Stochastic processes in embedded systems – Random number generation

(Copyright 2007 Sebastian Sohn – s-sohn [at] web [dot] de)

© 2007 Sebastian Sohn

1. Introduction

In many applications random numbers are needed. However, it's not possible to generate random numbers with any kind of software algorithm. CPUs are designed to behave deterministically, thus any algorithm behaves deterministically, too. Standard random libraries which are available for almost every compiler generate pseudo random numbers. The generated number series may look random but every generated number depends on its parent. If the algorithm is known the number series is predictable. For most applications this is not a problem but in security applications it is.

For example in automotive applications almost every electronic control unit (ECU) supports a security mode, in which operation parameter can be changed or a new key can be educated. A key can only unlock a car if the key is known by the car. The operation to educate a car for a new key must be well protected; otherwise it would be easy to steal the car.

To enter security mode, usually the tester needs to request a security seed from the ECU. The seed is a random number. The tester uses the seed to calculate the response by using a confidential algorithm and sends it back to the ECU. The ECU uses the same confidential algorithm and compares the received response with its own result. If both values are equal, the ECU enables access to security parameter.

There is one valid response for each seed. Thus it is important that the seed is not predictable, otherwise it would be much easier to crack the security system. There are pseudo random generators available which are very hard to crack. However, the safest seed would be a true random number. But as stated above a CPU can't produce true random numbers.

Nowadays microcontrollers have a lot of peripherals. Some peripheral can be used to generate true random numbers. A random source usually results stochastically distributed values. Some possibilities are discussed here.

I'm familiar with Atmel's AVR microcontroller family, so I have chosen an ATmega164P for doing some tests. It's an up-to-date very low power device that is also available for automotive applications. The ATmega164P has a lot of build-in random number sources, making it a good candidate for this project.

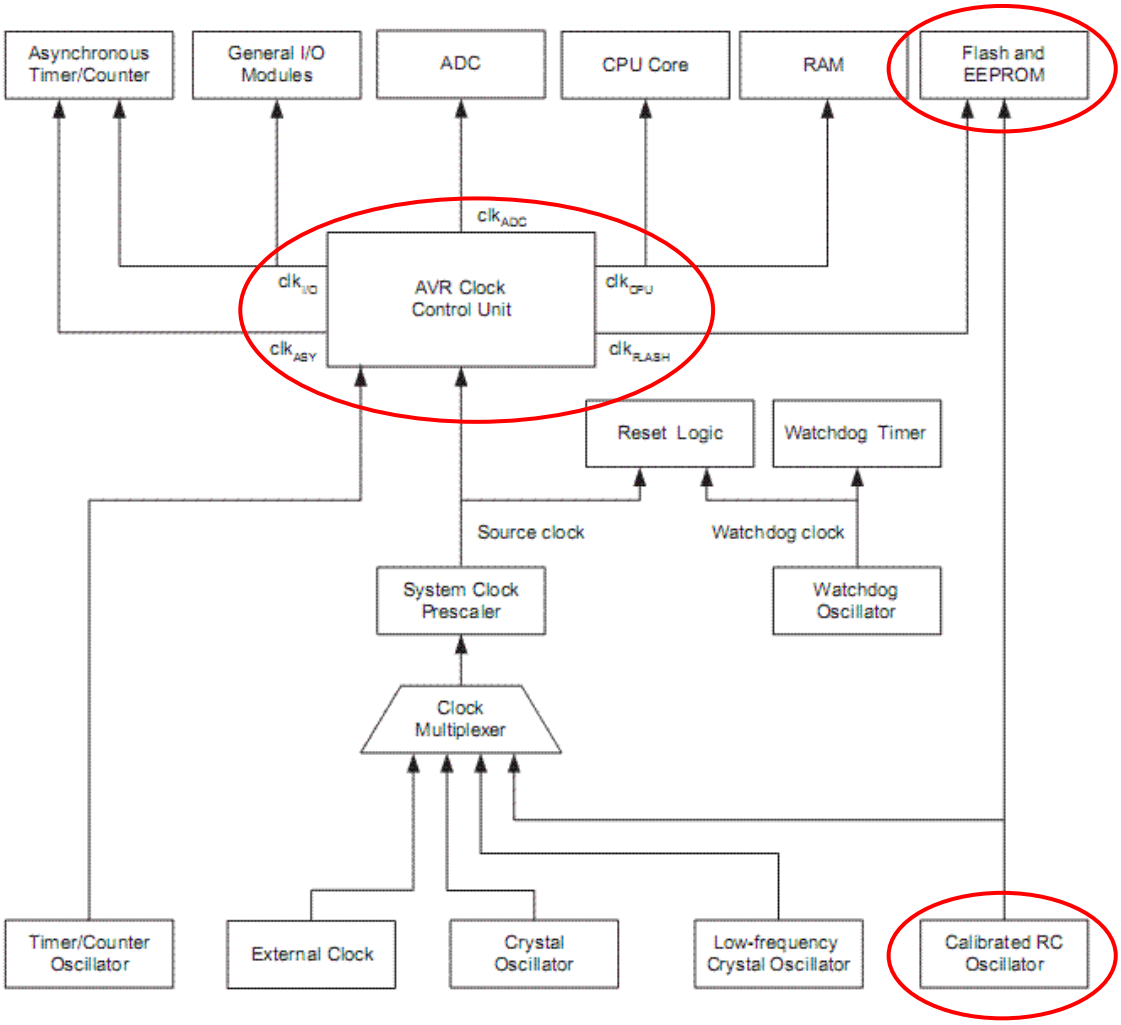
For development tools, I used AVR Studio version 4.13 build 528 and WinAVR-20070525. Both tools are royalty free. For debugging sessions I used the simulator in AVR Studio and an AVR Dragon which enables on chip debugging via JTAG interface.

© 2007 Sebastian Sohn

2. Random sources in the ATmega164P microcontroller

2.1. Internal EEPROM

The following schematic is taken from the ATmega164P datasheet:

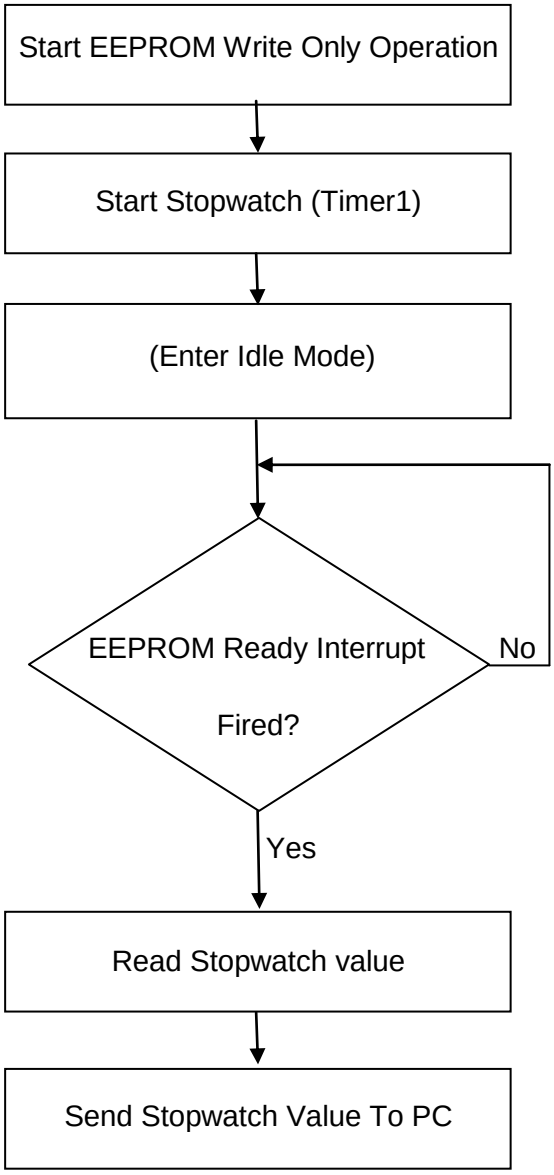


The schematic shows the EEPROM using two different clock sources. An EEPROM write operation is always timed by the calibrated internal oscillator while all other operations are timed by the AVR main clock. If the main clock is not the calibrated internal oscillator, the EEPROM uses two independent clock sources. Note that two independent clocks are never in phase lock. Since the two clocks are never in phase-lock, the time between edges of one clock to the other will vary randomly. Thus there will always be a randomly varying time difference of a few main clock cycles between any two EEPROM write cycles.

I have chosen a 9.216MHz crystal as main clock for the test. It's a multiple frequency of standard baudrates. A timer is used to measure the number of main clock cycles for an EEPROM write operation. The result is send via RS232 interface to a computer. Look at the ATmega164P source code for details (*eep.c*).

© 2007 Sebastian Sohn

2.1.1. Flowchart



The internal EEPROM of the ATmega164P supports 3 operating modes: 1. Atomic erase and write operation, 2. Erase only, 3. Write only. The erase operation resets an EEPROM byte to 0xFF; the write operation can set bits to zero. Thus when using write only mode and always write 0xFF to an EEPROM byte, the EEPROM content is never changed and the EEPROM memory does not wear out after 100,000 write operations (see datasheet).

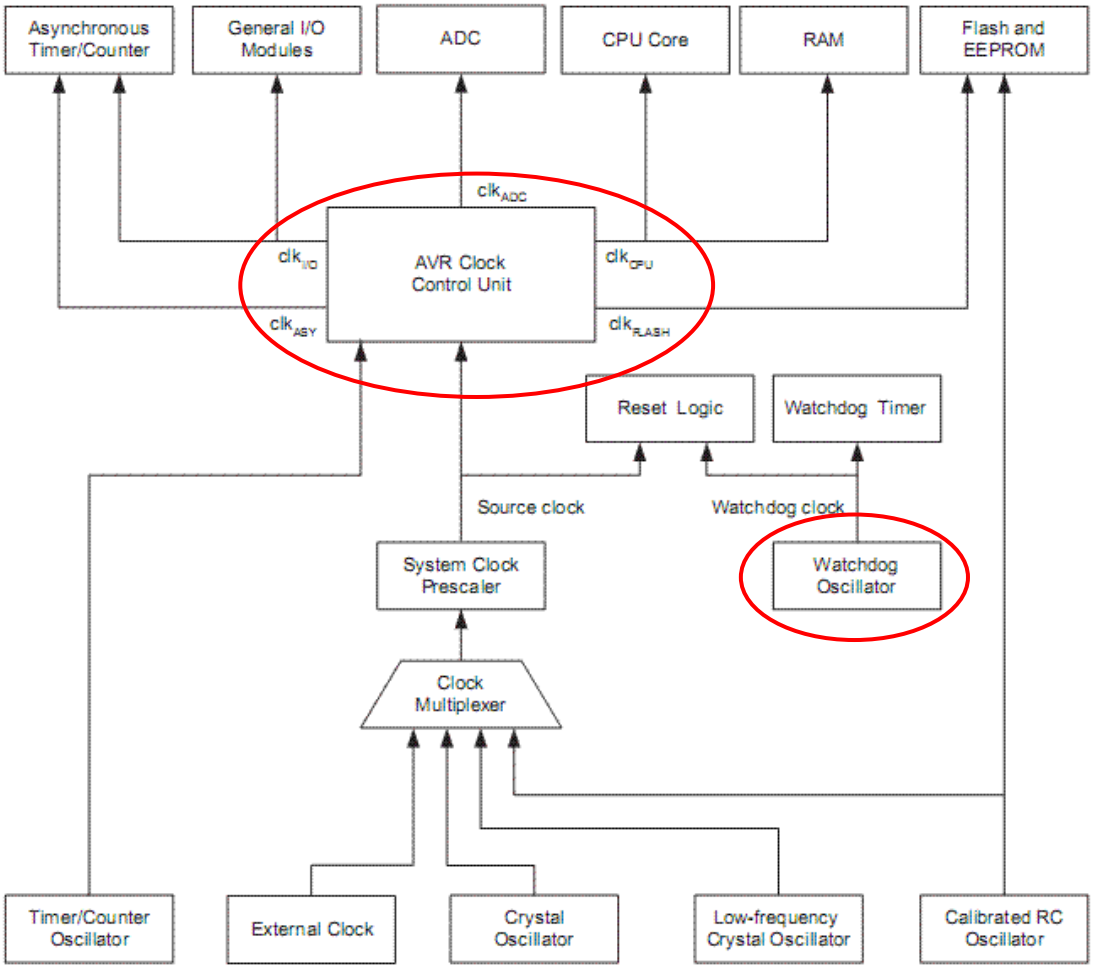
Entering Idle sleep mode is not necessary but avoids effects which are described in chapter 2.8. To test only the random source quality of the EEPROM it was necessary to switch off other side effects.

A single write-only operation takes 1.8ms.

© 2007 Sebastian Sohn

2.2. Watchdog reset

The following schematic is taken from the ATmega164P datasheet:



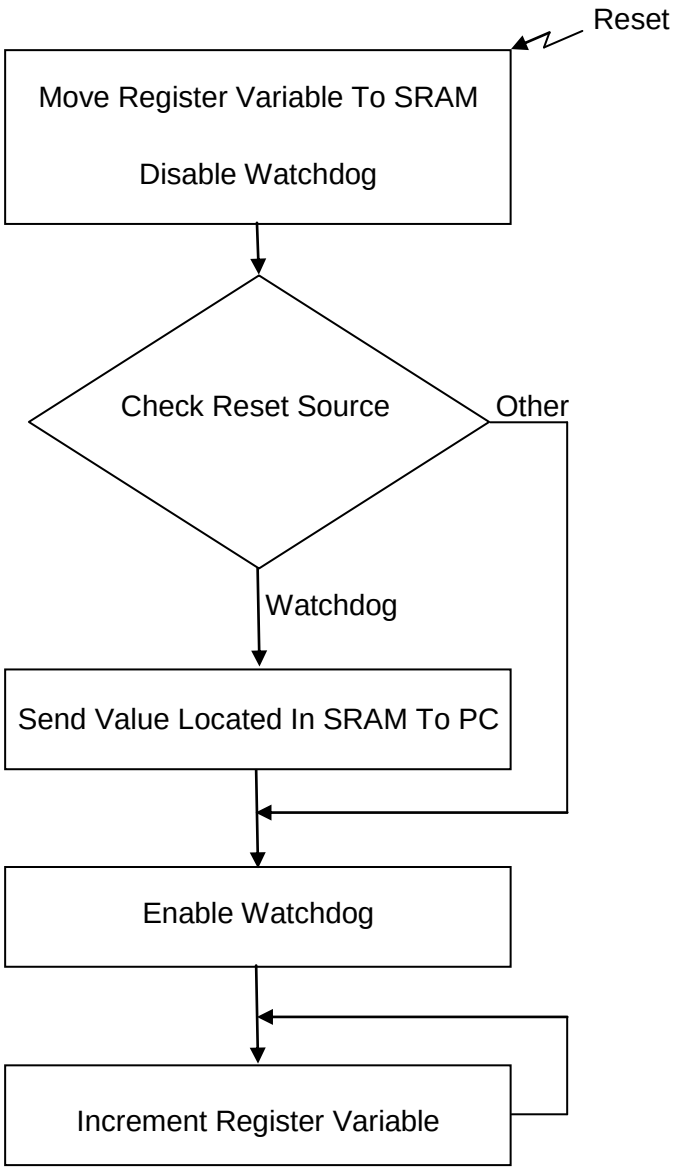
The ATmega164P has a build in watchdog. This watchdog is clocked by an internal stand-alone 128 kHz clock source. If the main clock is not the internal 128 kHz oscillator, two independent clocks are used. Note that two independent clocks are never in phase lock. Since the two clocks are never in phase-lock, the time between edges of one clock to the other will vary randomly. Thus there will always be a randomly varying time difference of a few main clock cycles between any two watchdog resets.

On a timeout the watchdog resets the ATmega164P. A reset affects all special function registers; they are reset to their default values. Thus a timer can't be used to measure the watchdog timeout. But the 32 general purpose registers and the internal SRAM are not affected – a variable can be continuously incremented instead of a timer. The AVR core can increment a 16-bit register variable very fast. In an endless loop it takes only 4 main clock cycles. After the watchdog is enabled, the register variable is continuously incremented until the watchdog resets the microcontroller. After the reset the 16-bit register variable is read and the value is

© 2007 Sebastian Sohn

send via RS232 interface. Look at the ATmega164P source code for details (*watchdog_reset.c*).

2.2.1. Flowchart



A watchdog reset does not affect the content of the 32 general purpose registers and the SRAM. The AVR core can increment an integer variable located in these registers very fast. However, usually every compiler adds some startup code to a program and this code might override the content of the register variable after a reset before it can be evaluated. Thus it is necessary to move the register content to the SRAM before the startup code is executed. In addition the SRAM variable mustn't be initialized by the startup code (e.g. the GNU C-compiler sets by default all not initialized variables to zero in the startup code). Look at the ATmega164P source code to get an example how this can be implemented with the GNU C-compiler (*watchdog_reset.c*).

