4% of the bits are "1", 51.6% are "0". For EVPmaker, this shouldn't make a notable difference, but nevertheless you have the option to use one of two unbiasing methods: The XOR correction method, and the Von Neumann method. The XOR method reads 2 bits and outputs a 1 if both are different and a 0 if they are equal. The Von Neumann method also reads 2 bits but outputs the first one if both are different and throws them away if they are equal. Example:

Raw output	10	11	00	10	10	01	00	01	10	10	01	01	11	00	10	00	10	10	01	01	11
XOR correction	1	0	0	1	1	1	0	1	1	1	1	1	0	0	1	0	1	1	1	1	0
Von Neumann	1			1	1	0		0	1	1	0	0			1		1	1	0	0	

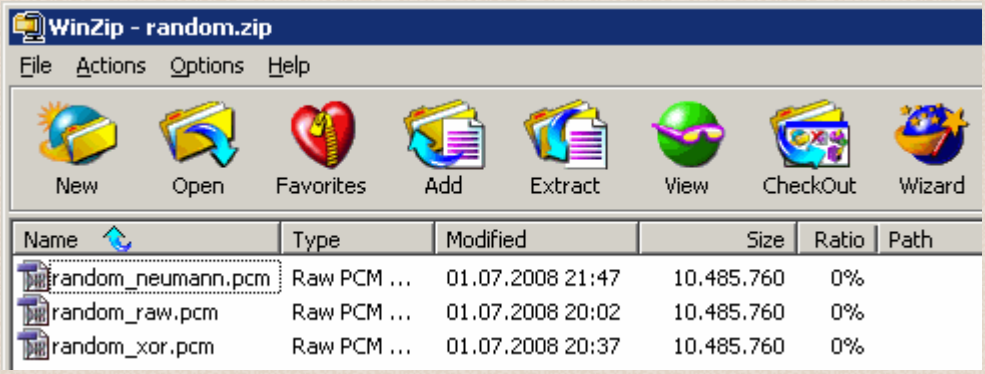
For a more detailed explanation, see for example **here**.

In the following table you can see and compare the results of these unbiasing methods. For each test, a 10 MB file was created, giving a total of 83,886,080 random bits, or 2,621,440 random numbers of 32-bit each. The sample rate used was 100,000 Hz.

Unbiasing method	Execution time	Bit rate	Average bit value	Deviation from 0.5
None (raw output)	17.5 min	80 kBit/s	0.484252	-3.150%
XOR correction	35 min	40 kBit/s	0.499724	-0.055%
Von Neumann	70 min	20 kBit/s	0.500100	+0.012%

As you can see, the XOR correction method needs twice as much time compared to the uncorrected raw output, but has less deviation from the ideal average bit value of 0.5, and the Von Neumann method needs four times as much time, but has an even less deviation.

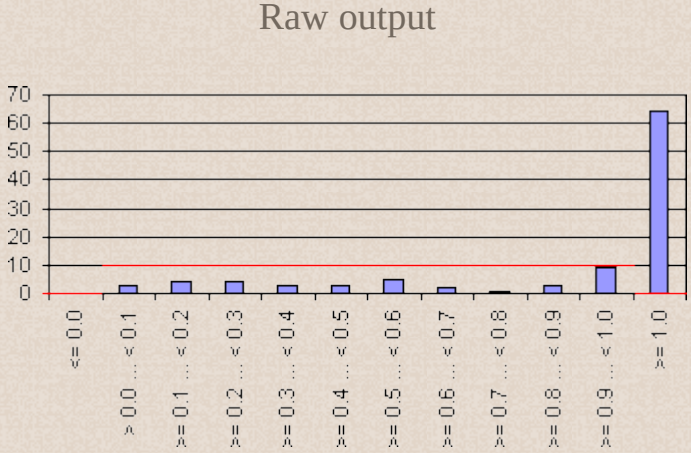
Another property of a sequence of good random numbers is that it can't be compressed further. I tested this with WinZip, and as can be seen, it wasn't able to reduce the file size of the random bit files generated from the device, even with the compression set to 'Maximum':



The Diehard test

For a more thorough test, I ran the three above 10 MB files through the "**Diehard**" test which is considered as a de facto standard for testing randomness. Since the original Diehard report looks kind of confusing, I wrote a little command line tool, called "**DiehardCSV**" (which can be **downloaded** as well), which extracts the "p-values" from the report and writes them into two CSV files: One file with all p-values (including the "KS test" values), and one summary file with the number of all p-values distributed over a 10% raster. The CSV files can be imported into Excel for further analysis. Here are the results for the three files:

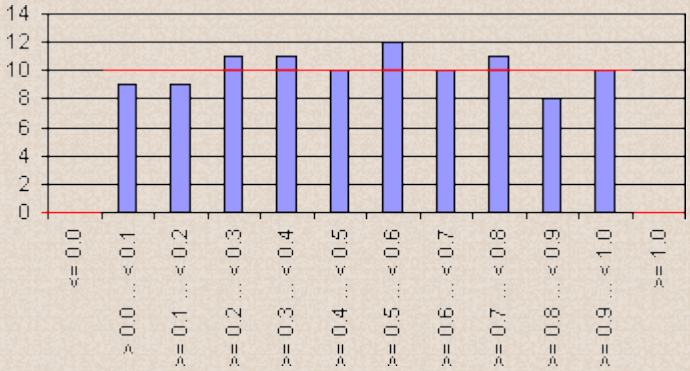
p-value Range	Number of p-values	Observed Percent	Expected Percent
= 0.0	0	0	0
> 0.0 ... < 0.1	6	3	10
= 0.1 ... < 0.2	8	4	10
= 0.2 ... < 0.3	9	4	10
= 0.3 ... < 0.4	6	3	10
= 0.4 ... < 0.5	7	3	10
= 0.5 ... < 0.6	11	5	10
= 0.6 ... < 0.7	4	2	10
= 0.7 ... < 0.8	3	1	10
= 0.8 ... < 0.9	7	3	10
= 0.9 ... < 1.0	19	9	10
= 1.0	140	64	0



Original report

p-value Range	Number of p-values	Observed Percent	Expected Percent
= 0.0	0	0	0
> 0.0 ... < 0.1	20	9	10
= 0.1 ... < 0.2	19	9	10
= 0.2 ... < 0.3	24	11	10
= 0.3 ... < 0.4	24	11	10
= 0.4 ... < 0.5	22	10	10
= 0.5 ... < 0.6	26	12	10
= 0.6 ... < 0.7	21	10	10
= 0.7 ... < 0.8	24	11	10
= 0.8 ... < 0.9	17	8	10
= 0.9 ... < 1.0	23	10	10
= 1.0	0	0	0

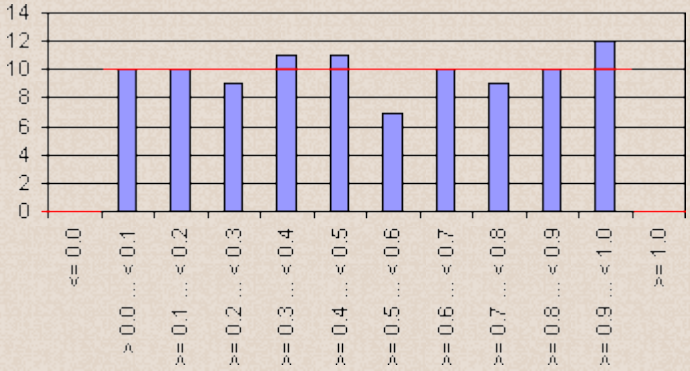
XOR correction



Original report

p-value Range	Number of p-values	Observed Percent	Expected Percent
= 0.0	0	0	0
> 0.0 ... < 0.1	23	10	10
= 0.1 ... < 0.2	23	10	10
= 0.2 ... < 0.3	20	9	10
= 0.3 ... < 0.4	24	11	10
= 0.4 ... < 0.5	24	11	10
= 0.5 ... < 0.6	15	7	10
= 0.6 ... < 0.7	23	10	10
= 0.7 ... < 0.8	20	9	10
= 0.8 ... < 0.9	22	10	10
= 0.9 ... < 1.0	26	12	10
= 1.0	0	0	0

Von Neumann



Original report

All p-values should occur evenly between (excluding) 0.0 and 1.0. While the corrected random numbers look pretty good, the raw output of the RBG didn't pass the Diehard test: 140 of the 220 p-values (64%) are 1.0 which indicates that there's something wrong. I'm no expert in mathematics, but I guess the reason for this is the relatively high bias of the output, i.e. the average bit value of 0.484. However, I think this is only relevant for certain applications like cryptography, scientific tests, or lottery. For EVPmaker, it should make no difference, so here also the direct raw output may be used.

Downloads

The above mentioned testing programs written by me can be downloaded here:

- **Download RBGTest** - displays the LPT port lines and writes random number files
- **Download RandomBitmap** - displays binary file contents as a bitmap
- **Download DiehardCSV** - converts Diehard reports into CSV files (all p-values & summary)

There's no installation program; just unzip the files into any folder and start the .exe files. The source code for Borland C++ Builder 4.0 is included, so you can adapt it and use it in your own programs.

The original "Diehard Battery of Tests of Randomness" can be downloaded from the website of the Department of Statistics at Florida State University:

- **Diehard Battery of Tests of Randomness**

Troubleshooting

EVPmaker as well as the RBGTest program uses the hwinterface32.dll from **www.logix4u.net** to access the computer's I/O ports and memory addresses, which is usually not possible under Windows because of its security mechanisms. This should work under all Windows versions except for 64-bit versions. On the first program start, a driver (hwinterface32.sys) is automatically copied to the Windows system32\drivers directory. This requires administrator rights, so if you're usually working as a standard (limited) user, you need to run the application as administrator once which should install the driver.

Homepage > **Software** > **EVPmaker** > Random Bit Generator