Zürcher
Hochschule
Winterthur

"It is impossible to predict the unpredictable."
-Don Cherry

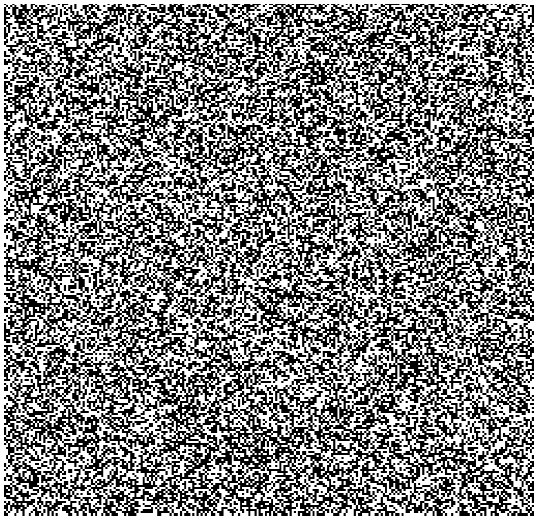


Abbildung 1 True white noise bitmap from random.org

True Random Number Generator

Generating True Randomness for Embedded Systems

PA1 Sna 03/2

Roland Siegert, IT3a
Andreas Stricker, IT3a

Dozent: Andreas Steffen

Table of Contents

1 Abstract

The Output of this Project is a Prototype of a True Random Bit Generator.

Of great interest is the Analysis of Entropy and proper statistical independence of the generated random bit sequences. To analyse the produced random data we used not only the Cryptool test suite but we also especially implemented Ueli Maurer's Universal Statistical Test.

We kept an an eye on two important issues all along the project. These were first true randomness and second the datarate. The combination of our hardware and the correcting software running on the target system produced a sastisfying amount of randomness. Allthoug the datarate is in need of improvement, we dare say that it is sufficient for purposeses such as generating crypto keys for SSL connections.

As it is a prototype's nature, there is scope for perfectioning.

2 Zusammenfassung

Wir entwickelten einen kostengünstigen Zufallsgenerator, der Daten mit guter Qualität liefert.

Die Datenrate mit 1Byte/s beeindruckt im Moment noch nicht, sollte aber für den Verwendungszweck, die SSL/TLS Implementierung auf einem IPC@CHIP System ausreichen.

Zudem ist hier noch ein riesiges Potential vorhanden, so dass durch Optimierung der Schaltung die Datenrate stark erhöht werden kann.

Für das Zielsystem wurde Software entwickelt, die aus einem Treiber besteht, der die Zufallsdaten von der Hardware-Schaltung einliest und über eine definierte Schnittstelle der Anwendung zur Verfügung stellt.

Auf dem Markt wurden keine kostengünstigen Zufallsgeneratoren gefunden, die eine nicht deterministische Quelle verwenden.

Ein weiteres Produkt diese Projektarbeit ist die Implementierung des Universal Statistical Test von Ueli Maurer, der kryptografische Defekte eines Zufallsgedächnis misst.

Neben Maurers Test zogen wir weitere zwei Testsuites für die Analyse der Zufallsgenerator-Daten bei und stellten interessante Vergleiche mit Zufallsdaten aus anderen Quellen an.

Nach der Interpretation der Testergebnisse kamen wir zum Schluss, dass es sich bei unseren generierten Bitfolgen um echt zufällige Folgen handelt.

Winterthur, 16.5.2003

Roland Siegert Andreas Stricker

3 Aufgabenstellung

Arbeit Nummer: PA1 Sna 03/2
Fachgebiet: Kommunikation
Dozent(en): Andreas Steffen, Büro: E509, Tel.: 434
Studiengang: --/IT/--
Anzahl Studenten: 2

3.1 Beschreibung:

In kryptografisch-sicheren Kommunikationprotokollen beruht die Sicherheit auf der Voraussetzung, dass die für die Verschlüsselung und die Authentisierung verwendeten Session-Keys völlig zufällig aus dem gesamten verfügbaren Schlüsselraum gewählt werden. Deshalb ist es notwendig, dass auf dem Zielsystem eine Zufallsquelle vorhanden ist, welche genügend Zufallsdaten liefern kann. In einem Embedded System kann dies problematisch werden, da weder auf Tastaturklicks oder Mausbewegungen, noch auf Zugriffsaktivitäten einer Festplatte zurückgegriffen werden kann und die durch die vorhandene Ethernetschnittstelle erfasste Netzwerkaktivität von aussen manipuliert werden kann. Glücklicherweise werden Embedded Systems meist zur Steuerung von Prozessen eingesetzt, so dass analoge und/oder digitale IO-Schnittstellen vorhanden sind, über die Zufallsinformation gewonnen werden kann.

In dieser Projektarbeit sollen einfache und kostengünstige Schaltungen untersucht werden, die als Zufallsquellen genutzt werden können. Über eine IO-Schnittstelle sollen die Zufallsdaten in den Prozessor übernommen werden. Mit Hilfe von statistischen Tests soll die Qualität der generierten Zufallssequenzen überprüft werden.

Weiterführende Informationen: http://world.std.com/~reinhold/truenoise.html

3.1.1 Voraussetzungen:

- Erfahrung in C++ oder C Programmierung
- Erfahrung im Umgang mit Microcontrollern von Vorteil
- Interesse am Gebiet der Netzwerksicherheit

3.1.2 Partnerfirma:

Arbeit im Rahmen des EDiSoN Projekts "Secure Communication in Distributed Embedded Systems"

4 Aufgabenplanung

4.1 Zeiteinteilung

Soll

Task	12	13	14	15	16	17	18	19	20
Planung									
Einarbeitung Theorie									
Einarbeitung System									
Hardware Evaluation									
Hardware									
Treiber, API									
Testsoftware									
Fehlersuche									
Auswertung, Bewertung									
Dokumentation									

Ist

Task	12	13	14	15	16	17	18	19	20
Planung									
Einarbeitung Theorie									
Einarbeitung Test Maurer									
Einarbeitung System									
Hardware Evaluation									
Hardware									
Treiber, API									
Testsoftware									
Fehlersuche									
Auswertung, Bewertung									
Dokumentation									

4.2 Aufgabenteilung

Planung	X	X
Einarbeitung Theorie	X	X
Einarbeitung Test Mauer	X	X
Hardware Evaluation	X	X
Hardware		X
Treiber, API	X	
Testsoftware Zielsystem	X	
Testsoftware PC		X
Fehlersuche	X	X
Auswertung, Bewertung	X	X
Dokumentation	X	X

5 Einleitung

Zufallszahlen spielen in der Kryptographie eine grosse Rolle: Passwörter/Passphrase und zufällige Primzahlen bestimmen die Sicherheit heutiger Systeme wesentlich.

Dabei ist es gar nicht trivial, gute Zufallszahlen zu generieren. Pseudozufallsgeneratoren (PRNG) scheiden aus, da hier immer ein Angriff auf den Seed möglich ist. Der Seed muss also sicher sein, wobei nur echte Zufallszahlen in Frage kommen: Wir stehen wieder am gleichen Ort.

Wesentlich an echten Zufallszahlengeneratoren (RNG) ist, dass die Zufallsquelle nicht deterministisch ist: Es ist unmöglich aus den bereits ausgelesenen Zahlen auf die nächste Zahl zu schliessen, die ausgelesen wird.

Hat man einen Zufallsgenerator gebaut, so gilt es diesen auf mögliche Schwächen zu prüfen. D.h. man muss ein Verfahren haben um die Qualität der Zufallszahlen zu beurteilen. Dazu werden die Zufallsdaten statistisch untersucht. Die Statistik bringt allerdings auch das Problem mit sich, dass man nie ganz ausschliessen kann, das ein guter Zufallsgenerator schlecht beurteilt wird. Abhilfe gibt es in dem man eine grössere Menge an Daten testet.

6 Hardware

6.1 Geeignete Generatoren

Als erstes musste nach geeigneten Zufallsgeneratoren (RNG) gesucht werden. Die Recherche in den vorgeschlagenen Dokumenten ([AREINHOLD], [RFC1750]) und im Internet ergaben folgende Möglichkeiten:

- Quantenmechanische Effekte
- Radioaktiver Zerfall [HOTBITS]
- Thermisches Rauschen an Widerständen und Dioden
- Radiowellen (Antenne) [R.ORG]
- Netzwerk-Verkehr
- LSB eines AD-Wandlers
- Fertige, kommerziell vertriebene RNG, die auf einer der genannten Techniken aufsetzen

Pseudo-Zufallsgeneratoren (PRNG) genügen nicht den Ansprüchen und werden weiter nicht mehr beachtet.

Als nächstes prüften wir die Machbarkeit und konnten folgende Varianten verwerfen:

- Quantenmechanische Effekte zu messen ist viel zu aufwändig und für uns nicht machbar.
- Radioaktiver Zerfall lässt sich mit einem Geigerzähler oder ev. mit einem Rauchdetektor messen. Beides erwies sich als zu teuer.
- Radiowellen können mit wenig Aufwand empfangen werden. Allerdings kann ein Angreifer hier mit einem Sender den RNG beeinflussen.
- Beim Netzwerk-Verkehr gilt das gleiche wie bei den Radiowellen.
- Auf dem zur Verfügung gestellten Testsystem (DK40) existiert kein AD-Wandler.
- Es finden sich einige RNG auf dem Markt. Jedoch fanden wir keinen für unter 100 Franken. Bei einem Mikrocontroller bedeutet das eine Verdopplung des Preises.

Schliesslich blieb nur noch die Variante mit dem Thermischen Rauschen. Während moderne Pentium-Prozessoren bereits einen entsprechenden RNG enthalten, konnten wir Bauteil finden werden, das eine entsprechende Schaltung enthält und somit ohne grossen Aufwand an das System hätte angeschlossen werden können.

Wir suchten dann im Internet nach RNG-Schaltungen. Wir fanden nur zwei verschiedene Varianten.

6.2 Hardware-Zufalls-Generatoren

6.2.1 Transistor-Rauschen

Die erste Schaltung auf Seite 24, die wir untersuchten überzeugte durch ihre Einfachheit: Drei Transistoren, ein paar Widerstände und Kondensatoren. Nach dem Zusammenbau stellte sich schnell heraus, dass die Verstärkung mittels eines Bipolaren Transistors nicht ausreicht, eine genügend grosse Amplitude zu erzeugen, welche ausreicht um der Hysterese des nachfolgend geschalteten Schmitt-Triggers zu folgen. Der Erstz des Transitor-Verstärkers durch einen Operations-Verstärker ergäbe eine genügend grosse Amplitude.

Dadurch nimmt die Zahl der Bauteile zu. Damit erreicht sie die gleiche Anzahl Bauteile, wie die nachfolgend vorgestellte Schaltung.