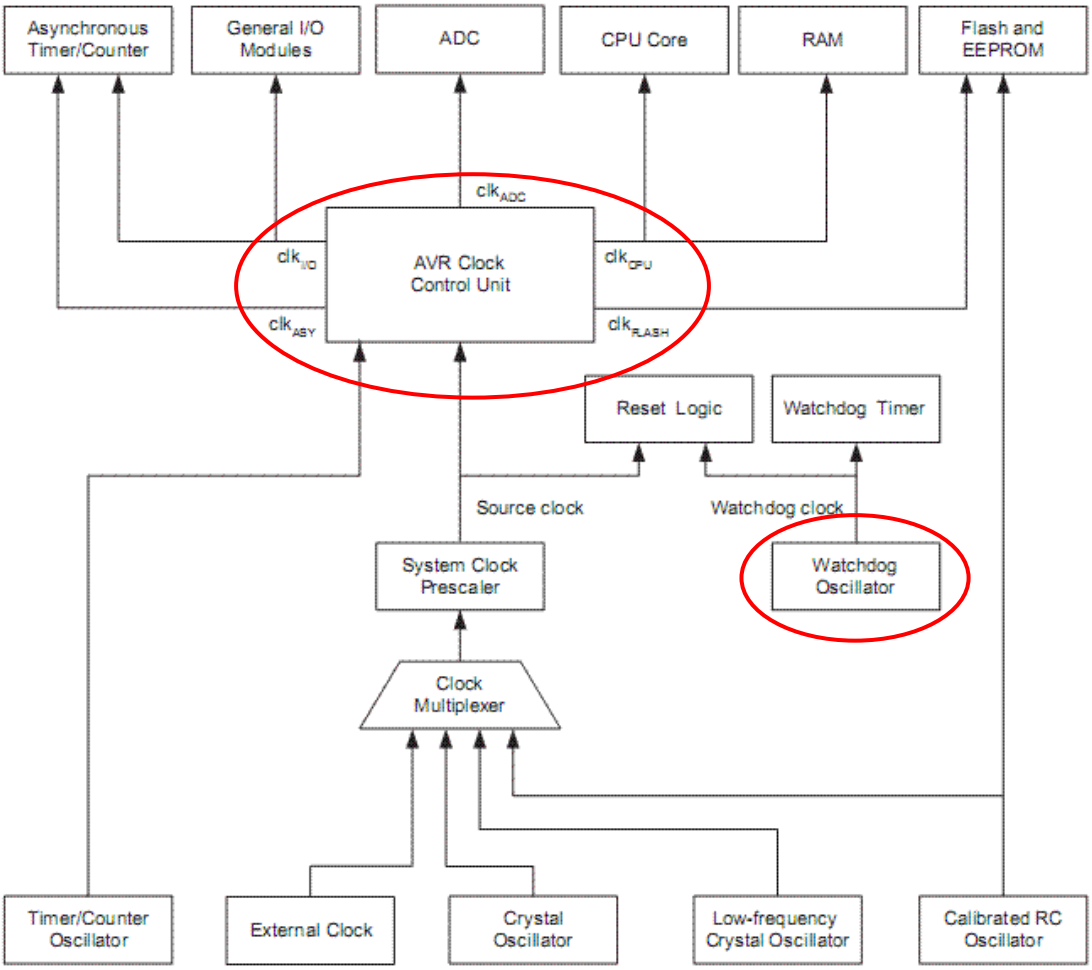
The shortest selectable watchdog timeout in the ATmega164P is 16ms. With the brownout detector enabled the controller can be configured to directly restart

© 2007 Sebastian Sohn

execution of the program with a delay of only 14 main clock cycles after the watchdog reset. Thus it is possible to operate one loop in nearly 16ms.

2.3. Watchdog interrupt

The following schematic is taken from the ATmega164P datasheet:

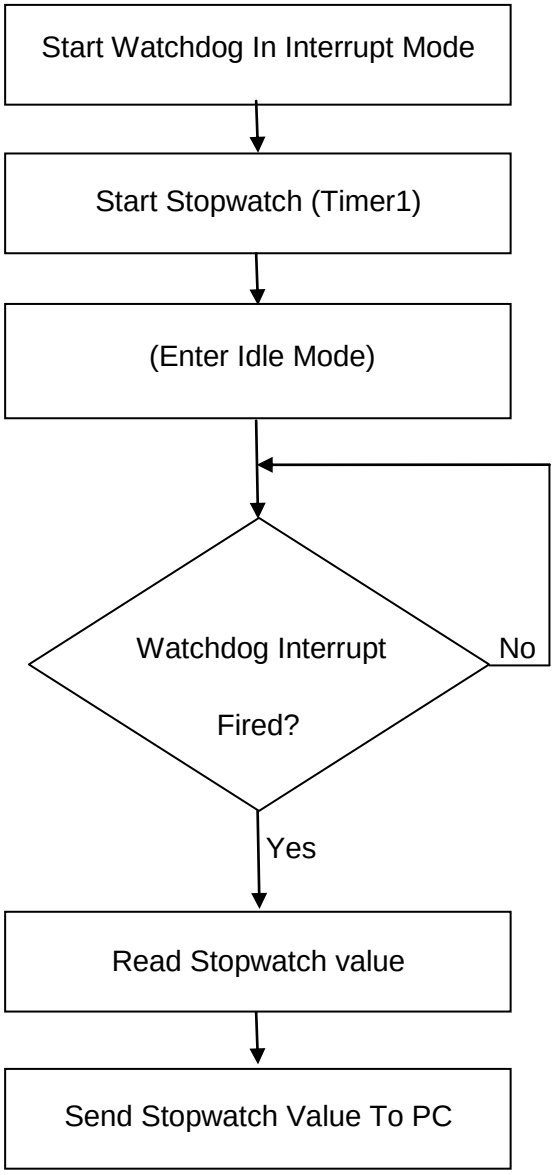


The watchdog of the ATmega164P has a special feature. It can operate in interrupt mode. In this mode the watchdog doesn't reset the microcontroller. It simply triggers an interrupt – no special function register is reset to its default value as a result of the interrupt. The watchdog is still clocked by the internal 128 kHz oscillator which is independent from the main clock source (9.216 MHz crystal). Thus there will always be a randomly varying time difference of a few main clock cycles between any two watchdog interrupts.

A timer clocked by the 9.216 MHz crystal and the watchdog are started simultaneous. The timer value is read every time the watchdog interrupt occurs and send via RS232 interface. Because of the phase shift between both clocks, the timer value will jitter. This jitter can be used to generate random numbers. Look at the ATmega164P source code for details (*watchdog_interrupt.c*).

© 2007 Sebastian Sohn

2.3.1. Flowchart



Entering Idle sleep mode is not necessary but avoids effects which are described in chapter 2.8. To only test the random source qualities of the watchdog interrupt it was necessary to switch off other side effects.

For the ATmega164P the shortest watchdog timer interrupt interval is 16ms.

© 2007 Sebastian Sohn

2.4. RS232 interface

For RS232 communication I have chosen a 9.216MHz crystal as main clock. This frequency enables standard baud rates with an error of 0.0%. A 16-bit timer of the ATmega164P can be set up to be always running at 9.216MHz. Every time new data is received, the actual value of the 16-bit timer is read and is sent via RS232 interface to the PC. The RS232 interface works asynchronously, so the exact timer value isn't predictable. Look at the ATmega164P source code for details (*RS232.c*).

The trigger could be generated via other interfaces, too (e.g. digital/analog input, LIN/CAN interface etc).

2.5. SPI

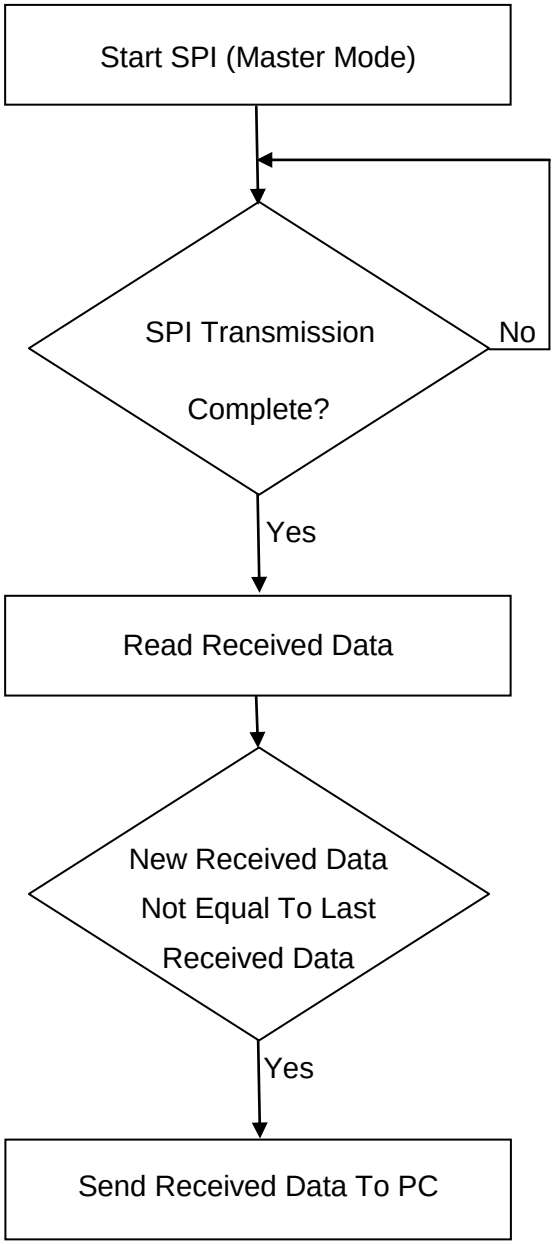
The ATmega164P has a **S**erial **P**eripheral **I**nterface (SPI). The SPI works as a simple shift register and is a standard peripheral for most microcontrollers.

To generate random numbers via SPI, an approximately 1.2 MHz external clock was connected to the SPI input pin. The SPI was configured as master and sampled the external clock at a different rate of 2.25 kHz.

The external and the SPI clock must be independent. This will ensure there is not a phase-lock between the two clocks. Because of the lack of phase-lock, it isn't predictable which value the SPI will read. Look at the ATmega164P source code for details (*spi.c*).

© 2007 Sebastian Sohn

2.5.1. Flowchart

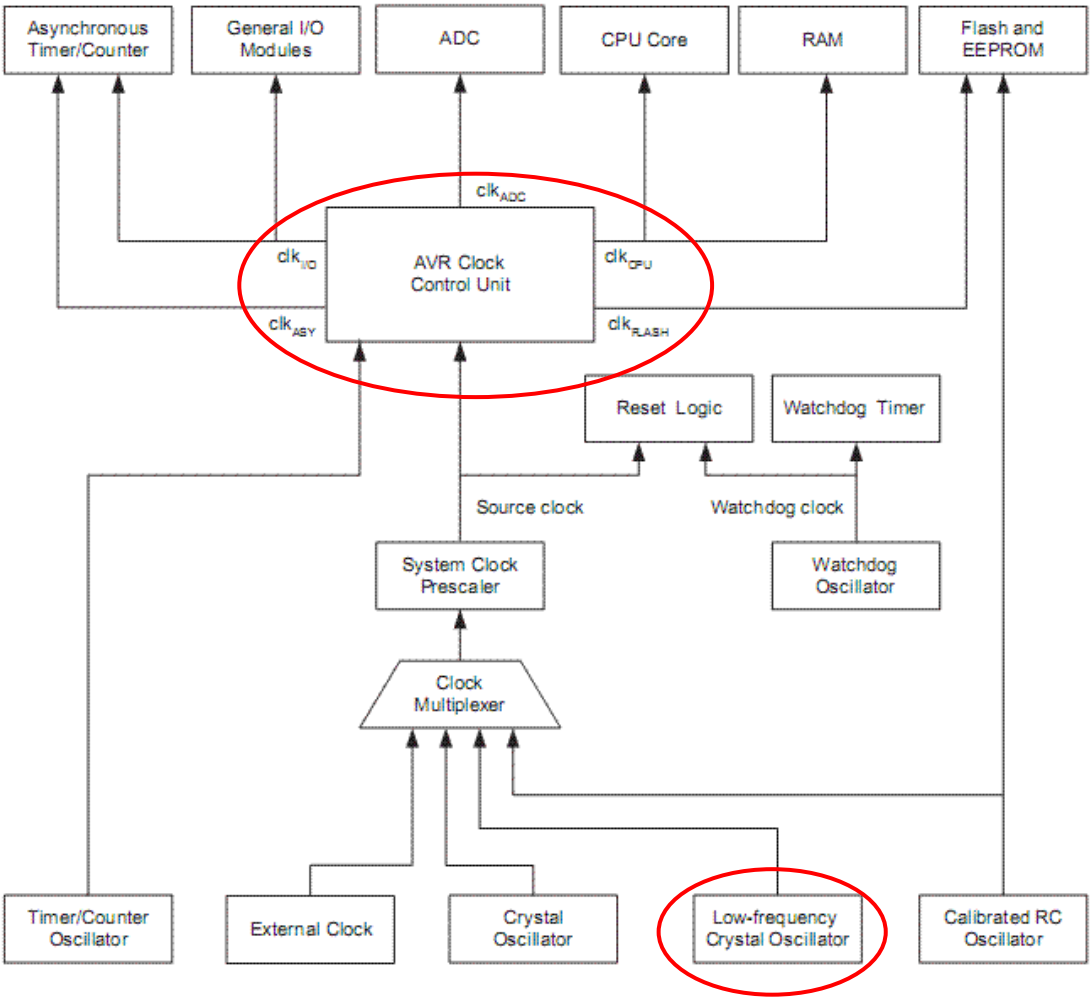


During the tests I recognized that the SPI often received the same value in series. So I added the condition to only accept received data if it is not equal to the last received data. The accepted consecutive data still had similar bit patterns. This method seems not to be optimal for random number generation.

© 2007 Sebastian Sohn

2.6. Low frequency crystal

The following schematic is taken from the ATmega164P datasheet:

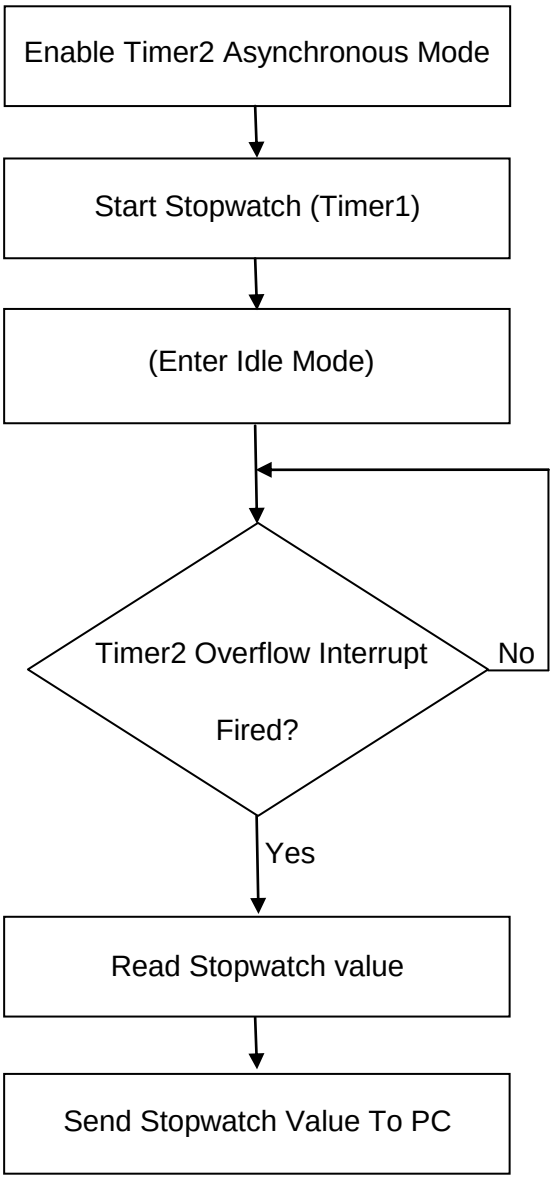


Timer2 of the ATmeaga164P can run in asynchronous mode. A 32 kHz crystal is used as clock source in this mode. Only timer2 uses the 32 kHz clock source – all other parts of the microcontroller run independent off this clock. Thus it is possible to run two timers with independent clocks (32kHz and 9.216MHz crystal)

An interrupt is triggered whenever Timer2 overflows and the value of the other timer is send via RS232 interface. There is always a phase difference between two independent clocks, thus the read timer value will vary. This varying value can be used to generate random numbers. Look at the ATmega164P source code for details (*low_freq_crystal.c*).

© 2007 Sebastian Sohn

2.6.1. Flowchart



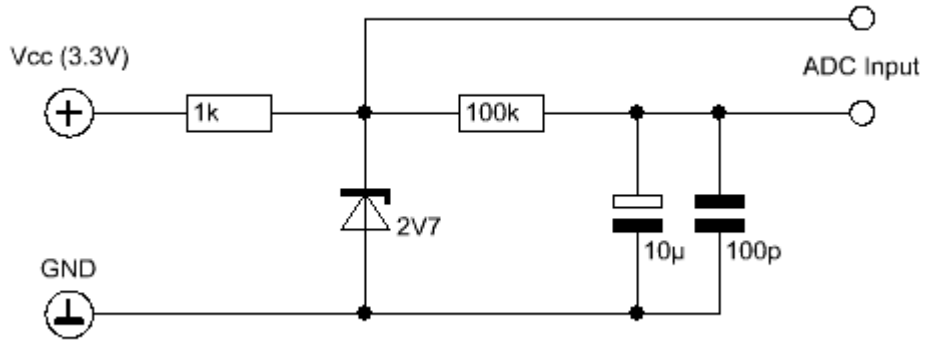
Entering Idle sleep mode is not necessary but avoids effects which are described in chapter 2.8. To only test the random source qualities of the asynchronous timer mode it was necessary to switch off other side effects.

Timer2 overflow interrupt is fired every 7.8ms (32768Hz / 256)

© 2007 Sebastian Sohn

2.7. ADC

The ATmega164P has a build-in 10-bit **A**nalog to **D**igital **C**onverter (ADC) with a selectable 200x gain stage. The gain stage is available in differential input mode. A zener diode always generates several µV of noise. The noise can be amplified via the build in 200x gain stage and measured by the ADC. Look at the ATmega164P source code for details (*adc.c*).



The 100kΩ resistor and the two capacitors eliminate the noise on one ADC input pin (low pass filter); only the DC zener voltage is available. The unfiltered zener voltage is connected to the other ADC pin. The ADC measures only the differential voltage between the two input pins which, in this case is the noise generated by the zener diode. This noise is measured by the build in ADC and send to a PC. However, the gain stage of the ADC does not have perfect inputs with infinite input resistance. Thus a small current does flow into the gain stage and an offset is added to the noise voltage. As long the offset does not saturate the ADC, this is not a problem in this application.

2.8. Timer ISR

Most instructions of the AVR core are executed in 1 or 2 cycles. But some instructions even take 3, 4 or 5 cycles. Here is a quote from the ATmega164P datasheet chapter “6.7.1. Interrupt Response Time”

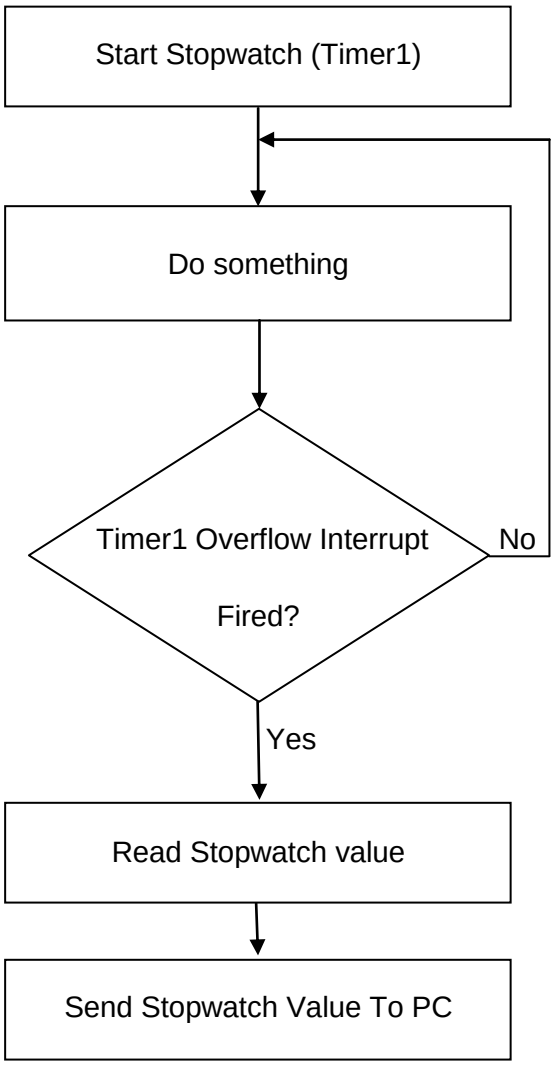
“If an interrupt occurs during execution of a multi-cycle instruction, this instruction is completed before the interrupt is served.”

The meaning of this quote is that the interrupt response time is not fixed. It depends on the actual executed instruction. If an interrupt is triggered during execution of a 5 cycle instruction, the jump to the **i**nterrupt **s**ervice **r**outine (ISR) could be delayed up to 4 clock cycles.

If a timer is configured to cyclically trigger an interrupt, the corresponding ISR is not executed at a fixed interval. The interval varies depending on the instruction which was executed when the interrupt was triggered. This varying delay can be measured and be used to generate random numbers. Look at the ATmega164P source code for details (*timer_isr.c*).

© 2007 Sebastian Sohn

2.8.1. Flowchart



It is important that the controller is not in any sleep mode when Timer1 interrupt is fired. The wake up time from a sleep mode is always equal.

Timer1 overflows every 3.6ms with a 9.216MHz clock source.

2.9. Input capture unit - TBD

The ATmega164P has a build in input capture unit. This periphery is used to measure frequencies and the duty cycle of square waves. It can be configured to store the actual value of Timer1 on a rising/falling edge on the input capture pin or the analog comparator input.

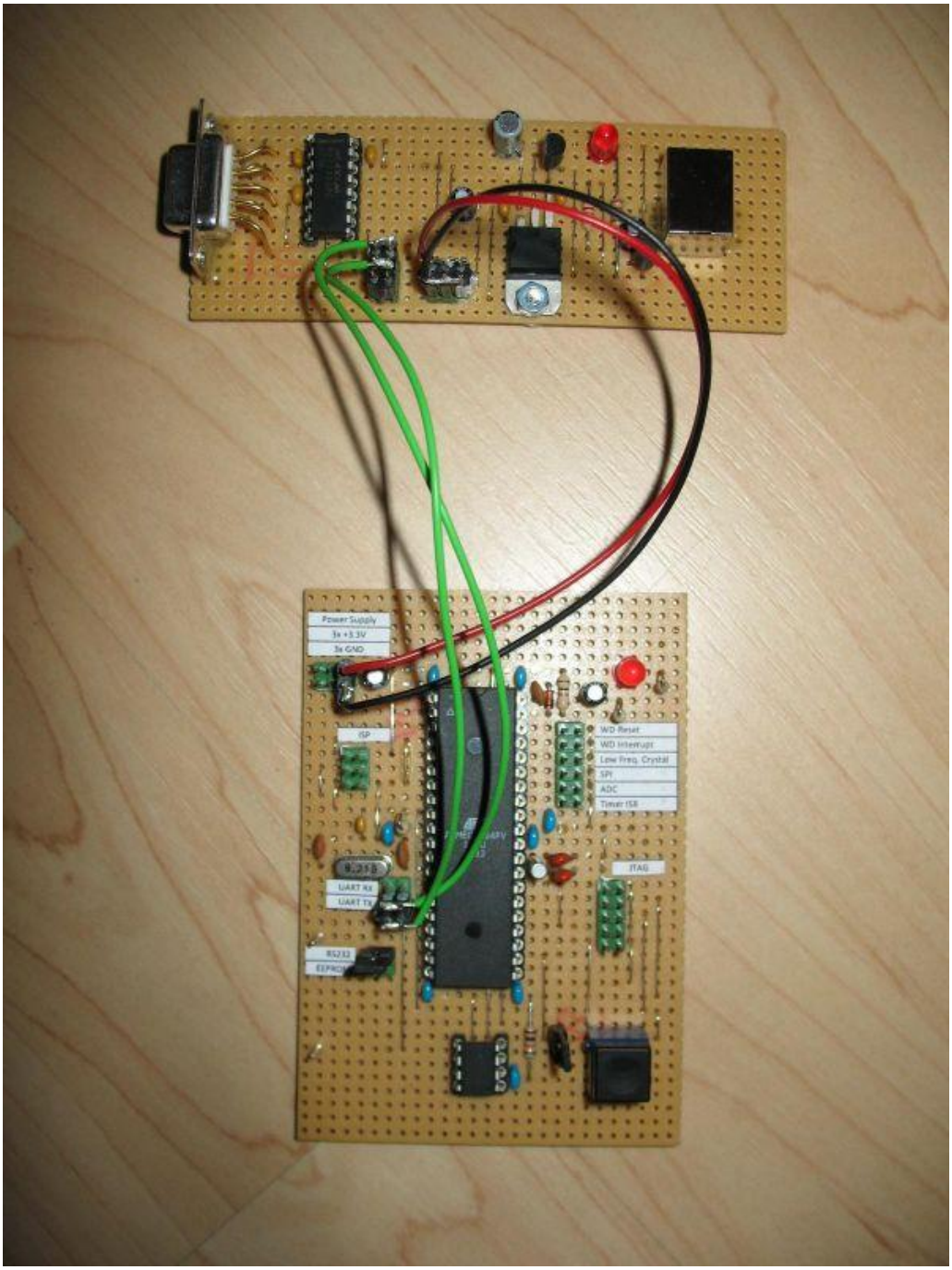
If an external signal is measured via the input capture unit, the captured Timer1 value might vary. This variance could be a random number source. I had this idea after I had finished my tests, so I only add a note here to avoid losing the idea.

© 2007 Sebastian Sohn

3. Hardware

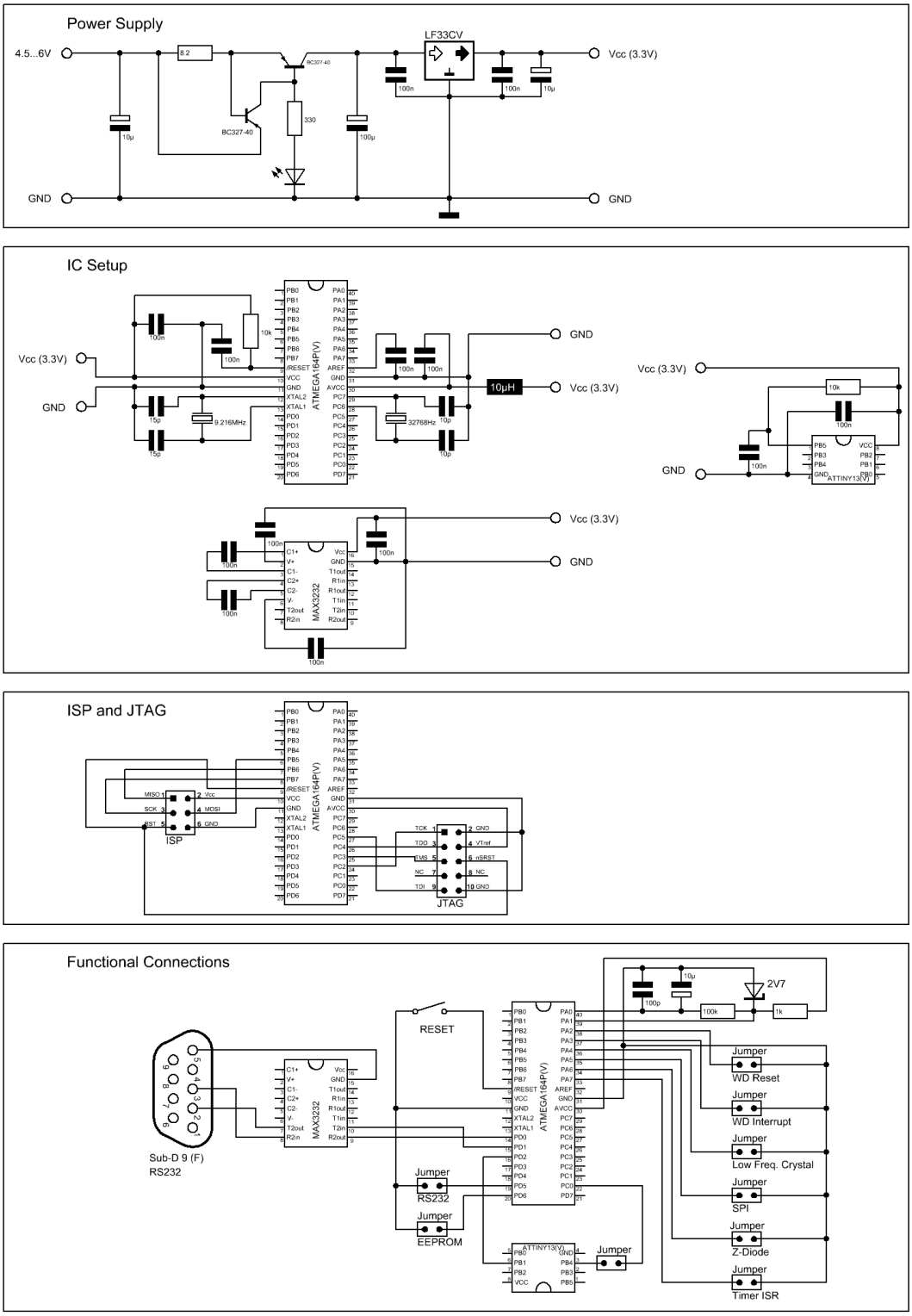
The hardware contains of two boards. The small board contains a 3.3V voltage regulator (LF33CV) and an RS232 level shifter (MAX3232). The big board contains an ATmega164PV which is used as random number generator with its necessary peripheral (9.216 MHz crystal, 32 kHz crystal, zener diode circuit, jumpers for operation mode selection and a button connected to the RESET pin of the ATmega164PV).

In addition the big board contains an ATtiny13V. The ATtiny13V generates the 1.2MHz clock which is used for random number generation via SPI and a random output which is necessary for random number generation via timer interrupt.



© 2007 Sebastian Sohn

3.1. Schematic



Note that each IC type is only used once in the complete circuit. The schematic is an attempt to split the complete circuit in logical parts to gain a clearer view.

© 2007 Sebastian Sohn

4. Number analysis tool

MS Excel is a good tool to analyze numbers and create charts. The build-in VBA IDE makes it very flexible and enabled me to program an analysis tool which checks randomness of a generated number series.

A true random source never repeats a number series and follows no mathematical algorithm. However, it would take forever to check if a number series never repeats. It is only possible to check a finite number series. I logged from each number source a series of 4 to 10 million numbers for doing my tests.

Even with finite sequences, it is very difficult to check if a number series is not the product of a mathematical algorithm (e.g. the digits of π never repeat, but they obviously can be generated from a mathematical algorithm). My tool can't determine if a number series is a product of any mathematical algorithm. If it could, I'd be rich and famous – but I'm not.

However, it is possible to check that a given sub-series never repeats in a large series of several million numbers.

The analysis tool generates 3 charts: "Number Statistics", "Number Statistics (Normalized)" and "Random Check". The number series must be stored hex formatted (no prefix like "0x") in a text file. Each number is located in a separate line.

4.1. Worksheet "Macro"

The sheet contains a button to select and check a text file which contains a number series. Some parameters for the random check macro can be set in this sheet. The chart values are stored in this sheet.

4.2. Chart "Number Statistics"

In the first step the occurrence of each logged number is counted and displayed in a chart. Stochastic procedures usually generate a Gaussian distribution. This chart is useful to determine how many bits of each generated number can be used to generate a large random number.

4.3. Chart "Number Statistics (Normalized)"

This chart contains the same information as the chart "Number Statistics", however all values are normalized. Each number is displayed both as a percentage of that number's occurrence, and the deviation of that occurrence from the number with the highest occurrence.

4.4. Chart "Random Check"

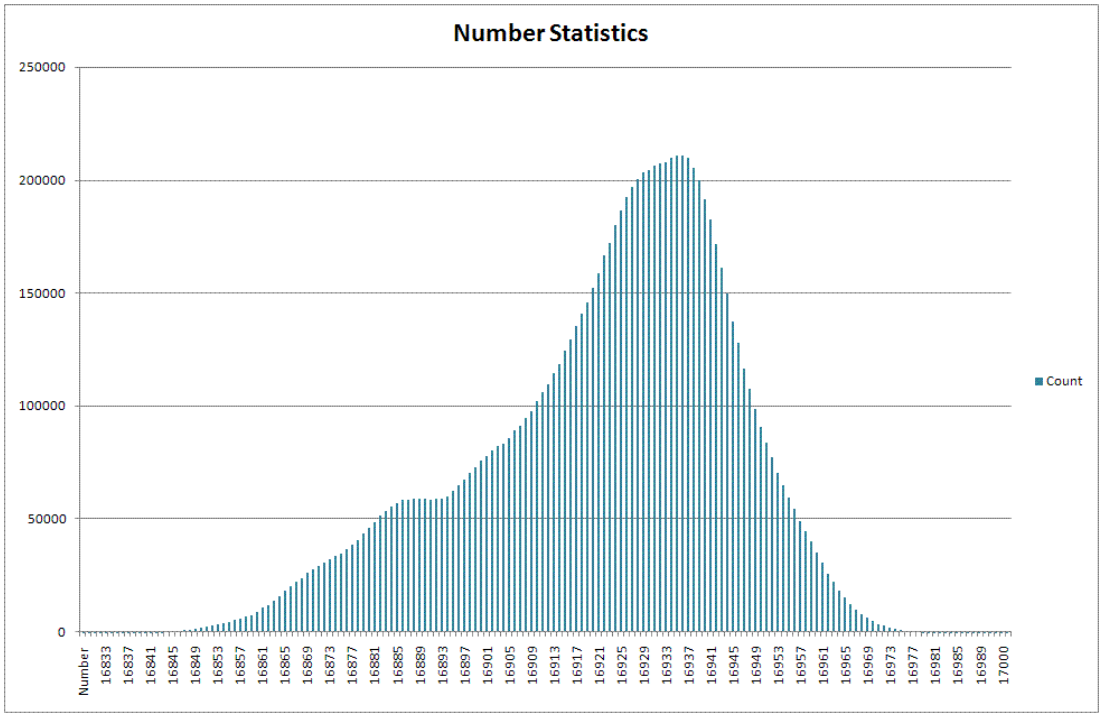
This chart shows the displacement of the occurrence of a sub series in the complete number series. If the displacement of the sub series is not constant or behaves periodically, the number series is marked as a true random number series.