6.2.2 Dioden-Rauschen

Die anfangs verwendete Schaltung wurde für weisses Rauschen in der Audiotechnik entwickelt (Illustration 2, Quelle: [NOISE]). Wir ziehen die Funktionstüchtigkeit, weil hier der Operations-Verstärker nicht direkt an der Speisung hängt, sondern noch über ein 120Ω-Widerstand geführt wird. So schlägt sich jede kleine Änderung des Ausgangs als eine Schwankung auf der Versorgung nieder. Diese führt wiederum dazu, dass die gesammte Schaltung anfängt zu Schwingen: Anstatt eines zufälligen Signals erhält man ein Rechteck-Signal mit ca. 50% Pulsweite.

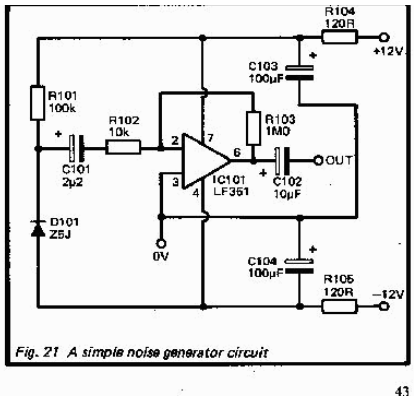


Illustration 2 Audio Noise Generator

Nach erfolglosen Versuchen mit dieser Schaltung mutierte sie durch viele Änderungen zu einer einfachen Verstärker-Schaltung, die das Rauschen über einer Diode verstärkt. In dieser Version ist die Schaltung auf Seite 25 zu finden.

Grundsätzlich könnte noch die Diode durch die beiden Transistoren der ersten Schaltung mit dem Prinzip des Transitor-Rauschens ersetzt werden.

Das Rauschen über der Diode wird mittels eines Kondensators (C1) vom DC-Anteil entkoppelt auf einen Operations-Verstärker (IC1A) geführt. Dieser verstärkt um den Faktor 1000. Danach wird über einen zweiten Verstärker (IC1B) nochmals um den Faktor 22 verstärkt und das Signal wiederum vom DC-Anteil entkoppelt (C2) auf einen Schmitt-Trigger (V2) geführt. Zum Schluss stehen rechteckige Pulse mit TTL-Pegel zur Verfügung, das problemlos über einen 1 Bit breiten digitalen Eingang eingelesen werden kann.

Da nur eine Versorgungsspannung von 24VDC zur Verfügung steht, wurde mittels einer 12V-Zehner-Diode (D2) ein virtueller Ground (GND1) erzeugt, so dass der Operations-Verstärker nun mit ±12V betrieben werden kann. Das funktioniert, da das Rauschen sowieso vom DC-Anteil entkoppelt werden muss.

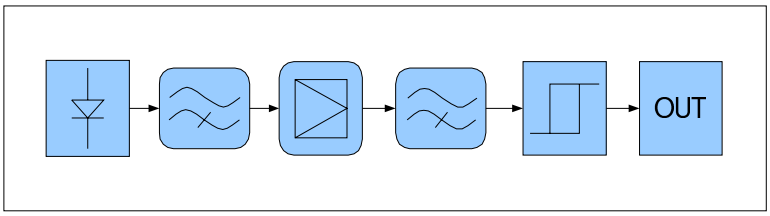


Illustration 3Filter-Schema der Schaltung mit Dioden-Rauschen

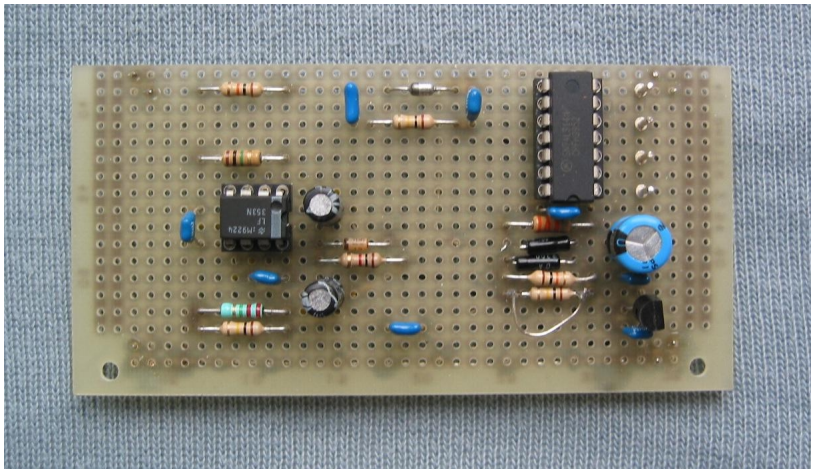
Beim Test zeigten sich dann massive Probleme: Zuerst war die Verstärkung (1000-Fach) zu klein, um der Hysterese zu folgen. Die Verwendung des zweiten Operations-Verstärker konnte dem zwar abhelfen, führte aber dazu, dass durch die erhöhte Last und die dadurch entstehenden Spannungsschwankungen auf der Versorgung wieder die gesammte Schaltung zu schwingen begann. Nach einer längeren Optimierungsphase mittels mehrfachen und dicken Versorgungsleitungen, Stütz- und Block-Kondensatoren, sowie auf möglichst kleine Last optimierte Verstärkung, konnte ein stabiler Zustand erreicht werden. Es resultierte aber ein Signal mit viel zu selten auftretenden Pulsen, so dass die Datenrate für eine praktische Anwendung nicht ausreichte (0.8 Bit/s).

Aufgrund der vielen Änderungen, welche die Schaltung schon hinter sich hatte, entstanden viele Störungen.

Dank einem sauberen Neuaufbau der Testplatine, gelang es die Datenrate um ca. den Faktor 10 erhöhen, was etwa 1 Byte/s entspricht.

Trotz des neuen Aufbaus begann auch diese Schaltung zu schwingen, sobald gewisse Leistungsspitzen auftreten. Das Problem konnten wir mit einer eigenständigen Verstärker-Platine lösen. Die Verstärker-Platine-Schaltung ist auf Seite 29 abgebildet.

Mit dieser Hardware wurden die hier untersuchten Zufallsdaten generiert.



6.2.3 Empfehlung für einen produktiven Zufallsgenerator

Für einen praktisch einsetzbaren Zufallsgenerator ist die bisherige Lösung ungeeigntet, da sie sehr nahe am Limit arbeitet. Eine andere Umgebung mit schlechterer Versorung oder mehr Störungen könnte dazu führen, dass die Schaltung wieder schwingt.

Das eigentliche Problem liegt darin, dass durch die Stromaufnahme des Verstärkers und des Schmitt-Triggers immer eine kleine, aber durchaus messbare Schwankung auf der Versorgung entsteht. Die enorme Verstärkung (>10^4) der Spannung über der Diode, verstärkt eben nicht nur das Rauschen, sondern auch die Störungen, die teils selbst erzeugt werden und teils von der Versorgung herrühren (Netzbrumm etc.). Der Netzbrumm kann noch mit einem Hochpass-Filter beherrscht werden, während den selbst produzierten Störungen so nicht beizukommen ist.

Ein Entwurf der hier empfohlenen Schaltung ist auf Seite 27 abgebildet.

Eine mögliche Lösung habe ich aus der Kommunikations-Technik abgeschaut: Bei der differenziellen Übertragung von Signalen wirken von aussen eingestreute Störungen auf beide Signalleitungen. Die Lösung für den Zufallsgenerator sieht dann zwei Pfade mit Dioden vor (D1, D2). Jeder Pfad wird vom DC-Anteil entkoppelt (C1, C2). Nun kann die Differenz der beiden Rausch-Signale mit Hilfe eines Instrumenten-(Differenzial-) Verstärkers verstärkt werden (IC1A,B,C). Symmetrische Störungen werden damit nicht mitverstärkt.

Ein weiters Problem ist der Mittelwert des Ausgangssignals der $\frac{1}{2} \cdot U_a$ betragen müsste, was aber fast unmöglich ist. Damit trotzdem ein Signal entsteht, in dem die „1“ und die „0“ jeweils eine Wahrscheinlichkeit von 0.5 besitzen, kann beim Einlesen die *von Neumann* Methode eingesetzt werden (siehe später). Jedoch bringt diese Methode einen Nachteil mit sich: Je mehr Eins- oder Null-Pulse überwiegen, umso geringer wird die Bitrate.

Ziel ist eine gleiche Häufigkeit von Nullen und Einsen. Der in den bisher realisierten Schaltungen eingesetzte Schmitt-Trigger (74LS14) ist da sehr ungünstig: Erstens ist ein weiteres Bauteil notwendig, obwohl noch jeweils ein freier Operations-Verstärker pro Gehäuse vorliegt, zweitens ist die Signal-Mitte der Umschalt-Level mit ca. 0.8V und 1.6V schwierig zu „treffen“. Bei der Verwendung des vierten Operations-Verstärkers (IC1D) lässt sich die Mitte gut auf 0V (virtueller Ground, GND1) legen und die Umschalt-Pegel sind einfach symmetrisch auf einen beliebigen Wert zu legen. Am Ausgang liegen dann auch gleich die ca. ±12V [1]. Das entspricht in Bezug auf den normalen Ground (GND) den 24V-Eingängen unseres Microcontrollers.

Damit kann eine RNG-Schaltung mit einem IC, 3 Dioden, 12 Widerstände, 8 kleinen Kondensatoren und 3 grossen realisiert werden.

1 Nicht ganz ±12V, da der Operationsverstärker nicht komplett ausschlagen kann.

6.2.4 Wasserfall-Bausatz

Dieser Bausatz, der eigentlich dazu gedacht ist, akustisch Wasserfall-Rauschen zu erzeugen, wird von Arnold Reinhold [AREINHOLD] als Zufallsgenerator empfohlen, da dieser echtes, weisses Rauschen erzeugt.

Gerne würde ich hier über dieses Model schreiben. Leider ist die Bestellung irgendwo zwischen Europa und Amerika im Meer versunken...

7 Software

7.1 Zielsystem

7.1.1 Zielsystem Hardware

Bei dem Zielsystem, an welches die Zufallszahlen erzeugende Hardware anzuschliessen war, handelt es sich um den Single Chip Embedded Webserver IPC@Chip von der deutschen Beck IPC GmbH.

Für unser Projekt ist das Essentielle dieses Geräts, dass unsere Hardware direkt an die 24V I/O-Pins angeschlossen werden kann und die Zufallssequenzen zwecks Analyse über eine TCP/IP-Verbindung weitergereicht werden können. Der Chip stand uns in einem für Entwicklungszwecke konzipierten Kit (DK40) zur Verfügung.

Weil dieser Embedded Controller Produkte LAN- und WEB-fähig macht, würde sich unser RNG, sobald er zur Produktreife getrieben wird, speziell für den Einsatz für sichere (verschlüsselte) Kommunikation in einem Netzwerk eignen.