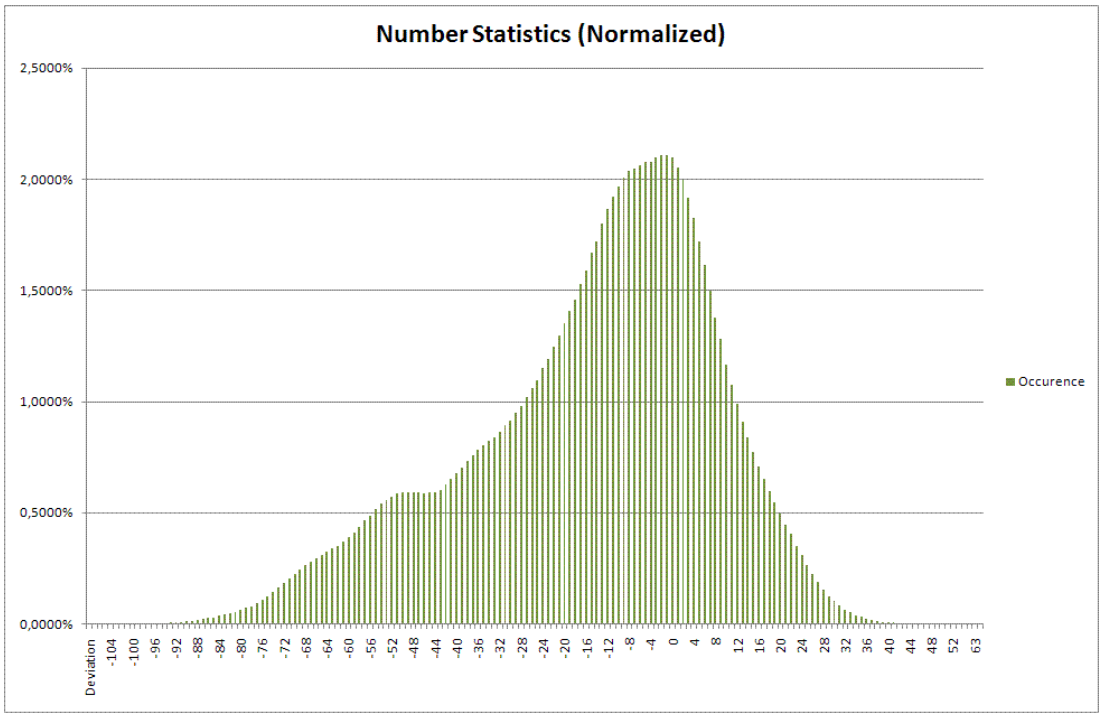
© 2007 Sebastian Sohn

5. Random check results

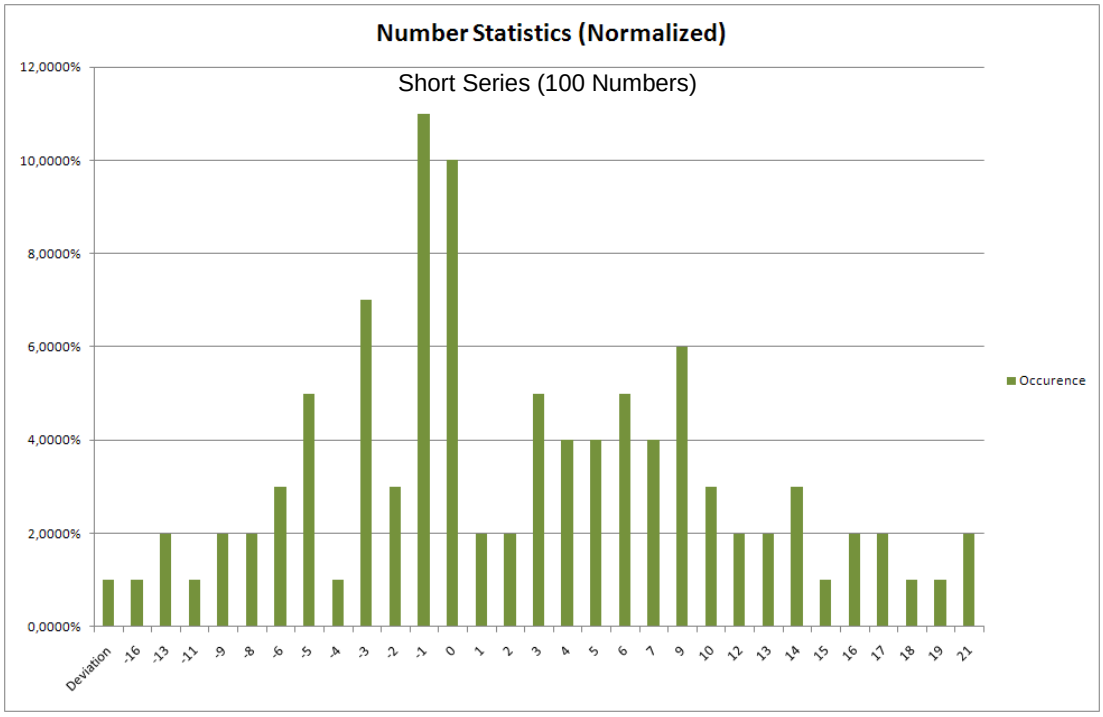
5.1. Internal EEPROM

5.1.1. Number Statistics



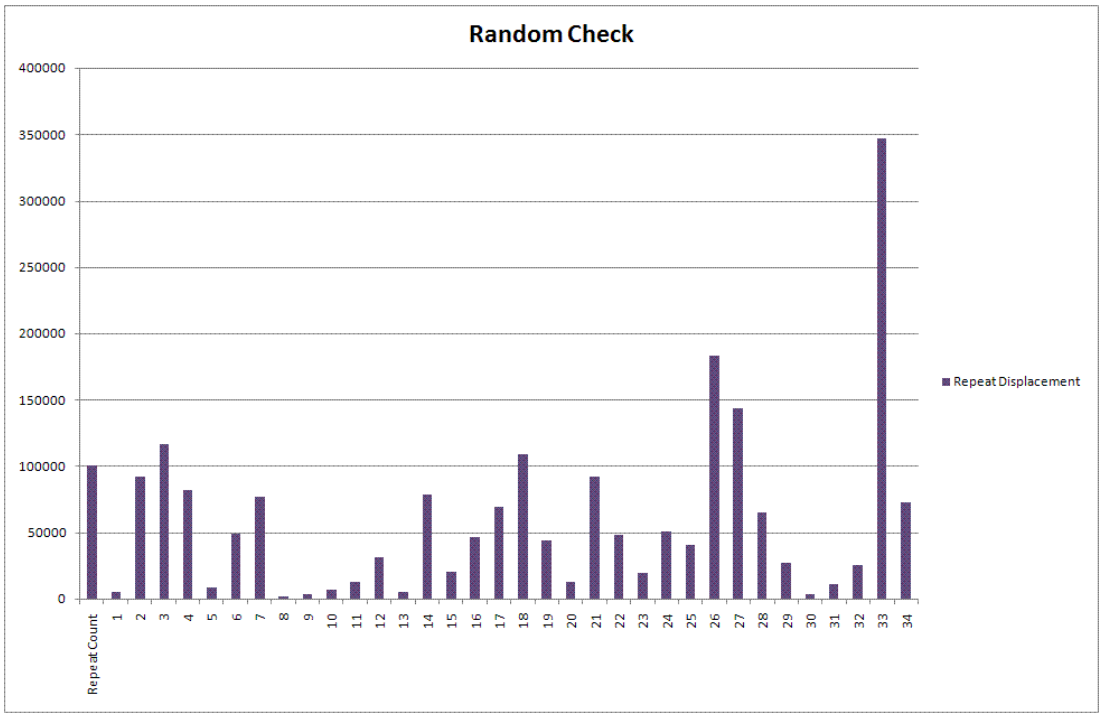


© 2007 Sebastian Sohn



A series of 10 million numbers was taken for this test. The values vary even in a short sub-series of 100 numbers. To generate nearly white noise the 4 LSBs of several numbers can be joined to a big number. It takes 8 passes to get a 32 bit random number.

5.1.2. Random Check

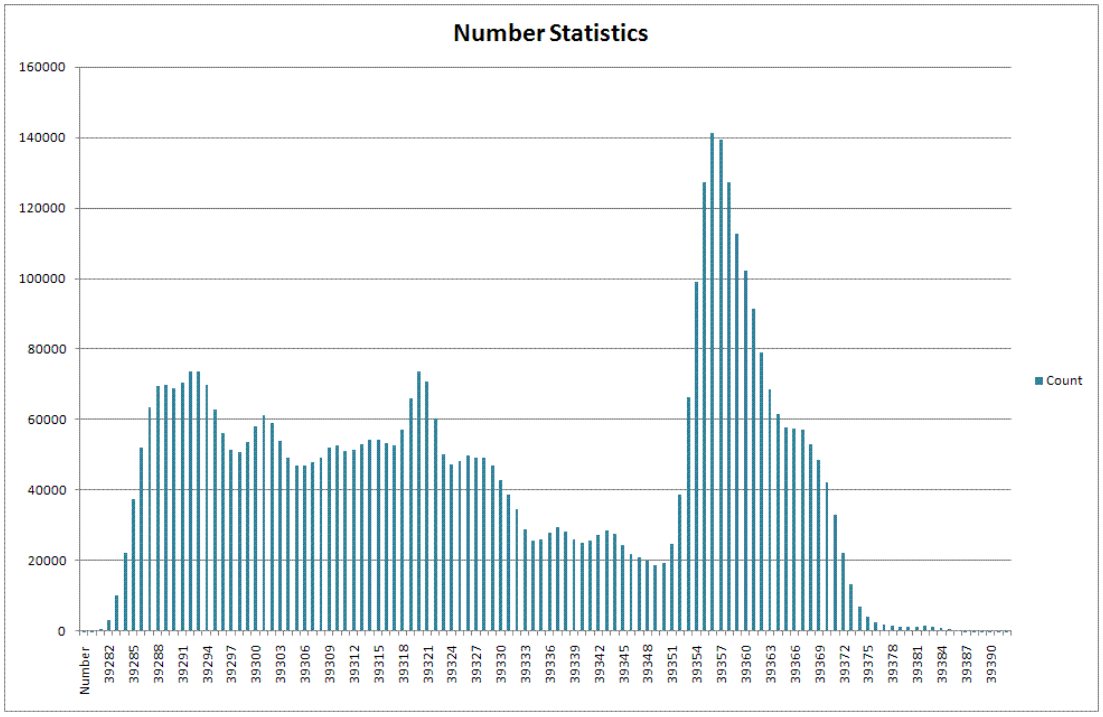


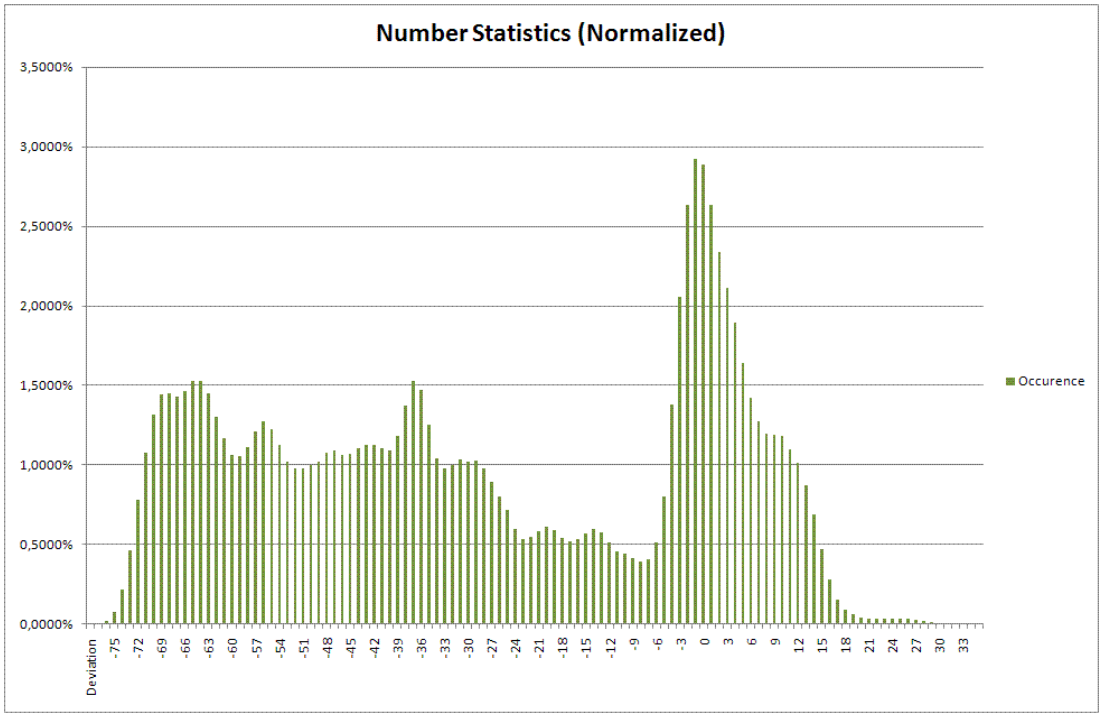
For the randomness check the sub-series “16886, 16881, 16877” was used. It occurred 35 times in the complete series. The deviation of the sub-series occurrence varies non-cyclic. The EEPROM method passes the random check.

© 2007 Sebastian Sohn

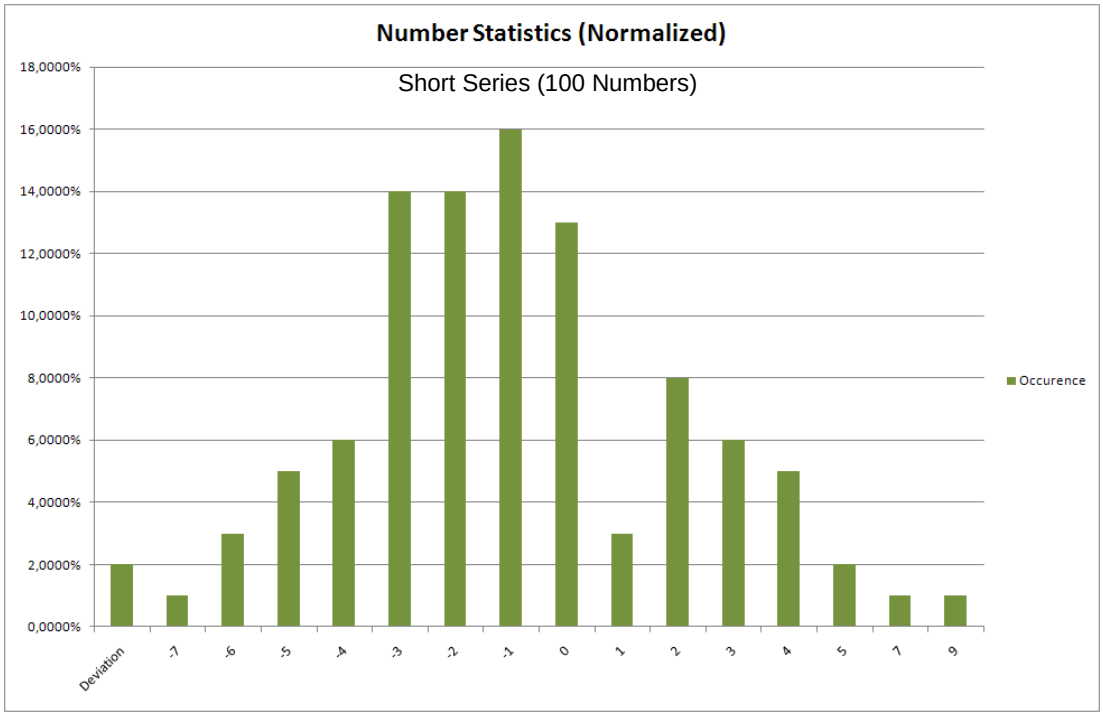
5.2. Watchdog Reset

5.2.1. Number Statistics



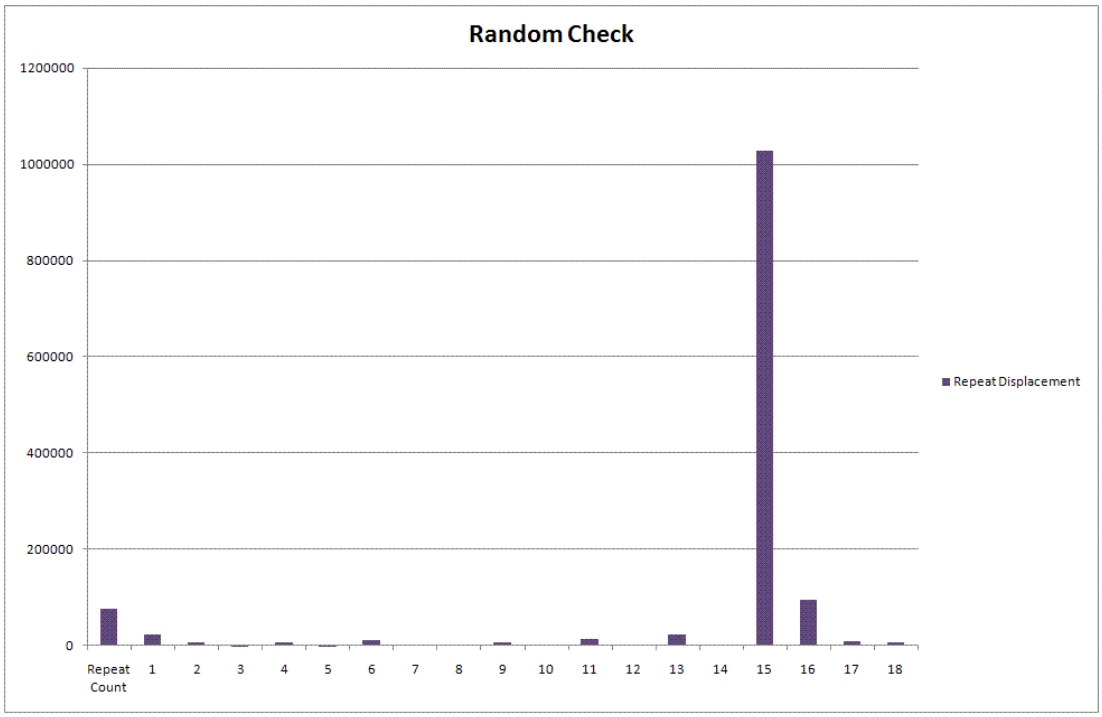


© 2007 Sebastian Sohn



A series of 4.8 million numbers was taken for this test. The values vary even in a short sub-series of 100 numbers. To generate nearly white noise the 2 LSBs of several numbers can be joined to a big number. It takes 16 passes to get a 32 bit random number.

5.2.2. Random check

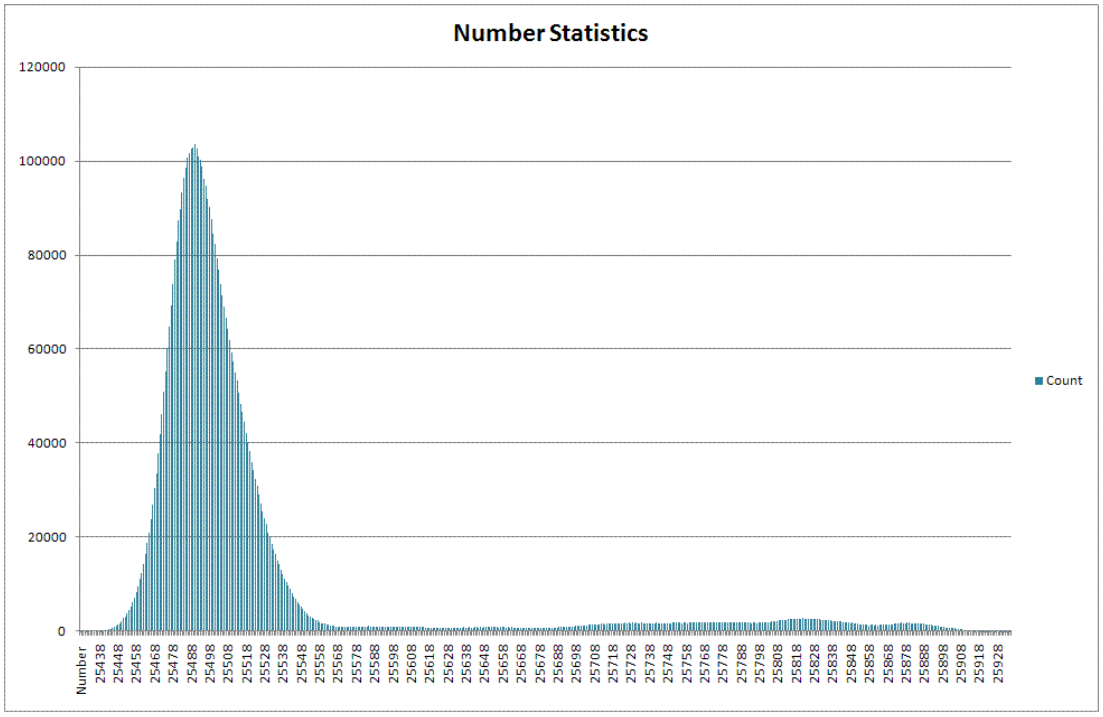


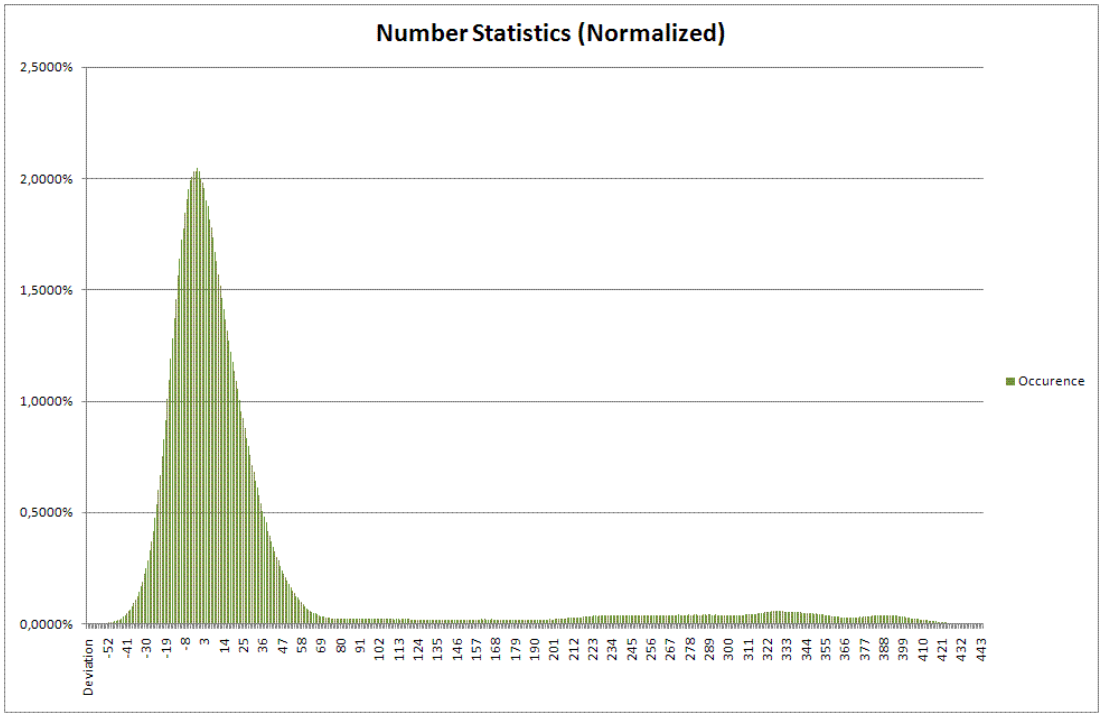
For the randomness check the sub-series “39368, 39368, 39367, 39369, 39371” was used. It occurred 19 times in the complete series. The deviation of the sub-series occurrence varies non-cyclic. The watchdog reset method passes the random check.

© 2007 Sebastian Sohn

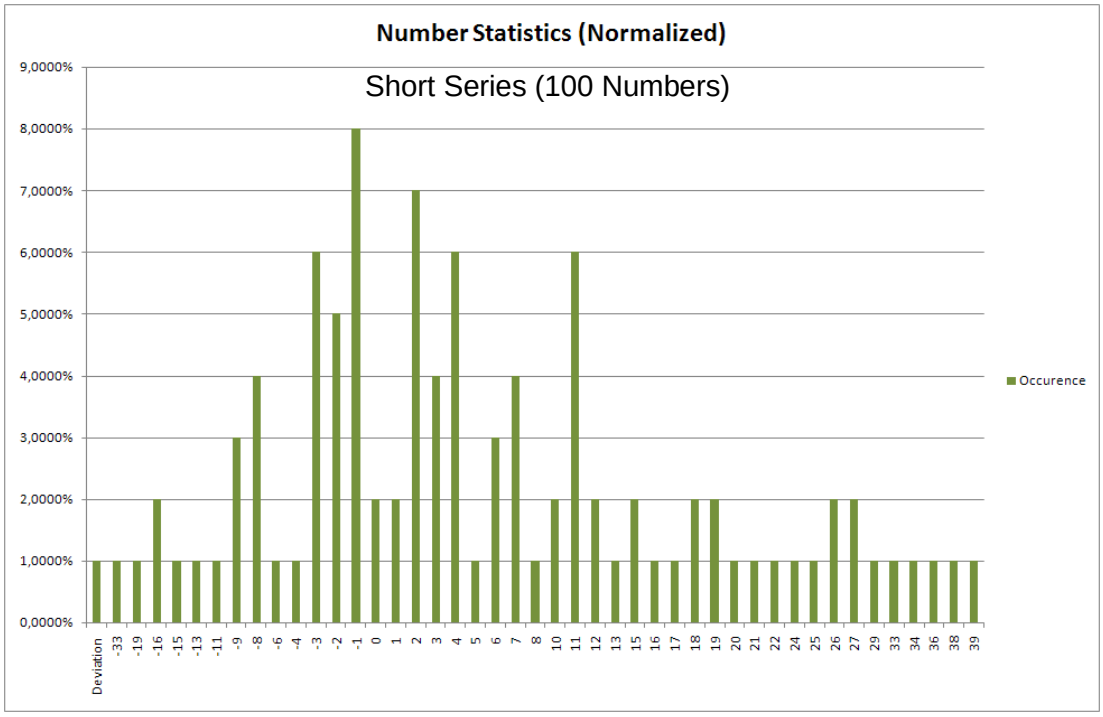
5.3. Watchdog Interrupt

5.3.1. Number Statistics



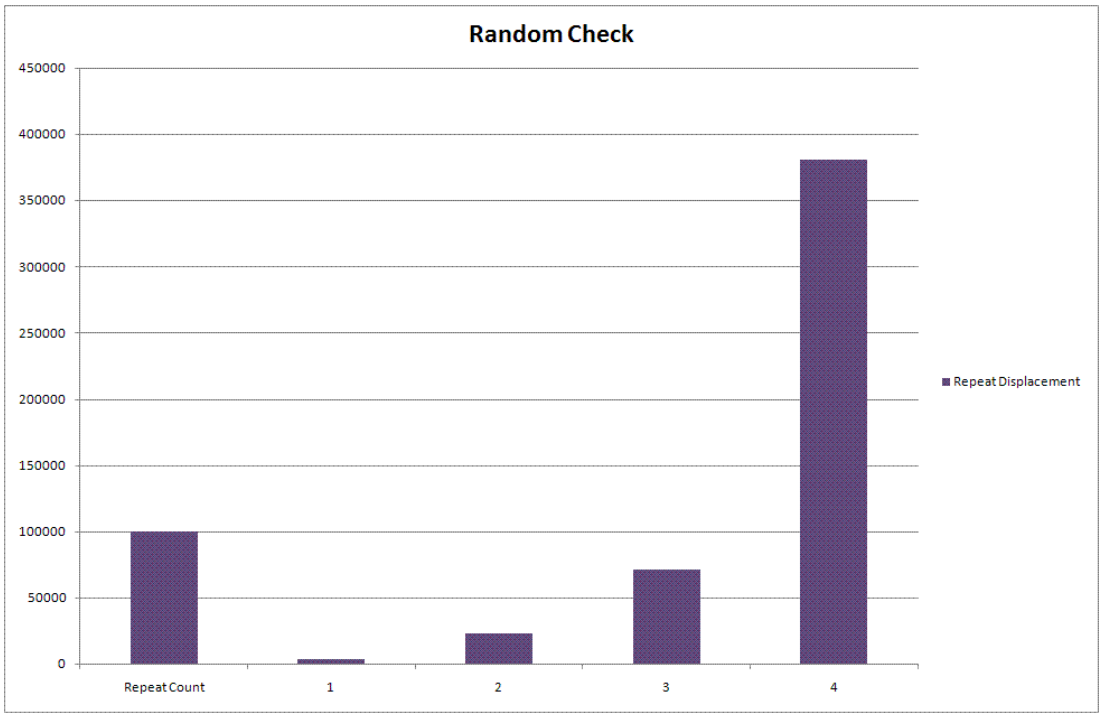


© 2007 Sebastian Sohn



A series of 5 million numbers was taken for this test. The values vary even in a short sub-series of 100 numbers. To generate nearly white noise the 4 LSBs of several numbers can be joined to a big number. It takes 8 passes to get a 32 bit random number.

5.3.2. Random Check

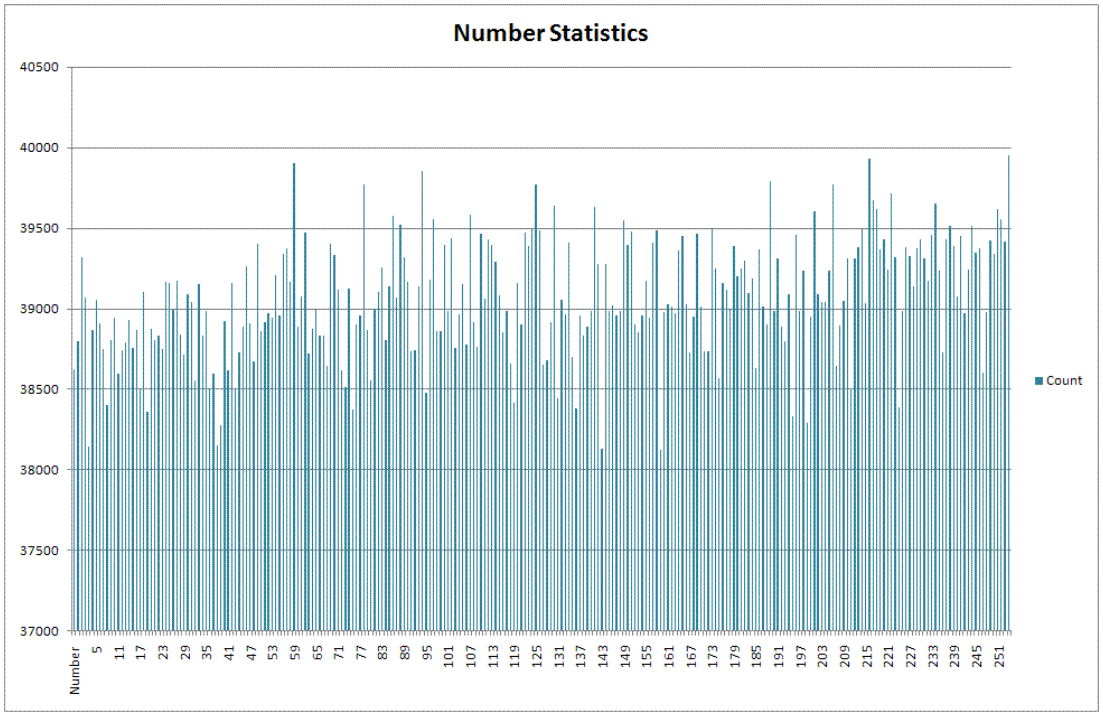


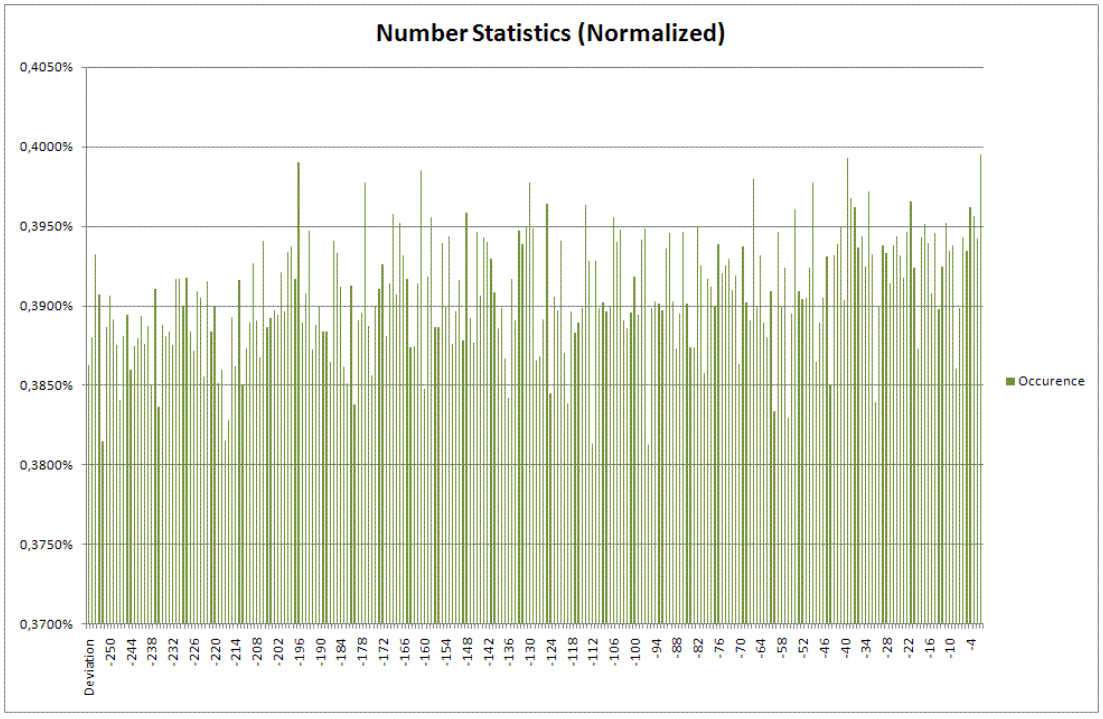
For the randomness check the sub-series “25540, 25535, 25531” was used. It occurred 5 times in the complete series. The deviation of the sub-series occurrence varies non-cyclic.