Der Hersteller [BECK] beschreibt sein Produkt genauer:

Der IPC@CHIP® ist eine sofort einsetzbare Kombination von Hardware und Software.
Die **Hardware** verfügt über einen 16bit 186-Prozessor, RAM, Flash, Ethernet, Watchdog, und einen Power-fail Detektor.
Die vorinstallierte **Software** basiert auf einem Echtzeitbetriebssystem (RTOS) mit File System, TCP/IP Stack, Webserver, FTP-Server, Telnet-Server und Hardware Interface Layer.

7.1.2 Zielsystem Software

Bedienung

Das Betriebssystem stellt über die serielle Schnittstelle eine Kommandoshell zur Verfügung. Die üblichsten Befehle entsprechen denen von MS-DOS. So wird unser Programm mit dem Programmnamen gestartet.

Die IP-Adresse der Clientapplikation ist der einzige obligatorische Parameter. Das Hashen der Zufallszahlen muss mit einer Option verlangt werden. Hier gehört dann auch die Angabe dazu, wieviel Prozent der Eingangs- bezüglich des Ausgangspuffers umfassen soll (s.unten).

Falscheingaben werden mit der Ausgabe einer kleinen Hilfe quittiert.

```
A:\RANDOM\>random

error input! - usage:

random IP [-cp CLIENT_PORT] [-hp HOST_PORT] [-md5 PERCENTAGE] [-pin HEX]

IP               Set the client's ip address to IP
-cp  CLIENT_PORT Set the client application's port to
                 which random data is to be sent to.
                 (default CLIENT_PORT is 7331)
-hp  HOST_PORT   Set listening port to CLIENT_PORT
-md5 PERCENTAGE  1. Enable hashing
                 2. Set the input buffer size in relation
                    to its Output buffer size ( = 128 Bit)
-pin HEX         The pin to which the Random Number Generator
                 is connected to. Give the pin in hex without
                 '0x'-prefix or 'h'-postfix.

error input will show this help message
```

Aufbau

Das Programm besteht aus einem Treiber- und einem Applikationsteil. Der Treiber umfasst sämtliche Funktionen, die die Generierung und Verarbeitung der Zufallsfolgen tangieren: Einlesen, von Neumann-Korrektur, Hashen.

Der Applikationsteil übernimmt die Einrichtung (Initialisierung) des genannten Threads. Danach wartet sie, bis der Pool voll ist und verschickt die Daten.

Funktion

Die main-Methode erstellt eine TCP/IP Verbindung mit der spezifizierten Adresse. Danach richtet sie einen Thread ein, der

- vom RNG Bits liest,
- diese mit der von Neumann Methode verarbeitet,
- falls spezifiziert, mit dem MD5-Algorithmus mischt
- und in den Pool schreibt.

Für unsere Zwecke genügte es, die Daten an ein PC-System weiterzuleiten, wo sie für die weitere Analyse gespeichert wurden.

Für einen produktreifen RNG müsste aus dem Pool eine bestimmbare Anzahl Bytes via System Call an den Client geliefert werden. Diesen System Call haben wir nicht implementiert.

Datenfluss

Die Illustration 5 gibt einen Überblick über den Datenfluss.

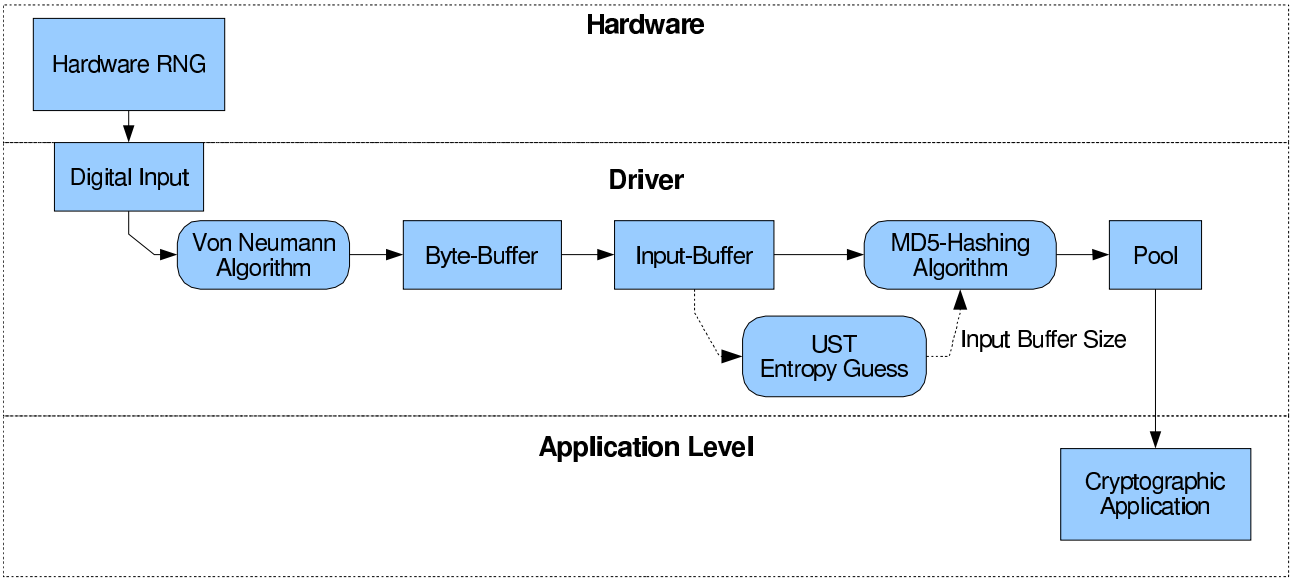


Illustration 5 Datenfluss

Es folgt eine kurze Beschreibung der Softwarekomponenten:

Digital Input

Die Pins des DK40 müssen in unserem Fall als Inputpins initialisiert werden, bevor von ihnen gelesen werden kann.

Von Neumann Algorithmus

Die rohe aufgezeichnte Bitsequenz unseres RNGs weist aus erwähnten elektronischen Gründen ein starkes Ungleichgewicht von Nullen und Einsen auf. In einem Verfahren von Von Neumann wird diese Verschiebung beinahe eliminiert:

- Es wird eine neue zunächst leere Sequenz erzeugt.
- Die RNG-Bitsequenz wird auf nicht überlappende Paare „00“, „01“, „10“ und „11“ untersucht.
- Für ein Paar „01“ wird der *neuen* Sequenz eine Null hinzugefügt.
- Für ein Paar „10“ wird der *neuen* Sequenz eine Eins hinzugefügt.
- Die Paare „00“ und „11“ werden ignoriert.

Die so gewonnene Folge repräsentiert die Zufallsdaten, von denen in unserem Projekt jeweils die Rede ist.

Byte Buffer

Unseren von Neumann Algorithmus lassen wir die einzeln erzeugten Bits in ein Byte schieben. Sobald der beschriebene Byte Buffer voll ist, wird dieses Byte in den Input Buffer übertragen.

Input Buffer

Der Input Buffer zeichnet sich dadurch aus, dass die Grenze, bis zu der er gefüllt werden soll, beim Progammstart gewählt werden kann. Das hat Konsequenzen für den Fall, dass der Input Buffer-Inhalt der MD5-Hash-Funktion zugeführt wird.

Seine maximale Grösse legten wir der Einfachheit halber auf 512 Byte fest. Eine dynamische Speicherallozierung bedarf bei unserem Zielsystem gewissen Vorkehrungen.

MD5-Hash-Funktion

Sollen die Zufallsdaten des Input Buffers zusätzlich gehashed werden, kann das durch die Option „-md5“ beim Progammstart angegeben werden.

Wir nützen die Eigenschaft des MD5 Hash Algorithmus aus, dass er Eingangsdaten gut mischt. Ziel des Mischens ist natürlich, unsere Zufallsdaten im Input Buffer noch sicherer zu machen. „Sicherer“ bedutet: Die Eingangsdaten sollen schwieriger erratbar werden.

Unsere Hashfunktion bildet beliebig viele Eingangsbit möglichst gleichmässig auf den Wertebereich, hier 128 Bit ab. Es handelt sich also um eine surjektive, aber nicht injektive Funktion mit der zusätzlichen Eigenschaft, den Wertebereich gleichmässig auszunutzen.

Die MD5 Hash-Funktion liefert so viel Entropie, wie hineingesteckt wird, maximal aber 128 Bit. Handelt es sich bei der Eingangssequenz um “schlechten Zufall”, enthält sie wenig Entropie pro Bit. Durch mehr Eingangsdaten wird absolut mehr Entropie in die Funktion gegeben, wodurch die Ausgangssequenz mehr Entropie pro Bit enthält.

Aus diesem Grund kann die Grösse des Input Buffers prozentual zur Grösse der Outputsequenz der Hashfunktion gewählt werden. Damit könnten wir, die Eingangsdaten zu Lasten der Datenrate qualitativ verbessern.

Der Output Buffer unserer MD5-Hash-Funktion umfasst 16 Byte. Das entspricht 128 Bit, also gerade der Datenmenge, wie sie für die häufig verwendete 128-Bit-Verschlüsselung benötigt wird.

Pool Buffer

Die Grösse des Pool Buffers legten wir auf 1024 Byte fest, seine Datenstruktur ist die des allgemein bekannten Ringpuffers. In unserem Programm ist ein separater Thread damit beschäftigt, den Pool ständig mit neuen Zufallsdaten zu beschreiben.

Der Zweck des Pools liegt auf der Hand: Ein Client soll auf Anfrage unverzüglich eine gewisse Menge Zufallsdaten erhalten, ohne auf die langsame Erzeugung warten zu müssen.

Weil der Pool von einem Thread gelesen und von einem anderen geschrieben wird, handelt es sich um das bekannte "Reader-Writer"-Problem: Der Pool muss durch eine Semaphore geschützt werden.

UST Entropy Guess

Der Einbau dieser Funktionalität scheiterte mangels Gleitkomma-Emulation in Threads ausserhalb der main()-Methode. Die Emulatin muss mit dem Befehl -fpreset(void) rinitialisiert werden. Leider versagte der im DK40-Handbuch beschriebene Weg. Die Alternative ist, die Entropy Guess Funktion für ganzzahlige Typen umzuschreiben.

Die Idee sei trotzdem erklärt: Wir wollten unser System automatisch auf Veränderung der Entropie reagieren lassen. Dabei schätzt der UST die Entropie der gerade eingelesenen Daten. Diese laufende Überprüfung erlaubt uns, die Eingangssequenz des MD5 Hash Algorithmus bei Verschlechterung der Entropie länger, bei Verbesserung gleich lang wie die Ausgangssequenz werden zu lassen.

Den Code dieser Funktion testeten wir mit unseren Zufallsdaten unter Linux.

Analyse-Software

Das Zielsystem hat zu wenig Speicher um eine genügend grosse Menge an Zufallsdaten zu speichern, die einer aussagekräftigen Beurteilung genügt. Deshalb werden die Zufallsdaten zur Auswertung an ein PC-System geschickt.