© 2007 Sebastian Sohn

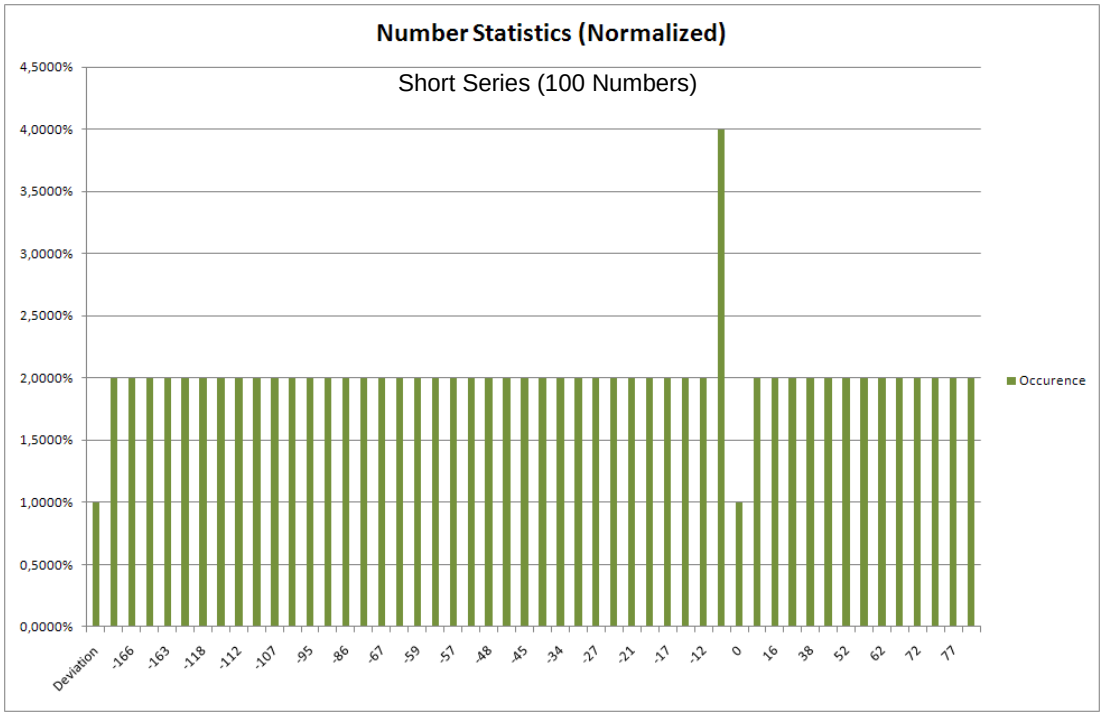
5.4. SPI

5.4.1. Number Statistics



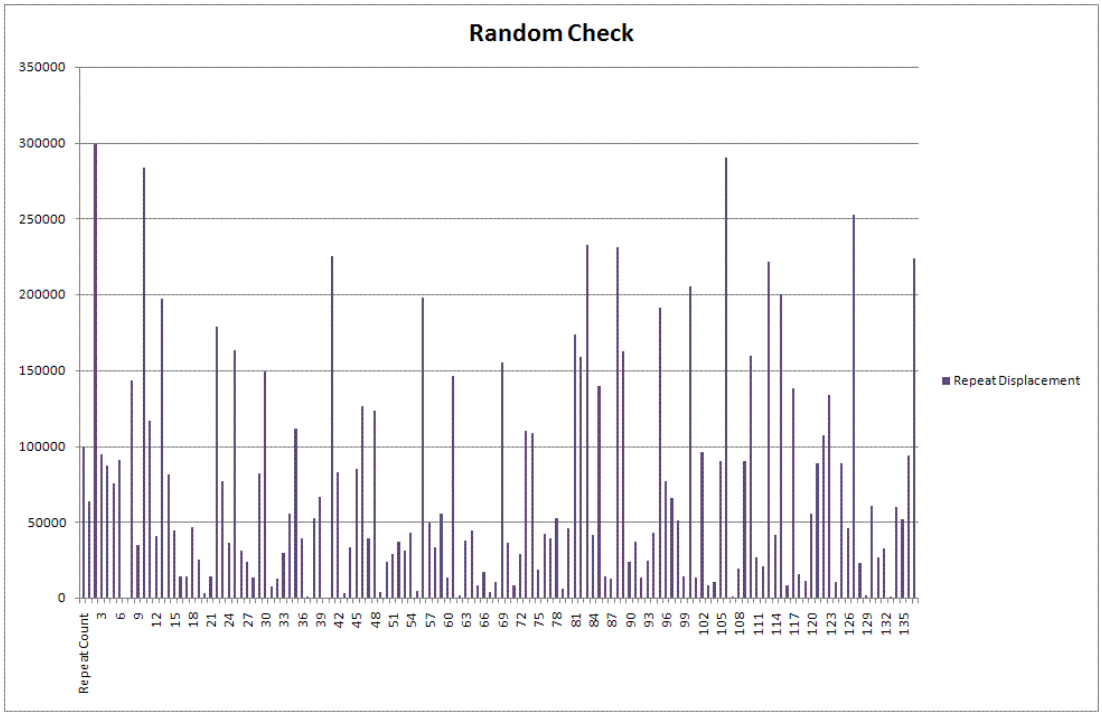


© 2007 Sebastian Sohn



A series of 10 million numbers was taken for this test. The occurrence of each number is almost equal. The values vary in a wide range even in a short numbers series. The result looks like white noise. All bits of a generated number can be used. It takes 4 passes to get a 32 bit random number. However, series of nearby numbers had similar bit patterns. This can be avoided by keeping the time displacement between generation of two numbers high (e.g. 100ms).

5.4.2. Random Check

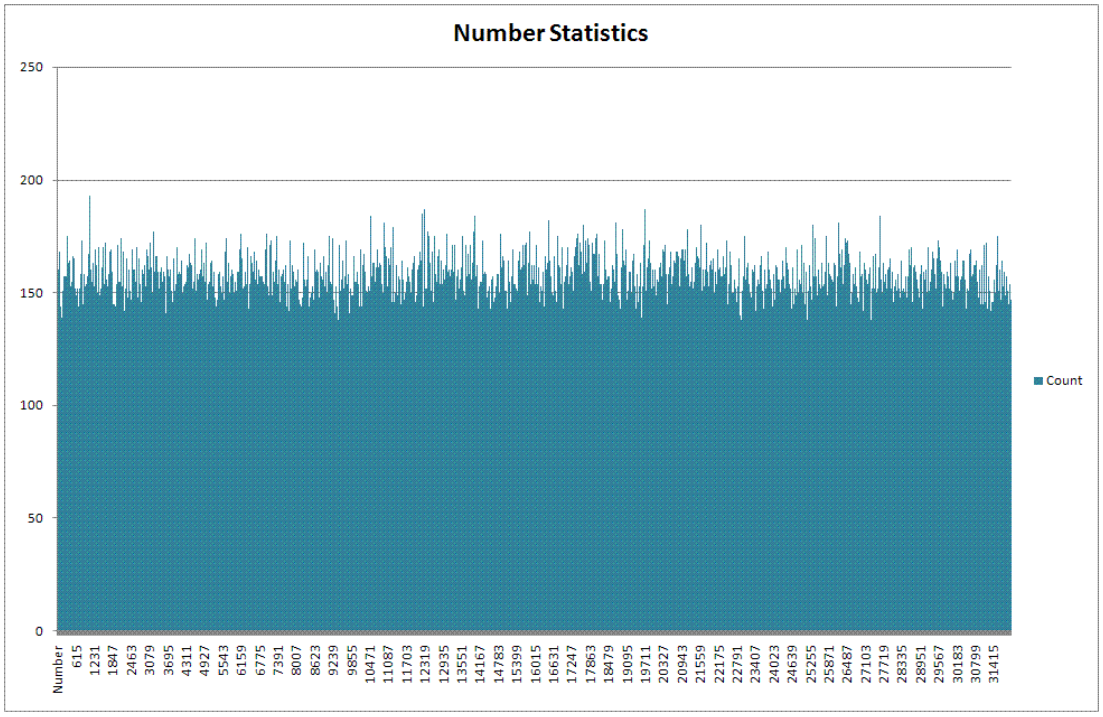


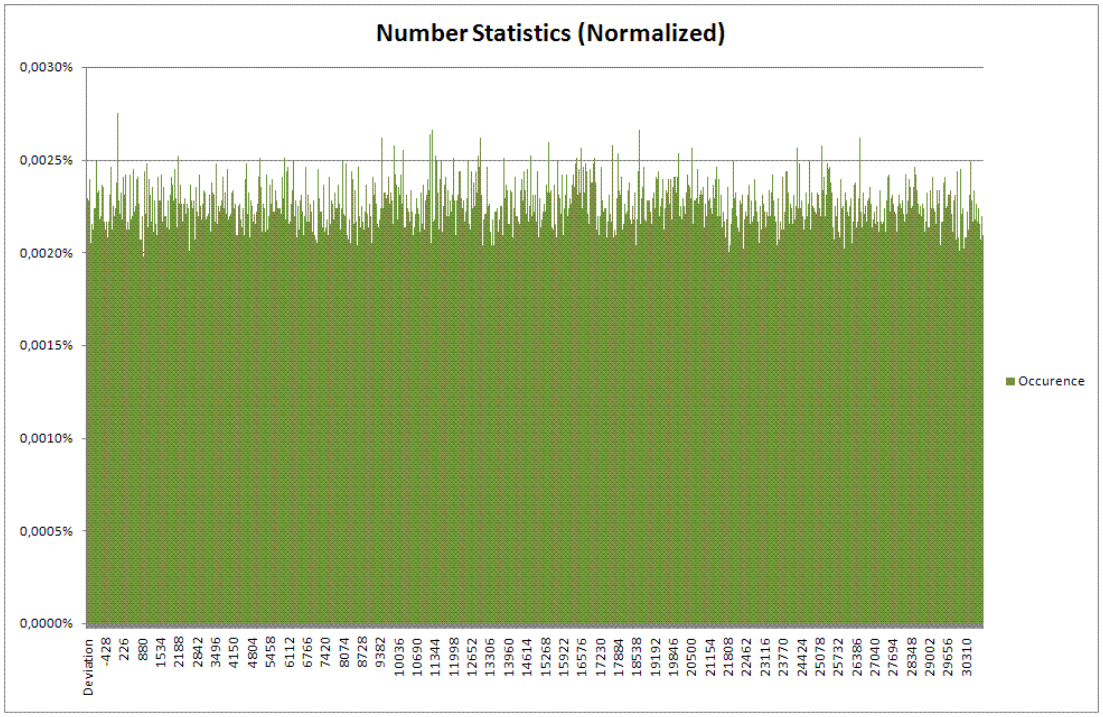
For the randomness check the sub-series “129, 154” was used. It occurred 138 times in the complete series. The deviation of the sub-series occurrence varies non-cyclic. The SPI method passes the random check.

© 2007 Sebastian Sohn

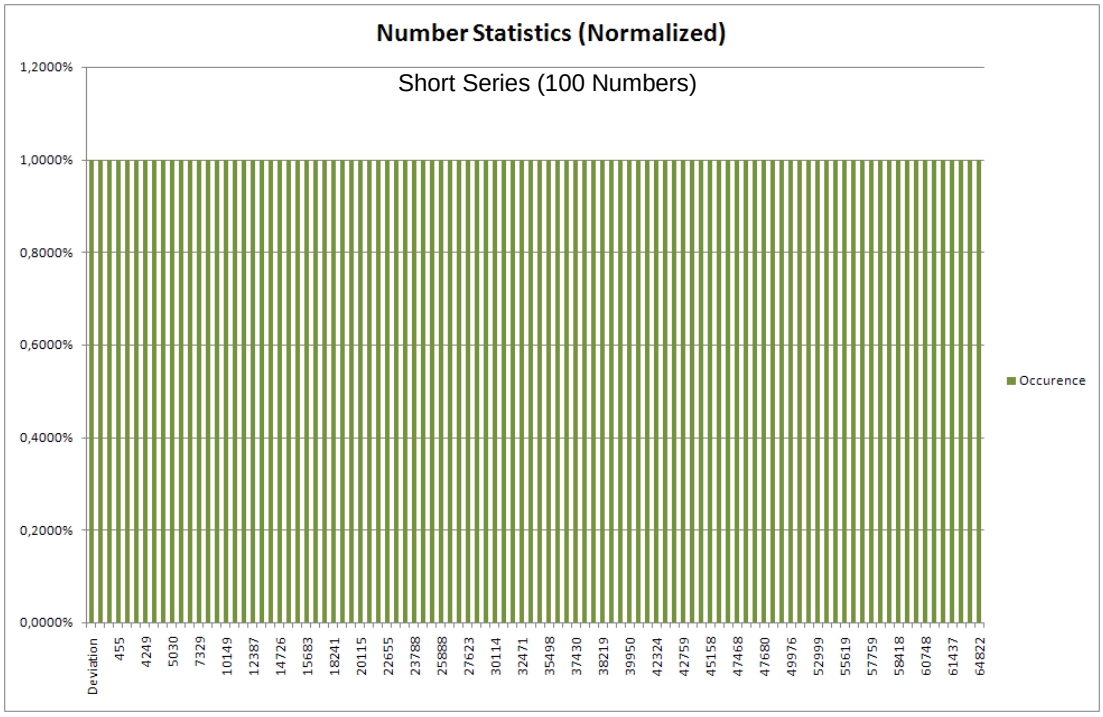
5.5. RS232 Interface

5.5.1. Number Statistics



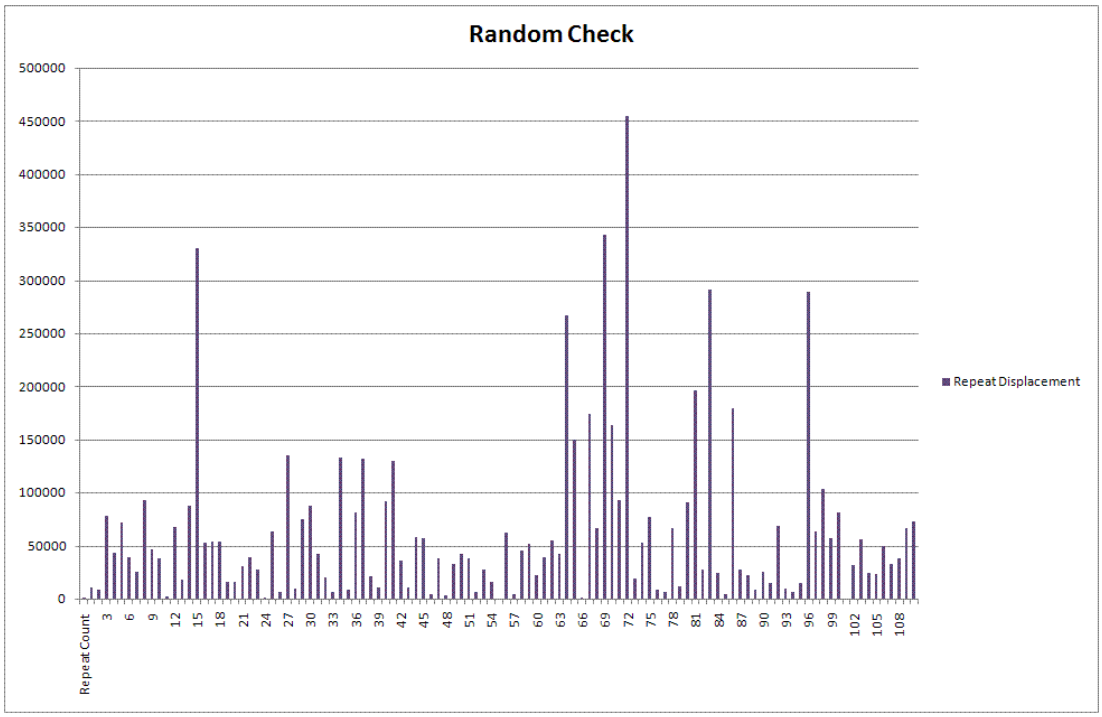


© 2007 Sebastian Sohn



A series of 7 million numbers was taken for this test. The values vary even in a short sub-series of 100 numbers. The result looks like white noise. All bits of a generated number can be used. With a 16 bit timer it takes 2 passes to get a 32 bit random value.

5.5.2. Random Check

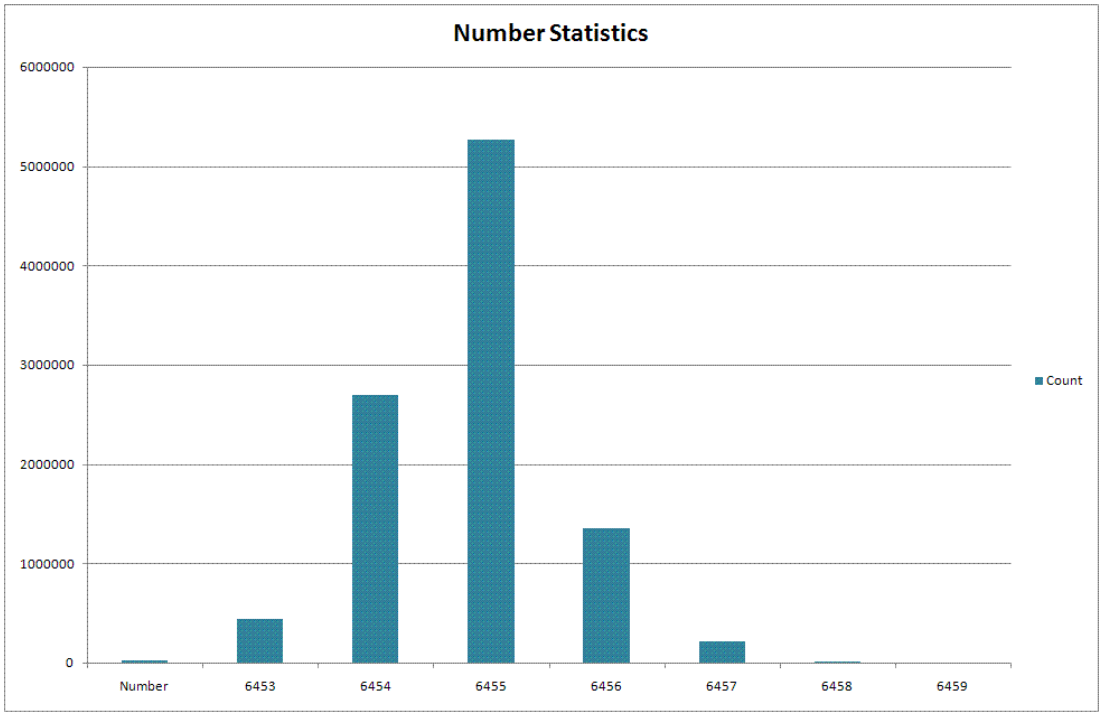


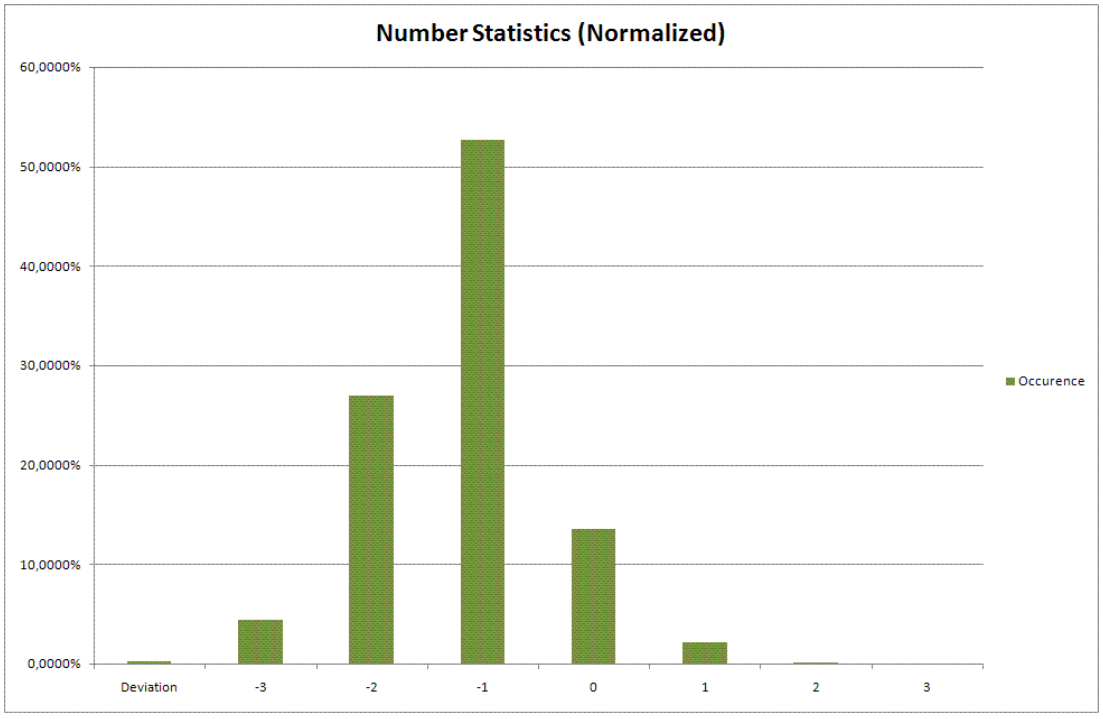
For the randomness check the sub-series “34999” was used. It occurred 111 times in the complete series. The deviation of the sub-series occurrence varies non-cyclic. The RS232 method passes the random check.

© 2007 Sebastian Sohn

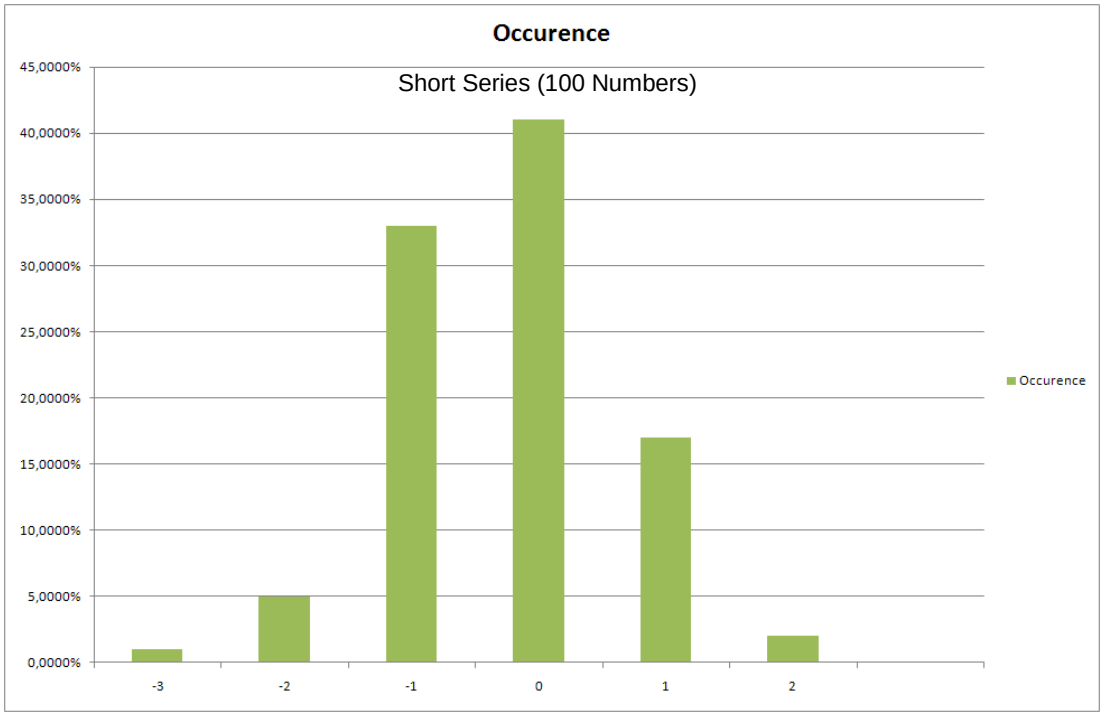
5.6. Low Frequency Crystal

5.6.1. Number Statistics



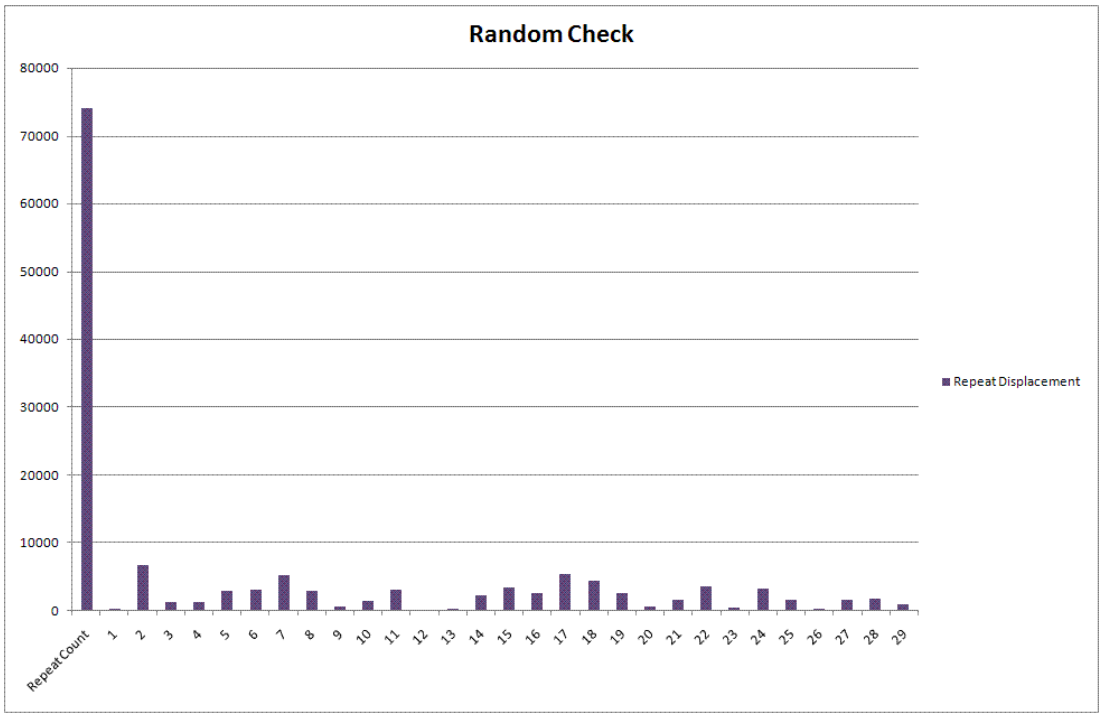


© 2007 Sebastian Sohn



A series of 10 million numbers was taken for this test. The values vary even in a short sub-series of 100 numbers. But the values vary only in a small range. Maybe because both clock sources use crystals which run at a very accurate and stable frequency. To generate a large random number the LSB of several numbers can be joined to a big number. It takes 32 passes to get a 32 bit random number.

5.6.2. Random Check

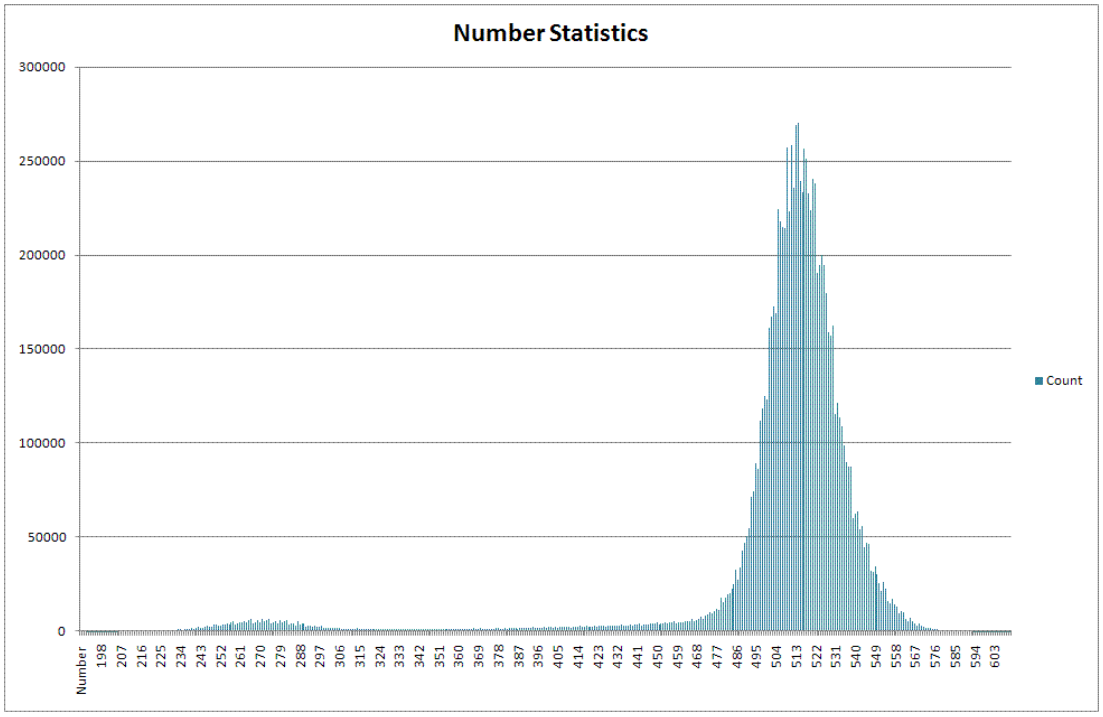


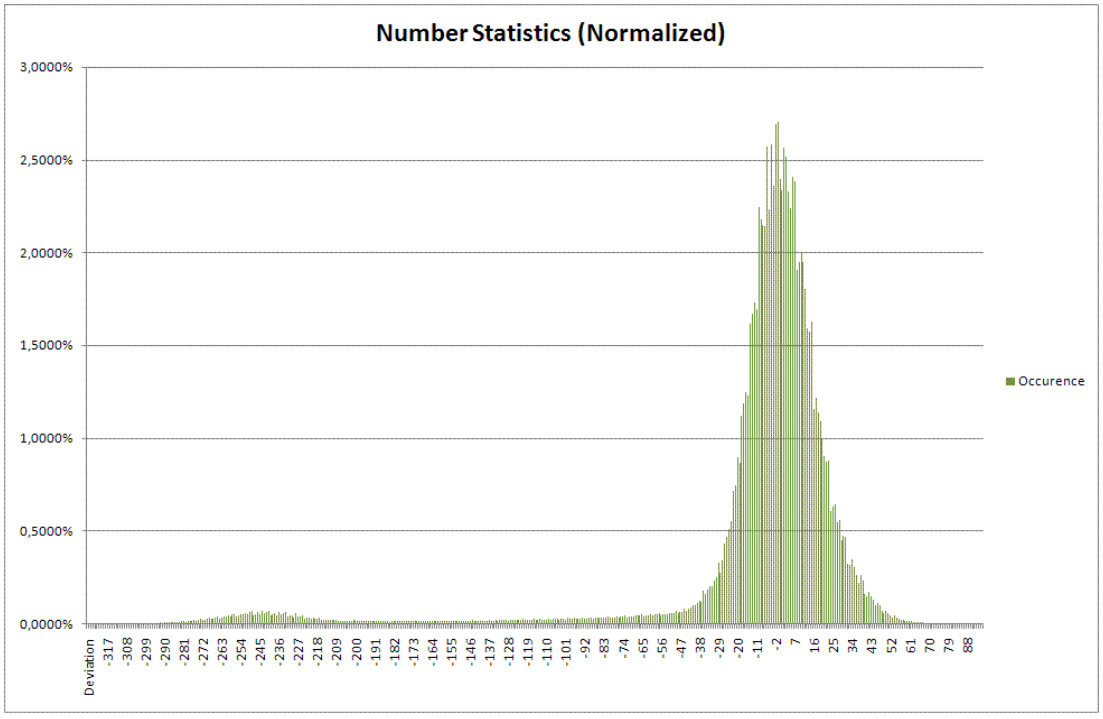
For the randomness check a sub-series of 32 numbers was used. It occurred 30 times in the complete series. The deviation of the sub-series occurrence varies non-cyclic. The low frequency crystal method passes the random check even though the values only vary in a small range.