7.1.3 Readrand

Obwohl es grundsätzlich Tools wie Netcat und hexdump gibt, entschloss ich mich, ein kleines Tool zu schreiben, das auf einem wählbaren Port auf eine TCP-Verbindung wartet und alle dort eintreffenden Daten entweder binär oder hexadezimal auf stdout ausgibt.

Das Tool folgt den üblichen Unix-Konventionen und listet mittels der Option –`help` alle möglichen Optionen auf.

```
# ./readrand --help
Read Random Numbers over TCP/IP from Beck DK40@CHIP

syntax ./readrand [-b] [-p PORT] [-c COLUMN] [-h]

        -p PORT     Set listening port to PORT (default 7331)
        -c COLUMN   Set columns a line in hex mode (default 16)
        -b          Set to binary mode instead of hex output
        -h          Show this help message
```

8 Stattest

Testsuites für die Untersuchung der Qualität von Zufallsfolgen gibt es drei weitverbreitete ([CRYPTOOL], [DIEHARD], [ENT]). Keine von diesen kann aber den Universal Stattistical Test (UST) von Ueli Maurer [MAURER] durchführen.

Deshalb schrieben wir eine eigene Test-Suite, welche diesen Test durchführen kann. Neben dem UST sind noch grundlegende Tests zur Beurteilung der Zufallsdaten vorhanden: Bias, Entropie und Histogramm.

Da die Hauptspeicher heutiger Rechner genügend gross sind, liest das Tool gleich die gesamten Daten in den Hauptspeicher, wodurch die Tests selbst bei 10MByte grossen Dateien schnell abgearbeitet sind.

Auch dieses Tool folgt den üblichen Unix-Konventionen und gibt durch die Angabe der Kommando-Zeilen-Argument –`help` bereitwillig Auskunft über die möglichen Optionen:

```
./stattest --help
Statistical test for randomness, Andreas Stricker, Roland Siegert

syntax: ./stattest [OPTION] [-h] FILE
PARAMETERS:
        FILE                 File to test for randomness
OPTIONS:
      --entropy    -e       Only return entropy
      --bias       -b       Only return bias
      --histogram  -i       Only return histogram
      --universal  -u       Only return universal statistical test
      --rejectrate RATE     Set rejection rate to RATE used by
                   -r RATE   universal statistical test (Default 0.001)
      --blocksize  -L       Set blocksize L for universal statistical test
      --fixedq     -Q       Set initialisation length to fixed size: 10*2^L
      --maxsize SIZE        Only read first SIZE Bytes of file. It possible
                   -m SIZE   to use 'k' for kilobytes and 'M' for megabytes
      --help       -h       Show this help
```

Die Implementierung des Tests ist in der Lage, nicht nur eine fixe Anzahl Initialisierungsschritte Q abzuwarten, sondern erkennt selbständig, ob schon vorher alle Werte bereits einmal aufgetreten sind.

9 Analyse der Zufallsdaten

9.1 Tests

9.1.1 Universal Statistical Test (UST)

Bisherige Tests gehen immer von einer speziellen Schwäche des RNG aus und überprüfen dann diese Annahme. Der UST [MAURER] ist in der Lage alle Schwächen zu erkennen, die von Quellen mit Gedächnis (BMS) mit $M \leq L$ Gedächtnis herrühren[2]. Damit eignet er sich nicht, einen PRNG zu testen, da dann L mit der Grösse des PRNG Register übereinstimmen müsste, was aber aufgrund dessen Grösse mit heutiger Technik nicht möglich ist.

Die Idee hinter diesem Test wurde von dem Komprimierungs-Experten Ziv eingeführt: Mittels eines universellen Kompressions-Algorithmus wird eine Sequenz komprimiert. Eine Zufalls-Sequenz ist nicht komprimierbar.

Maurer zeigt, dass der Testparameter f_{TU} eng mit der Entropie zusammenhängt. Bis auf eine Konstante reicht f_{TU} bei einer echt zufälligen Sequenz an die Entropie heran. Die Erwartungswerte einer echten Zufallssequenz können berechnet werden und wurden von Maurer tabelliert. Ebenso die Varianz. Da die Abweichung des Testergebnisses der Standard-Abweichung folgt, lässt sich prüfen ob das Ergebnis innerhalb der zu erwartenden Grenzen liegt.

9.1.2 ENT

Die Testsuite ENT [ENT] kann einige von Knuth [TAOCP] empfohlene Tests durchführen. Unter anderem: Entropieberechnung, χ^2-Test, Monte Carlo Analyse für π und serielle Korrelation.

9.1.3 Diehard

Die Diehard Testsuite [DIEHARD] wird selbst bei kommerziellen Zufallsgeneratoren als Referenz angegeben. Der Autor pflegt die Suite schon seit einem viertel Jahrhundert.

Leider ist fast keine Dokumentation zu den einzelnen Tests gegeben, und die Ergebnisse müssen selbst interpretiert werden. Die dafür notwendige Theorie konnten wir nicht erarbeiten.

Die bis zum Ende der Projektarbeit erzeugte Datenmenge reicht nicht für alle Tests dieser Suite.

2 L ist ein Parameter des UST, M ist die Anzahl Bit des Gedächnis der BMS

9.2 Testresultate

9.2.1 Datenerfassung

Neben den Daten unserer Zufallsschaltung testeten wir auch andere Daten, so dass es möglich ist die Werte zu vergleichen. Es konnten nicht von allen Daten-Quellen mehr als 1MByte erfasst werden. Darum wurden die Tests an jeweils dem ersten 1MBytes, den ersten 3MBytes und und den ersten 5MBytes der Probedaten durchgeführt.

Daten	*Beschreibung*
RNG Biased	Zufallsdaten ohne Von Neumann Korrektur, Verhältnis von „1“ zu „0“ grösser als 99:1, nur 1011712 Bytes
earlier-RNG-Rectangle	Stark schwingende Schaltung, fast Rechteck, da Rechteck-Schwingung mit kleiner Sampling-Frequenz aufgenommen, nur 1035264 Bytes
RNG No MD5	Nur Von-Neumann-korrigerte Daten, nicht gehashed
RNG With MD5	Daten mit MD5 gehashed im Verhältnis 1:1
`/dev/random`	Während Internet-Surfen aufgezeichnete Zufallsdaten von SuSE Linux 2.4.19-4GB Random-Device
`/dev/urandom`	Pseudozufallsdaten von SuSE Linux 2.4.19-4GB Pseudo-Random-Device
random.org 10meg001	Beispiel Datei von www.random.org, Stichproben von den anderen 12 Dateien zeigten sehr ähnliche Ergebnisse. Random.org generiert die Zufallsdaten mit Hilfe eines Radio-Empfängers, eingestellt auf einen ungenutzen Kanal.
linux-2.4.20.tar.bz	Komprimiertes Original-Kernel-Archiv. (bzip2-Argument: -9)
linux-2.4.20.tar	Original-Kernel-Archiv, nicht komprimiert

Die Vorliegenden Daten wurden jeweils mit dem selbst entwickelten `stattest`, mit `ent` und mit dem Cryptool analysiert.

Test	*Beschreibung*
Bias	DC-Anteil des Signals. Ein echt zufälliges Signal sollte 0.5 erreichen
Entropie	Entropie pro Byte (Echter Zufall sollte 8 erreichen)
UST	f_{TU} Universal Statistical Test, Erwartungswert für echt zufällige Folgen: 7.1837 Parameter: L = 8, Q = 2560, K=1046016
UST Accepted	Liegt der Wert innerhalb der Standardabweichung? (p = 0.01)
χ^2	Zu dieser Wahrscheinlichkeit in Prozent überschreitet der Wert einer echt zufälligen Folge diesen Wert.
Montecarlo π	Montecarlo-Analyse für π. Bei echter Zufallsfolge konvergiert der Wert gegen π.
Montecarlo Error	Fehler der Montecarlo-Analyse in Prozent.
serial correlation	Serielle Korrelation. Je näher bei Null um so besser.

Test	*Beschreibung*
Frequency Test	Gibt Aufschluss über die Gleichverteilung von Nullen und Einsen.
Poker Test	Testet ob alle Teilfolgen der Länge m gleich häufig auftreten.
Runs-Test	Testet auf monoton steigende oder fallende Teilsequenzen.
Longrun	Es darf keine Folge von Nullen oder Einsen grösser als 34 sein.
Serial-Test	Prüft ob aufeinanderfolgende Bits voneinander abhängig sind.
FIPS PUB-140 Testbatterie	Führt die oben genannten Tests an den ersten 2500Bits der Daten aus. Die Testbatterie ist bestanden, wenn alle Tests bestanden wurden.

	Bias	Entropie	UST	UST Accepted	χ^2	Montecarlo π	Montecarlo Error	serial correlation
earlier-RNG-Rectangle	0.58745	5.278157	4.5143	✖	0.01	2.9313	6.69	-0.085937
RNG No MD5	0.50036	7.999840	7.1845	✔	75	3.1342	0.24	0.000219
RNG With MD5	0.50017	7.999829	7.1824	✔	50	3.1370	0.15	-0.001409
/dev/random	0.50000	7.999826	7.1821	✔	50	3.1397	0.06	0.001711
/dev/urandom	0.50008	7.999833	7.1826	✔	50	3.1443	0.09	-0.001795
random.org 10meg001	0.50011	7.999805	7.1829	✔	25	3.1401	0.05	0.000417
linux-2.4.20.tar.bz	0.48455	7.986680	7.0157	✖	0.01	3.2082	2.12	0.056330
linux-2.4.20.tar	0.42450	5.397908	4.1437	✖	0.01	3.9999	27.32	0.471712

Tabelle 1 Resultate für 1MByte Daten

	Bias	Entropie	UST	UST Accepted	X^2	Montecarlo Value	Montecarlo Error	serial Correlation
RNG No MD5	0.50075	7.999921	7.1845	✔	0.05	3.1398	0.06	0.000066
/dev/urandom	0.50000	7.999944	7.1828	✔	50	3.1414	0.01	-0.000487
random.org 10meg001	0.49987	7.999933	7.1834	✔	5	3.1424	0.03	0.000267
linux-2.4.20.tar.bz2	0.48347	7.988405	7.0601	✖	0.01	3.2138	2.30	0.048925
linux-2.4.20.tar	0.42381	5.356422	4.0540	✖	0.01	3.9999	27.32	0.516348

Tabelle 2 Resultate für 3MByte Daten

	Bias	Entropie	UST	UST Accepted	X^2	Montecarlo Value	Montecarlo Error	serial Correlation
RNG No MD5	0.50073	7.999951	7.1842	✔	0.01	3.1387	0.09	-0.000194
/dev/urandom	0.50002	7.999965	7.1832	✔	50	3.1413	0.01	0.000068
random.org 10meg001	0.49990	7.999963	7.1835	✔	25	3.1429	0.04	-0.000082
linux-2.4.20.tar.bz2	0.48441	7.989345	7.0514	✖	0.01	3.2119	2.24	0.051410
linux-2.4.20.tar	0.40439	5.442433	3.8042	✖	0.01	3.9999	27.32	0.471712

Tabelle 3 Resultate für 5MByte Daten

	Frequency Test α: 0.05	Poker-Test α: 0.05, T: 3	Runs-Test α: 0.05 L: 3	Longrun Länge: 34	Serial-Test α: 0.05	FIPS PUB-140-1 Testbatterie
earlier-RNG-Rectangle	✖	✔	✖	✖	✖	✖
RNG No MD5	✖	✔	✖	✔	✖	✔
RNG With MD5	✔	✔	✔	✔	✔	✔
/dev/random	✔	✔	✔	✔	✔	✔
/dev/urandom	✔	✔	✔	✔	✔	✔
random.org 10meg001	✔	✔	✖	✔	✔	✔
linux-2.4.20.tar.bz	✖	✔	✖	✖	✖	✖
linux-2.4.20.tar	✖	✔	✖	✖	✖	✖

Tabelle 4 Resultate des Cryptools für 1MByte Daten

9.3 Interpretation der Testergebnisse

Der Vergleich der Testergebnisse der Tabelle 1 von zufälligen Sequenzen zeigt, dass jeder der Tests in der Lage ist, nicht zufällige Daten von zufälligen zu unterscheiden. Hingegen ist keiner in der Lage die Pseudo-Zufallsfolge von `/dev/urandom` von einer echten Zufallsfolge zu unterscheiden.

Hingegen fällt auf, dass diese Pseudozufallsfolge in allen Tests sehr gute Resultate erhält.

Die Ergebnisse von `/dev/random` und der mit MD5 [RFC1321] gehashten Zufallsfolge unseres RNG liegen nahe beisammen und haben – abgesehen von der Pseudozufallsfolge – bei allen Tests am besten abgeschnitten.

Das nächste Paar, das nahe beisammen liegt, ist unsere ungehashte Zufallsfolge und die Daten von Random.org [R.ORG].

Am Schluss erscheinen die nicht zufälligen Folgen.

In Tabelle 2 und Tabelle 3 ist zu sehen, dass die Ergebnisse mit zunehmender Zahl an Daten besser werden. Es gibt keine Verschiebung zwischen den Verhältnissen der Ergebnisse.

Tabelle 4 zeigt die Ergebnisse des Tests mit Cryptool. Hier kommt unsere Zufallsfolge ohne MD5 schlecht weg: Drei von sechs Tests hat sie nicht bestanden, wobei der Frequency Test mit $\alpha = 0.01$ erfolgreich bestanden wird. Die Frage ist nun, ob unsere Zufallsdaten schlecht sind.

Wir sind der Meinung, dass der Poker-Test nicht ausssagekräftig ist, da selbst ungenügende Daten, wie das Kernel-Archiv den Test bestehen. Der Runs-Test wird selbst von den Daten von random.org nicht bestanden. Der Serial Test zeigt Schwächen in Bitfolgen, die durch Abhängigkeiten von jeweils zwei benachbarten Bits entstehen können. Der Frequency Test berücksichtigt die gleichmässige Verteilung von Nullen und Einsen.

Der UST entdeckt diese Schwachstellen in der Zufallsfolge (RNG No MD5) nicht. Damit ist davon auszugehen, dass entweder eine BMS mit mehr Speicher als der UST (hier 8 Bit) vorliegt, oder dass der Zufallsblock in den 5% Unsicherheitsbereich fällt. Testweise wurde der UST mit L = 16 durchgeführt (Tabelle 5). Dabei hat die Folge den Test problemlos bestanden. Dennoch bleibt eine gewisse Unsicherheit.

Damit wagen wir die Behauptung: *Die erzeugten Zufallszahlen genügen den Anforderungen. Es sind keine gravierende Mängel zu erkennen.*

	f_{TU}	Accepted
RNG No MD5	15.1664	✔
/dev/urandom	15.1678	✔
random.org 10meg001	15.1658	✔
linux-2.4.20.tar.bz	15.0835	✖
linux-2.4.20.tar	6.6128	✖

Tabelle 5 UST mit L=16, Q=655350, K=1495040, p=0.01

10 Fazit

10.1 Hardware

In [MAURER], S. 2 steht zu lesen, es sei eine anspruchsvolle Ingenieuraufgabe, eine Schaltung zu entwickeln, welche aus physikalischen Zufallsprozessen (in unserem Fall das Rauschen einer Diode) eine Zufallssequenz erzeugt, die

- kein Übergewicht entweder an Nullen oder Einsen (Bias) erzeugt und bei der
- zwischen den Bits keine Abhängigkeiten bestehen.

Diese Aussage bewahrheitete sich. In dieser Projektarbeit gelang die Erstellung einer elektronischen Schaltung, welche zu zufälligen Zeiten eine Eins, meistens aber eine Null liefert.

Dass die Einsen tatsächlich zufällig auftreten, konnten wir durch statistische Analyse der Rohdaten, d.h. nicht durch Software korrigierte Sequenzen, die am Ausgang unserer Schaltung auftraten, zeigen.

10.2 Software

Gegen das Übergewicht zu Gunsten der Nullen in unserer Zufallssequenz erwies sich die in [RFC1750] beschriebene von Neumann-Methode als sehr geeignet. Erst die Kombination mit der den Bias korrigierenden Software macht unsere Hardware zu einem Random Bit Generator. Die Analyse der Daten ergab eine befriedigend grosse Entropie.

Im Sinne schwieriger erratbar können die Daten mittels Hashing verbessert werden. Diese Funktion sollte unbedingt in einen RNG eingebaut sein, falls sich die Zufallsdaten durch irgendeine Beeinflussung der Schaltung (Netzbrumm oder ähnlich) verschlechtern.

10.3 Statistik

Die Analyse der um den Bias korrigierten Daten mit dem UST zeigte, dass diese Daten zufällig sind.

Allerdings bestanden dieselben Daten folgende statistischen Tests nicht: Frequency Test, Runs Test und Serial Test.

Das Mischen der Daten mit dem MD5-Hash Algorithmus (gleich viele Input- wie Output-Bits) verbesserte die Statistik der Daten: Die gehashten Daten bestanden nun alle Tests. Auf die Sicherheit hat das keinen Einfluss, da ein Angreiffer hier auf die nicht gehashten Daten zielen wird.

Damit wagen wir die Behauptung: *Die erzeugten Zufallszahlen genügen den Anforderungen. Es sind keine gravierende Mängel zu erkennen.*

10.4 Datenrate

Anhand unseren Rohdaten (Ohne von Neumann-Korrektur) konnten wir sehen, dass über 99% der Bits Eins sind. Hier liegt das grösste Verbesserungspotential unseres Produkts: Kann die Zahl zufälliger Pulse erhöht werden, steigt die Datenrate massiv.

11 Schlusswort

Die Projektarbeit war unerwartet stark hardwarelastig. Die Analoge Schaltungstechnik mussten wir uns erst einmal erarbeiten. So verbrachten wir viel Zeit mit try and error, da immer wieder Effekte auftauchten, die wir nicht oder zuwenig beachtet haben.

Da die finale Hardware erst gegen das Ende der Arbeit zur Verfügung stand, bliebt gerade noch genügend Zeit um Zufallsdaten aufzuzeichnen. Damit verschob sich der Zeitpunkt, ab dem wir beginnen konnten die Daten zu analysieren gegen den Abgabetermin. Das führte zu einer hektischen Schlussphase.

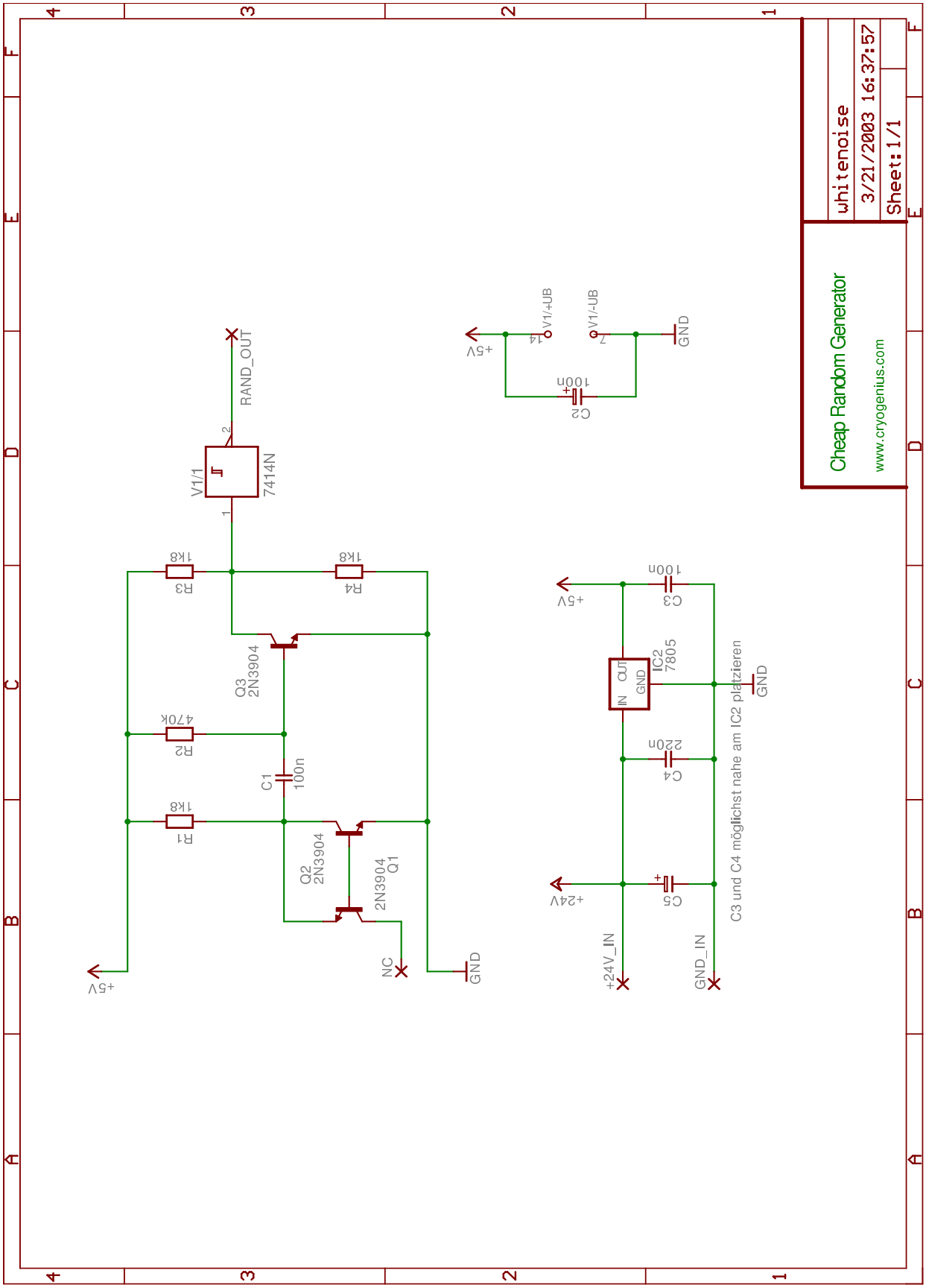
Ein interessanter Aspekt der Arbeit war die Interdisziplinarität: Wir beschäftigten uns mit Hardware, Statistik, Mikrocontrollern und Software, sowie den allgemeinen Grundlagen der Zufallszahlen.