© 2007 Sebastian Sohn

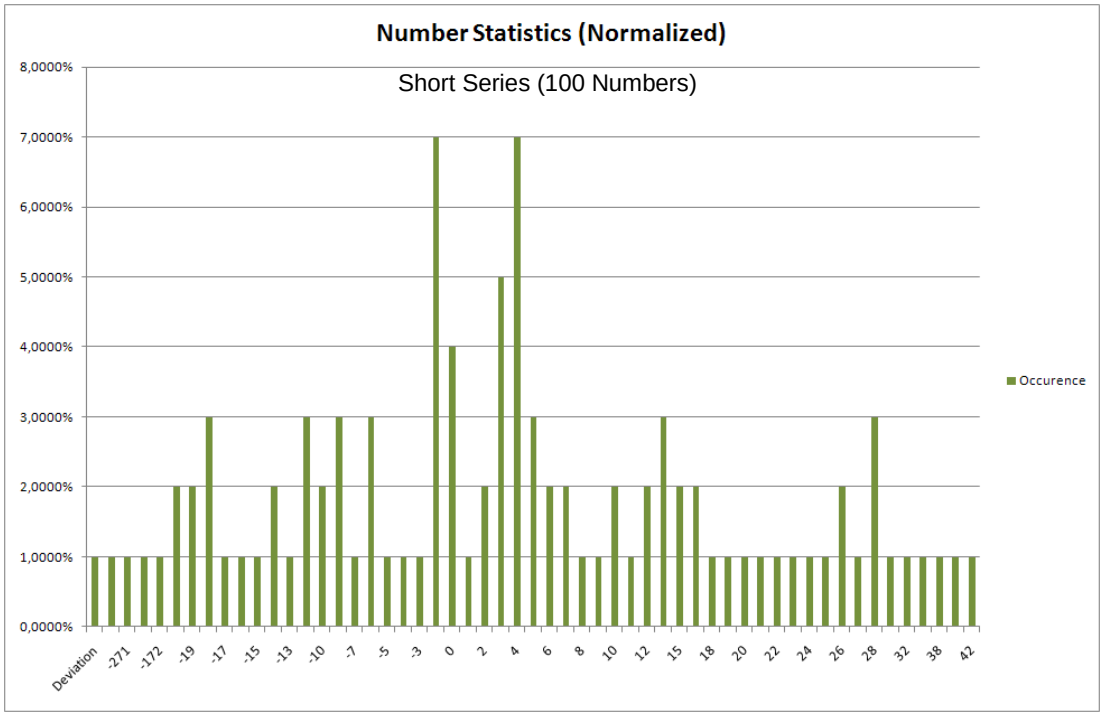
5.7. ADC

5.7.1. Number Statistics



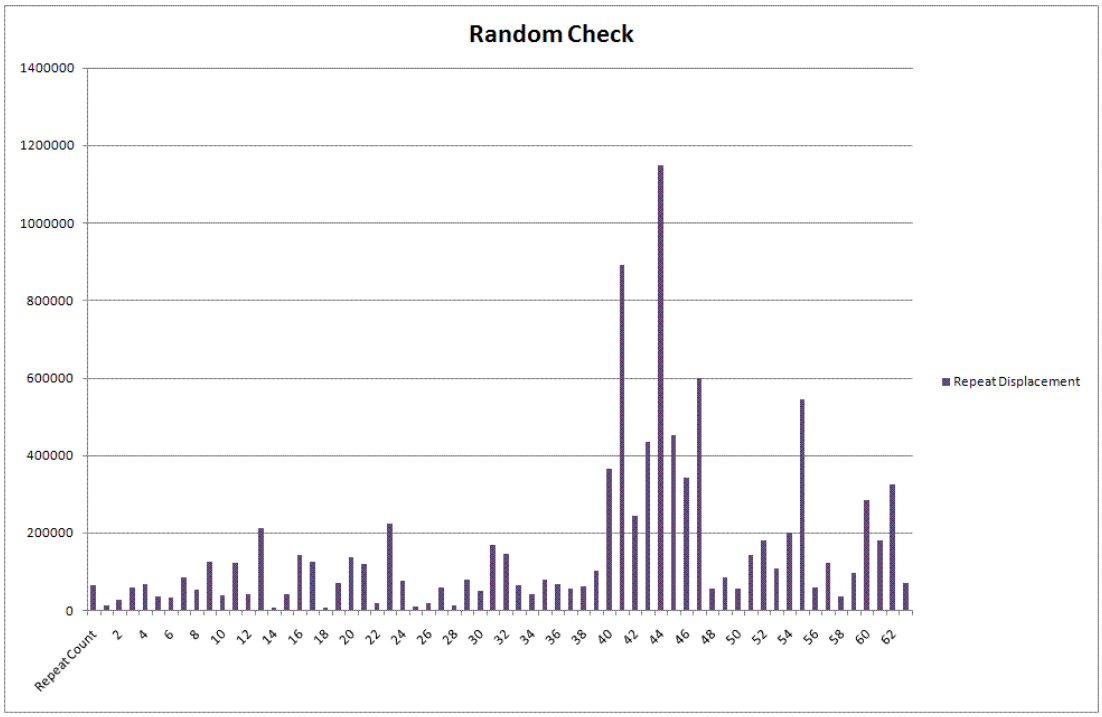


© 2007 Sebastian Sohn



A series of 10 million numbers was taken for this test. The values vary even in a short sub-series of 100 numbers. To generate nearly white noise the 4 LSBs of several numbers can be joined to a big number. It takes 8 passes to get a 32 bit random number.

5.7.2. Random Check

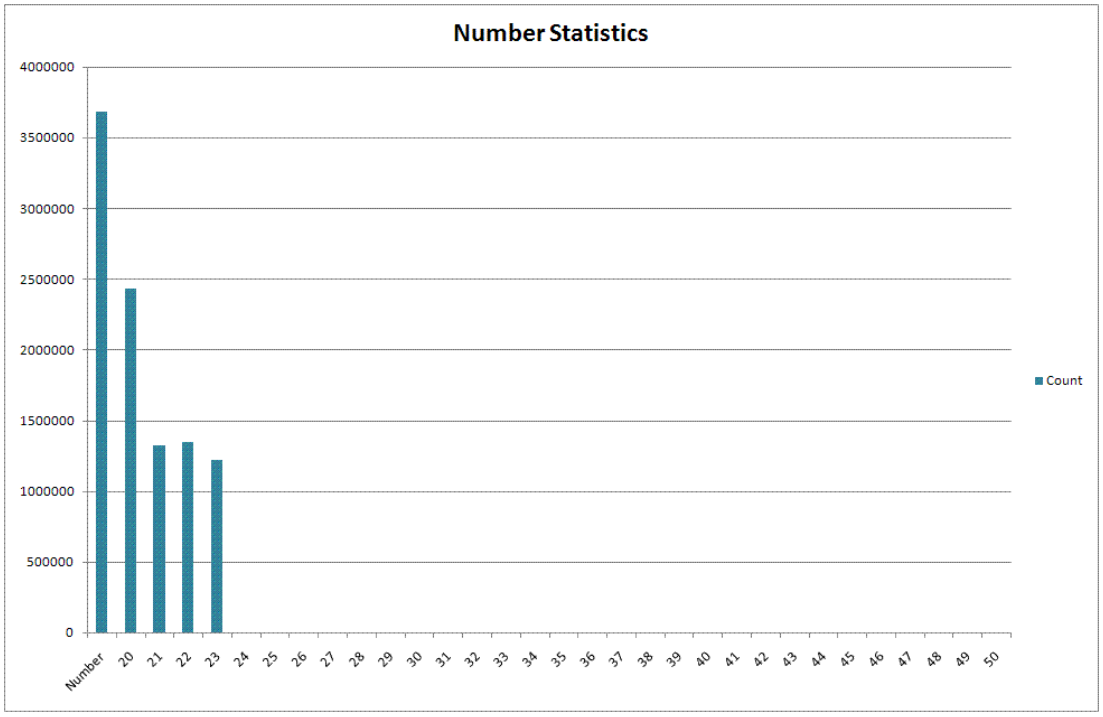


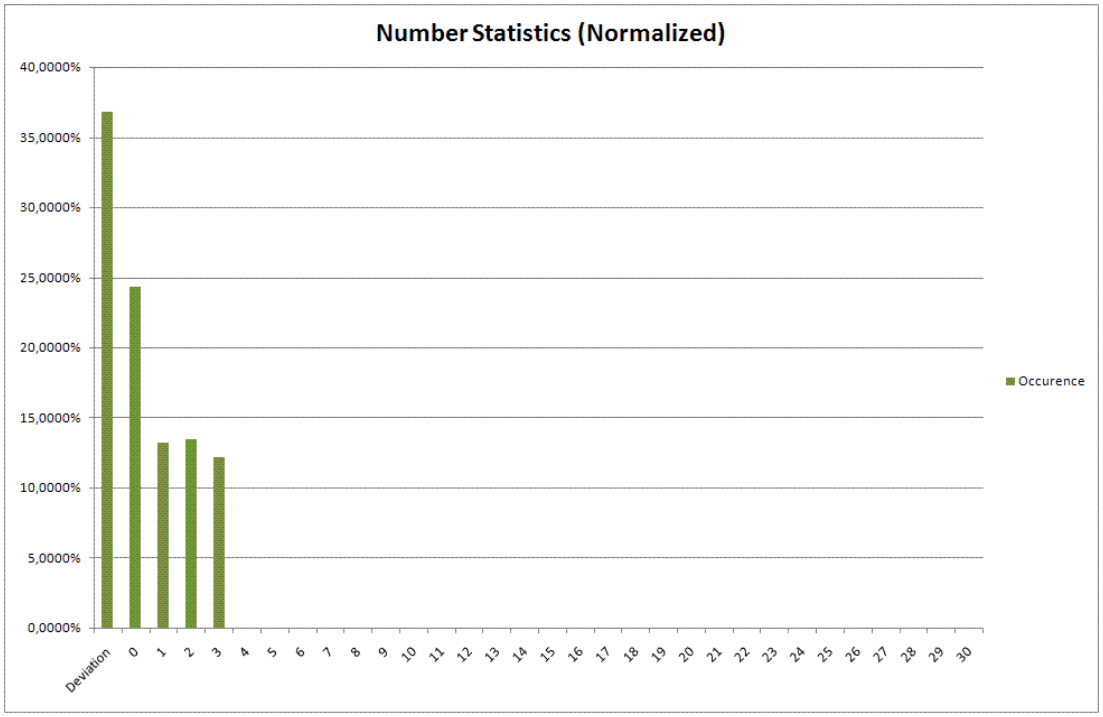
For the randomness check the sub-series “524, 504, 507” was used. It occurred 64 times in the complete series. The deviation of the sub-series occurrence varies non-cyclic. The ADC method passes the random check.

© 2007 Sebastian Sohn

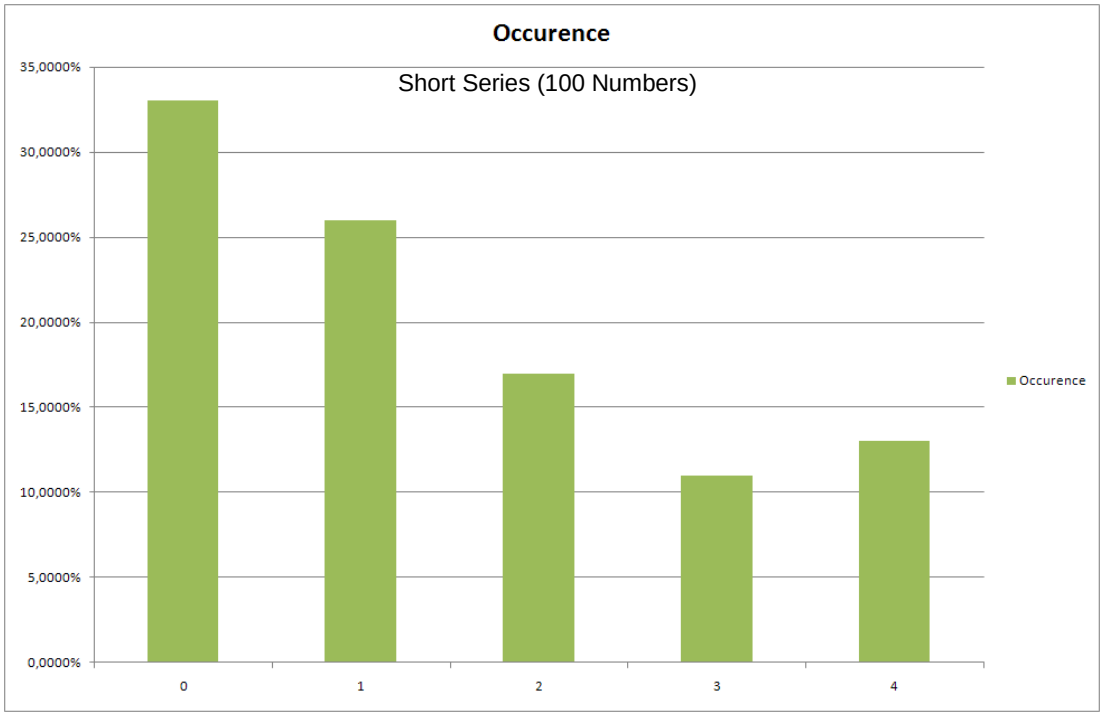
5.8. Timer ISR

5.8.1. Number Statistics



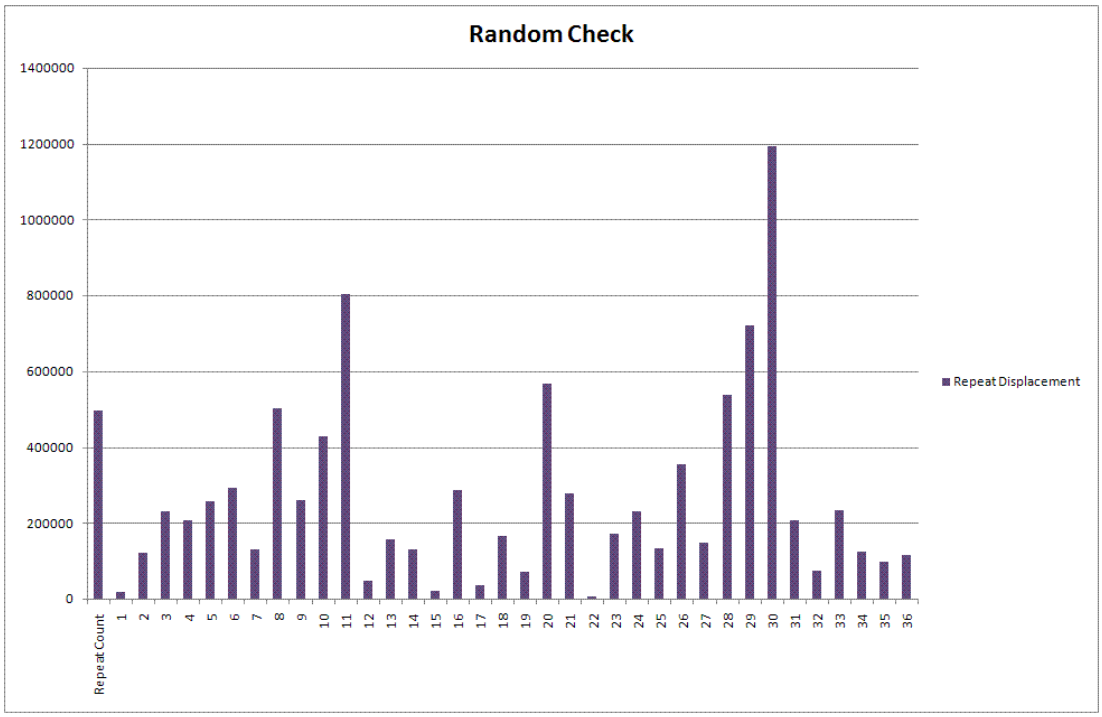


© 2007 Sebastian Sohn



A series of 10 million numbers was taken for this test. The values vary even in a short sub-series of 100 numbers. To generate a large random number the LSB of several numbers can be joined to a big number. It takes 32 passes to get a 32 bit random number.

5.8.2. Random Check



For the randomness check the sub-series “21, 20, 23, 23, 21, 22” was used. It occurred 37 times in the complete series. The deviation of the sub-series occurrence varies non-cyclic. The timer interrupt method passes the random check.

© 2007 Sebastian Sohn

6. Conclusion

9 methods were mentioned and 8 methods were successfully tested to generate true random numbers in an embedded system. Random number generation via the input capture unit was not tested. Each method has its strengths and weaknesses. However, the best method would always be the one which uses available periphery and fulfills the timing requirements.

Method	Resources	Time	Random Bits
Internal EEPROM	1 byte of internal EEPROM EEPROM Ready interrupt 1 Timer	1.8ms	4
Watchdog Reset	Watchdog RESET All other applications are blocked	16ms	2
Watchdog Interrupt	Watchdog Watchdog interrupt 1 Timer	16ms	4
RS232 (or other asynchronous interface)	Asynchronous interface 1 Timer	Min 10ms	16
Low Freq. Crystal	32kHz crystal 2 Timer	7.8ms	1
ADC	1 Z-Diode, 2 Resistors, 1 Elko cap 2 ADC input pins ADC	250µs	4
Timer ISR	1 Timer Timer interrupt	10ms	1
Input Capture Unit	TBD	TBD	TBD

7. Donate

Did you find this article valuable? If you would like to make a donation, send some money to the evangelic church in Nuembrecht. For many years our church has supported projects in Tanzania (east Africa), including Training centre Bangala, Schools, Education of pupils, Bumbuli Hospital, Water support, and so on.

All organization work for these projects is done by volunteers in Germany and Tanzania, thus 100% of your donation is used for the Tanzania projects instead of wasting it on expensive overhead. All projects include close collaboration with the people in Tanzania, which avoids wasting money for useless projects.

If you have questions, please contact one of our pastors (Pastor → Pfarrer). You'll find their e-mail addresses on the homepage of the evangelic church in Nuembrecht: www.ev-kirche-nuembrecht.de. Click on "Kontakte" to find several E-Mail addresses listed. Choose one of the "Pfarrer" addresses.

Of course, you can get a donation receipt for tax purposes.

Ev. Kirche Nuembrecht Bank Details:	
Germany	***International***
Konto: 21 11 32 70 10	IBAN: DE75 3846 2135 2111 3270 10
BLZ: 384 621 35	BIC: GENODED1WIL
Reason for transfer: Tanzania	

© 2007 Sebastian Sohn