Zusammenfassend: Eine nicht ganz einfache, aber durchaus interessante Arbeit.

12 Elektrische Schaltungen

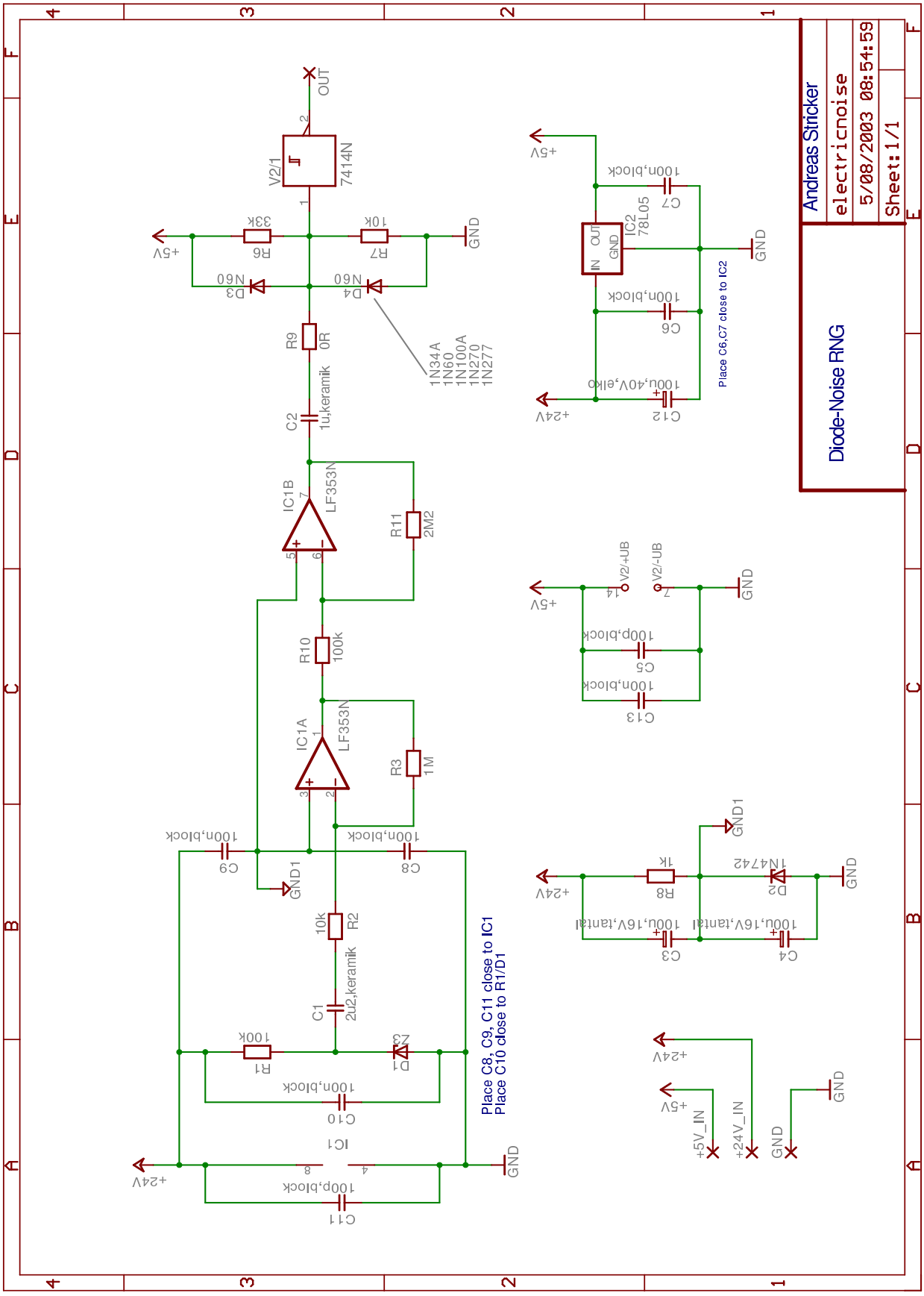
Im folgenden sind die Schaltungen aufgeführt, mit denen die Tests durchgeführt wurde sowie der Vorschlag für eine zukünftige Schaltung.

12.1 Transistor Schaltung



5/16/2003 10:27:14 f=0.99 /home/andy/eagle/Whitenose/whitenoise.sch (Sheet: 1/1)

12.2 Dioden-Rauschen

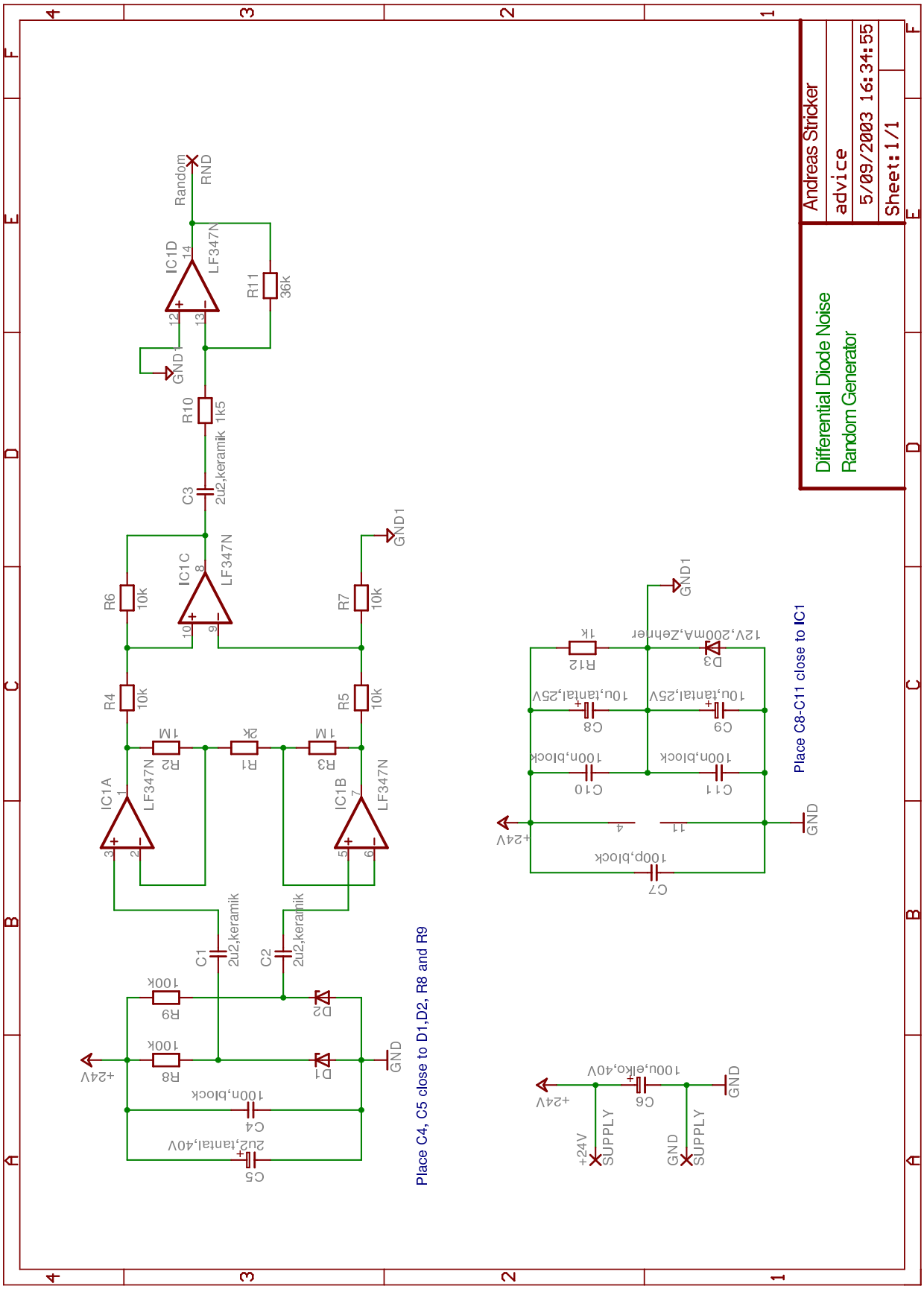


5/16/2003 10:06:53 f=0.99 /home/andy/eagle/Whitenose/electricnoise.sch (Sheet: 1/1)

Partlist exported from electricnoise.sch at 5/16/2003 10:28:07

Part	Value	Device	Package	Description
+5V_IN		2,54/0,8	2,54/0,8	THROUGH-HOLE PAD
+24V_IN		2,54/0,8	2,54/0,8	THROUGH-HOLE PAD
C1	2u2,keramik	CAPNP-5	C-5	
C2	1u,keramik	CAPNP-5	C-5	
C3	100u,16V,tantal	ELC-2,5L	ES-2,5L	
C4	100u,16V,tantal	ELC-2,5L	ES-2,5L	
C5	100p,block	CAPNP-2,5	C-2,5	
C6	100n,block	CAPNP-2,5	C-2,5	
C7	100n,block	CAPNP-2,5	C-2,5	
C8	100n,block	CAPNP-5	C-5	
C9	100n,block	CAPNP-5	C-5	
C10	100n,block	CAPNP-5	C-5	
C11	100p,block	CAPNP-2,5	C-2,5	
C12	100u,40V,elko	ELC-2,5L	ES-2,5L	
C13	100n,block	CAPNP-5	C-5	
D1	Z3	ZENER-DIODEZD-10	ZDIO-10	Z-Diode
D2	1N4742	ZENER-DIODEZD-10	ZDIO-10	Z-Diode
D3	N60	1N4004	DO41-10	DIODE
D4	N60	1N4004	DO41-10	DIODE
GND		2,54/0,8	2,54/0,8	THROUGH-HOLE PAD
IC1	LF353N	LF353N	DIL08	OP AMP
IC2	78L05	78LXX	78LXX	VOLTAGE REGULATOR
OUT		2,15/1,0	2,15/1,0	THROUGH-HOLE PAD
R1	100k	RESEU-12,5	R-12,5	
R2	10k	RESEU-12,5	R-12,5	
R3	1M	RESEU-12,5	R-12,5	
R6	33k	RESEU-12,5	R-12,5	
R7	10k	RESEU-12,5	R-12,5	
R8	1k	RESEU-12,5	R-12,5	
R9	0R	RESEU-12,5	R-12,5	
R10	100k	RESEU-12,5	R-12,5	
R11	2M2	RESEU-12,5	R-12,5	
V2	7414N	7414N	DIL14	Hex schmitt trigger INVERTER

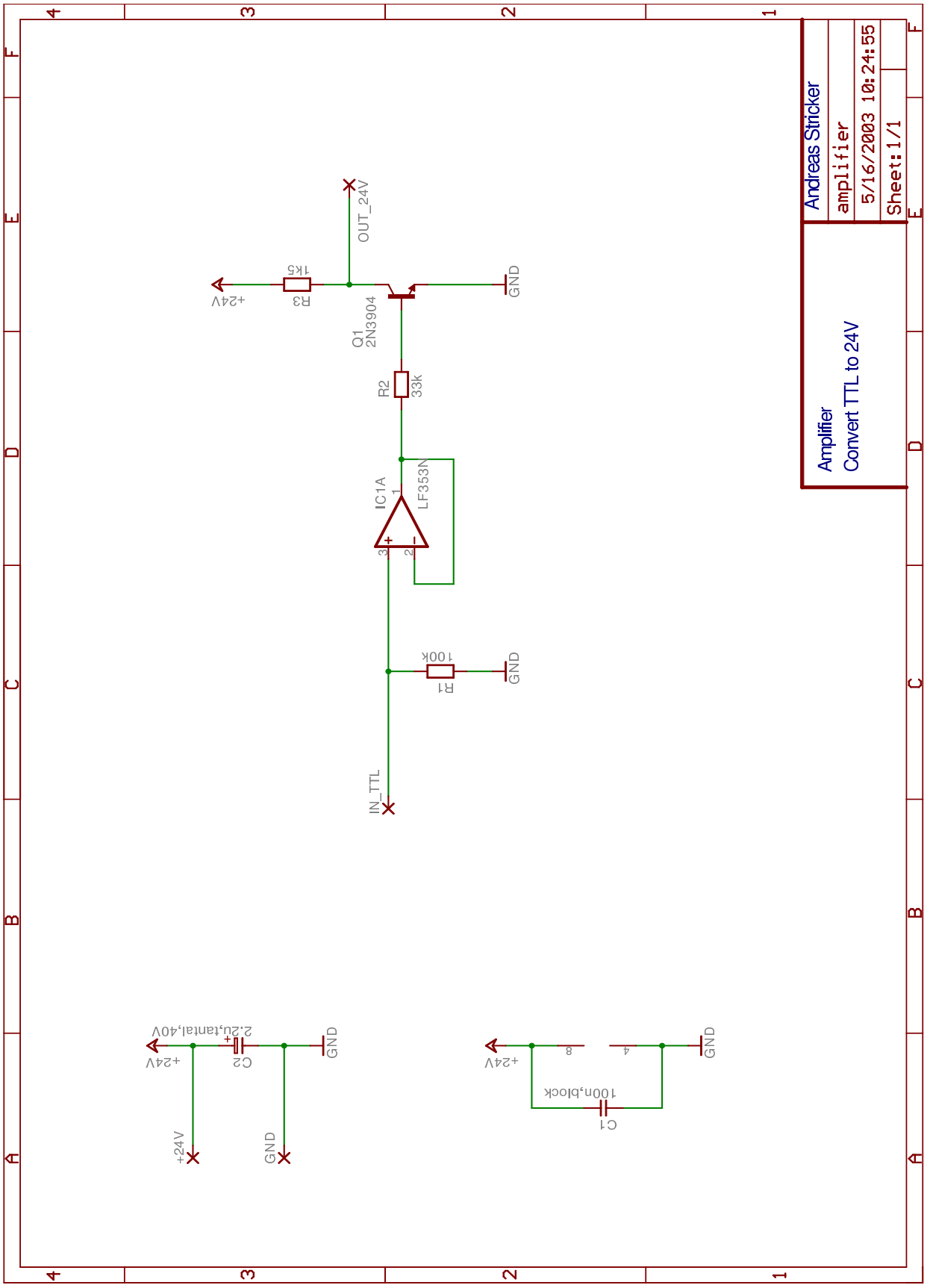
12.3 Vorschlag für störungsunterdrückende Schaltung



Partlist exported from advice.sch at 5/16/2003 10:05:14

Part	Value	Device	Package	Description
+24V	SUPPLY	2,15/1,0	2,15/1,0	THROUGH-HOLE PAD
C1	2u2,keramik	CAPNP-5	C-5	
C2	2u2,keramik	CAPNP-5	C-5	
C3	2u2,keramik	CAPNP-5	C-5	
C4	100n,block	CAPNP-2,5	C-2,5	
C5	2u2,tantal,40V	ELC-2,5	ES-2,5	
C6	100u,elko,40V	ELC-2,5L	ES-2,5L	
C7	100p,block	CAPNP-2,5	C-2,5	
C8	10u,tantal,25V	ELC-2,5L	ES-2,5L	
C9	10u,tantal,25V	ELC-2,5L	ES-2,5L	
C10	100n,block	CAPNP-5	C-5	
C11	100n,block	CAPNP-5	C-5	
D1		ZENER-DIODEDO35Z10	DO35Z10	Z-Diode
D2		ZENER-DIODEDO35Z10	DO35Z10	Z-Diode
D3	12V,200mA,Zehner	ZENER-DIODEP1-12	P1Z12	Z-Diode
GND	SUPPLY	2,15/1,0	2,15/1,0	THROUGH-HOLE PAD
IC1	LF347N	LF347N	DIL14	OP AMP
R1	2k	RESEU-5	R-5	
R2	1M	RESEU-5	R-5	
R3	1M	RESEU-5	R-5	
R4	10k	RESEU-5	R-5	
R5	10k	RESEU-5	R-5	
R6	10k	RESEU-5	R-5	
R7	10k	RESEU-5	R-5	
R8	100k	RESEU-5	R-5	
R9	100k	RESEU-5	R-5	
R10	1k5	RESEU-5	R-5	
R11	36k	RESEU-5	R-5	
R12	1k	RESEU-5	R-5	
RND	Random	2,15/1,0	2,15/1,0	THROUGH-HOLE PAD

12.4 Pegel-Anpassung TTL zu 24V


Andreas Stricker
amplifier
5/16/2003 10:24:55
Sheet: 1/1
Amplifier
Convert TTL to 24V
OUT_24V
+24V
R3
1K5
GND
Q1
2N3904
R2
33k
IC1A
LF353N
R1
100k
IN_TTL
2.2u,tantal,40V
C2
+24V
GND
+24V
GND
100n,block
C1
+24V
GND
5/16/2003 10:25:22 f=0.99 /home/andy/eagle/Whitenose/amplifier.sch (Sheet: 1/1)

12.5 Glossar

Bias
Offset der Sequenz. Übergewicht von Nullen oder Einsen.

Hash
Eine Hashfunktion ist eine Einwegfunktion. Einweg bedeutet, das ein Funktionswert nicht mit vertretbarem Aufwand einem Eingangswert zugewiesen werden kann.

Abkürzungen

ADC Analog-Digital-Converter: Analog-Digital-Wandler
BMS Binary Memory Source
LSB Least Significant Bit: Niederwertigstes Bit
MD5 Kryptografisch sichere Prüfsumme (Digest, Hash)
RNG Random Number Generator
PRNG Pseudo Number Random Generator
UST Universal Statistical Test

13 Literaturliste

[RFC1750] Randomness Recommendations for Security
RFC1750
Eastlake, Crocker & Schiller

[RFC1321] The MD5 Message-Digest Algorithm
RFC1321
R. Rivest, MIT Laboratory for Computer Science and RSA Data Security, Inc.

[MAURER] A Universal Statistical Test for Random Bit Generators
http://www.crypto.ethz.ch/research/srt/
Ueli Maurer, Professor of Computer Science

[TAOCP] The Art of Computer Programming, Volume 2, Seminumerical Algorithms
ISBN 0-201-89684-2
Donald E. Knuth

[AREINHOLD] Random Noise Sources from Diceware.com
http://world.std.com/~reinhold/truenoise.html
Arnold, Reinhold

[DIEHARD] DIEHARD, a battery of tests for random number generators
http://stat.fsu.edu/~geo/
George Marsaglia

[HOTBITS] HotBits: Genuine random numbers, generated by radioactive decay
http://www.fourmilab.ch/hotbits/

[ENT] ENT: A Pseudorandom Number Sequence Test Program
http://www.fourmilab.ch/random/

[R.ORG] random.org: True Random Number Service
http://www.random.org/

[CRYPTOOL] Cryptool: Demonstrations- und Referenzprogramm für Kryptographie
http://www.cryptool.de/

[NOISE] The Penfold Book
http://www.armory.com/~rstevew/Public/SoundSynth/00_PenfoldSynth/penoise.html

[BECK] Beck Buissness Center Homepage
http://www.bcl.de/

[RTOS] IPC@Chip Documentation @CHIP-RTOS
http://www.bcl.de/files/documentation/api/APIDOC0104.PDF

14 Quelltext Dokumentation

truerandom Data Structure Index

truerandom Data Structures

Here are the data structures with brief descriptions:

truerandom File Index

truerandom File List

Here is a list of all documented files with brief descriptions:

truerandom Data Structure Documentation

stat_values_t Struct Reference

Data Fields

double **expected_value**
double **variance**

Detailed Description

Datatype for table

The documentation for this struct was generated from the following file:
universal.c

test_params_t Struct Reference

`#include <universal.h>`

Data Fields

double **ftu**
the test resut fTU closly related to entropy.

size_t **parameter_Q**
the test parameter Q.

size_t **parameter_K**
the test parameter K.

size_t **parameter_L**
the test parameter L.

double **rejection_rate**
choosen rejection rate.

double **t1**
lower border for test result.

double **t2**
higher border for test result.

bool **accepted**
test result (accepted = true, rejected = false).

Detailed Description

Datatype of test result

The documentation for this struct was generated from the following file:
universal.h

truerandom File Documentation

bias.c File Reference

Module bias: Calculate bias of a sequence.

```
#include <stdlib.h>
#include "bias.h"
#include "stattest.h"
```

Functions

double **calc_bias** (buffer_t buf, size_t buf_len)

Calculate the bias of the buffer. The bias calculation is accelerated by using a precalculated look-up table.

Detailed Description

Module bias: Calculate bias of a sequence.
Calculate the bias of a sequence in buffer.

Author:
Andreas Stricker , Roland Siegert

Version:

Id:
bias.c,v 1.5 2003/05/15 07:54:54 stricand Exp

Function Documentation

double calc_bias (buffer_t *buf*, size_t *buf_len*)

Calculate the bias of the buffer. The bias calculation is accelerated by using a precalculated look-up table.

Parameters:
buf the buffer with random data
buf_len length of the buffer

Returns:
the bias of the data relatet to a single bit. True Random should return 0.5

bias.h File Reference

Module bias: Calculate bias of a sequence.

```
#include "stattest.h"
```

Functions

double **calc_bias** (buffer_t buf, size_t buf_len)

Calculate the bias of the buffer. The bias calculation is accelerated by using a precalculated look-up table.

Detailed Description

Module bias: Calculate bias of a sequence.
Calculate the bias of a sequence in buffer.

Author:
Andreas Stricker , Roland Siegert

Version:

Id:
bias.h,v 1.3 2003/05/15 07:54:54 stricand Exp

Function Documentation

double calc_bias (buffer_t *buf*, size_t *buf_len*)

Calculate the bias of the buffer. The bias calculation is accelerated by using a precalculated look-up table.

Parameters:

buf the buffer with random data
buf_len length of the buffer

Returns:

the bias of the data relatet to a single bit. True Random should return 0.5

driver.c File Reference

Functions concerning the random bit sequence.

```
#include <stdio.h>
#include <dos.h>
#include <driver.h>
#include <TCPIPAPI.H>
#include <TCPIP.H>
#include <CONIO.H>
#include <float.h>
#include "rtos.h"
#include "globals.h"
#include "md5.h"
#include "global.h"
#include "random.h"
```

Defines

#define **PFE_INT** 0xA2
#define **HAL_INT** 0xA1
#define **TCP_INT** 0xAC
#define **TASK_STACKSIZE** 1024
#define **DEBUG_DRIVER**
#define **DEBUG**
#define **SEND_DATA**
#define **MD_CTX** MD5_CTX
#define **MDInit** MD5Init
#define **MDUpdate** MD5Update
#define **MDFinal** MD5Final

Functions

void huge **fill_the_pool** (void)
void **init_dk40** (unsigned mask)
unsigned char **read_dk40_io** (void)
void **open_von_Neuman_src** (void)
while (rnd_buffer_pos< max_index_rndbuf)
void **create_pool_thread** (void)

Variables

int **sd**
char **rndbuf** [RND_BUFFER_SIZE]
char **poolbuf** [POOL_BUFFER_SIZE]
int **max_index_rndbuf**
int **in_index_pool**
int **out_index_pool**
int **pool_full**
int **md5_hashing**
unsigned **pin_mask**
kBytes to compute the amount of read random bytes just for debugging double **kBytes** = 0

inregs To initialize the microcontroller union REGS **inregs**
outregs To initialize the microcontroller union REGS **outregs**
segregs To initialize the microcontroller struct SREGS **segregs**
wPio, wInp, wOut To initialize the microcontroller unsigned **wPio**
wPio, wInp, wOut To initialize the microcontroller unsigned **wInp**
wPio, wInp, wOut To initialize the microcontroller unsigned **wOut**
pins To initialize the microcontroller unsigned **pins**
int **read_timeout** = READ_TIMEOUT
The semaphoreID of a semaphore that protects the pool.int **pool_semaphoreID**
pool_sem[4] A name for a semaphore that protects the pool.char **pool_sem** [4] = {'P','S','E','1'}
int **pool_thread_run**
unsigned int **task_stack** [TASK_STACKSIZE]
threadID The threadID.int **threadID**
TaskDefBlock The pool-thread context TaskDefBlock **pool_thread_defblock**
unsigned char **byte** = '\0'
int **shift_pos** = 0
int **rnd_buffer_pos** = 0
x

Detailed Description

Functions concerning the random bit sequence.
Tested with Borland C/C++ 5.02 on a SC12 with Bios version 0.64.

Author:
Roland Siegert , Andreas Stricker
Version:

Id:
driver.c,v 1.15 2003/05/15 20:09:02 siegerol Exp

Define Documentation

#define HAL_INT 0xA1

HAL == "Hardware Abstraction Layer"

#define PFE_INT 0xA2

PFE == "Pin Function Enabler"

#define SEND_DATA

In our Application we want random data to be sent to a PC-System

#define TASK_STACKSIZE 1024

to establish a thread a stack must be reserved

Function Documentation

fill_the_pool (void)

This function represents the poolt-hread. The random buffer is invoked to be filled. If specified the data of the random buffer is hashed. The random buffer is copied to the pool. The modulo incrementation of the in- out-indexes of the pool make it a circular buffer. By the SEND_DATA Flag we signalize the Data to be sent immediately. Note that we do not send the data out of the pool.

void init_dk40 (unsigned *mask*)

Prepares the microcontroller to read the input pins

Parameters:

mask Note that we let all 8 pins configure for input.

void open_von_Neuman_src (void)

Detects pairs "01" or "10" at the input pin.

Work of the inner while loop: for a "01" a 0-bit and for a "10" a 1-bit is saved in a byte.

Work of the outer while loop: when the byte is complete it is saved in the send buffer.

Variable Documentation

init_dk40 Prepares the microcontroller to read the input pins

Parameters:

mask To set a dk40-pin as an input pin, the corresponding bit of this mask must be 1.

TaskDefBlock The pool-thread context TaskDefBlock pool_thread_defblock

Initial value:

```
{
  fill_the_pool,
  {'P','O','O','L'},
  &task_stack[TASK_STACKSIZE],
  TASK_STACKSIZE*sizeof(int),
  0,
  26,
  0,
  0,0,0,0
}
```

int sd

extern defined socket descriptor

task_stack

The stack for the pool-thread (dynamic allocation is not available)

driver.h File Reference

Functions concerning the random bit sequence.

Defines

#define **DRIVER_H**

Functions

void **init_dk40** (unsigned mask)
unsigned char **read_dk40_io** (void)
void **open_von_Neuman_src** (void)
void huge **fill_the_pool** ()
void **create_pool_thread** (void)

Detailed Description

Functions concerning the random bit sequence.
Tested with Borland C/C++ 5.02 on a SC12 with Bios version 0.64.

Author:
Roland Siegert , Andreas Stricker
Version:

Id:
driver.h,v 1.8 2003/05/15 20:09:03 siegerol Exp

Function Documentation

void open_von_Neuman_src (void)

Detects pairs "01" or "10" at the input pin.
Work of the inner while loop: for a "01" a 0-bit and for a "10" a 1-bit is saved in a byte.
Work of the outer while loop: when the byte is complete it is saved in the send buffer.

entropie.c File Reference

Module entropy: Calculate entropy of the buffer.

```
#include <math.h>
#include "entropie.h"
#include "stattest.h"
#include "histogram.h"
```

Functions

double **calc_entropy** (size_t bufsize, histogram_t histogram)

Calculate entropy of buffer. Note that this mean we also need a histogramm even if we don't display it.

Detailed Description

Module entropy: Calculate entropy of the buffer.
Calculate entropy of the buffer.

Author:
Andreas Stricker , Roland Siegert

Version:

Id:
entropie.c,v 1.5 2003/05/16 14:07:33 stricand Exp

Function Documentation

double calc_entropy (size_t *bufsize*, histogram_t *histogram*)

Calculate entropy of buffer. Note that this mean we also need a histogramm even if we don't display it.

Parameters:
bufsize size of the buffer
histogram the previous generated histogram

Returns:
entropy of the buffer. Should be close to BLOCK_LENGTH.

See also:
build_histogram() (*p.48*)

entropie.h File Reference

Module entropy: Calculate entropy of the buffer.

```
#include <stdlib.h>
#include "stattest.h"
#include "histogram.h"
```

Functions

double **calc_entropy** (size_t bufsize, histogram_t histogram)

Calculate entropy of buffer. Note that this mean we also need a histogramm even if we don't display it.

Detailed Description

Module entropy: Calculate entropy of the buffer.
Calculate entropy of the buffer.

Author:
Andreas Stricker , Roland Siegert

Version:

Id:
entropie.h,v 1.4 2003/05/16 14:07:33 stricand Exp

Function Documentation

double calc_entropy (size_t *bufsize*, histogram_t *histogram*)

Calculate entropy of buffer. Note that this mean we also need a histogramm even if we don't display it.

Parameters:

bufsize size of the buffer
histogram the previous generated histogram

Returns:

entropy of the buffer. Should be close to BLOCK_LENGTH.

See also:

build_histogram() (*p.48*)

fileops.c File Reference

Module fileops: Read file into buffer.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdlib.h>
#include "fileops.h"
#include "stattest.h"
```

Functions

size_t **fill_buffer_from_file** (char *filename, buffer_t *buffer, size_t maxsize)

Fill Buffer with content of file, specified with "filename". Open the file and fill buffer with content. Note that the file size is not restricted because this function read never more than MAX_BUF_SIZE bytes from file.

Detailed Description

Module fileops: Read file into buffer.
Read a file into local buffer

Author:
Andreas Stricker , Roland Siegert

Version:

Id:
fileops.c,v 1.5 2003/05/15 07:54:54 stricand Exp

Function Documentation

size_t fill_buffer_from_file (char * *filename*, buffer_t * *buffer*, size_t *maxsize*)

Fill Buffer with content of file, specified with "filename". Open the file and fill buffer with content. Note that the file size is not restricted because this function read never more than MAX_BUF_SIZE bytes from file.

Parameters:

filename name of the file
buffer pointer to a buffer_t, later points to the data read
maxsize only read maxsize data

Returns:

0 if error or number of Bytes read from file (buffersize).

fileops.h File Reference

Module fileops: Read file into buffer.

```
#include "stattest.h"
```

Functions

size_t **fill_buffer_from_file** (char *filename, buffer_t *buffer, size_t maxsize)

Fill Buffer with content of file, specified with "filename". Open the file and fill buffer with content. Note that the file size is not restricted because this function read never more than MAX_BUF_SIZE bytes from file.

Detailed Description

Module fileops: Read file into buffer.

Read a file into local buffer

Author:

Andreas Stricker , Roland Siegert

Version:

Id:

fileops.h,v 1.4 2003/05/15 07:54:54 stricand Exp

Function Documentation

size_t fill_buffer_from_file (char * *filename*, buffer_t * *buffer*, size_t *maxsize*)

Fill Buffer with content of file, specified with "filename". Open the file and fill buffer with content. Note that the file size is not restricted because this function read never more than MAX_BUF_SIZE bytes from file.

Parameters:

filename name of the file
buffer pointer to a buffer_t, later points to the data read
maxsize only read maxsize data

Returns:

0 if error or number of Bytes read from file (buffersize).

globals.h File Reference

Global definition for program settings.

Defines

#define **GLOBALS** .H
#define **BYTE_SIZE** 8
#define **STD_CLIENTPORT** 7331
#define **STD_HOSTPORT** 7330
#define **POOL_BUFFER_SIZE** 1024
#define **RND_BUFFER_SIZE** 16
#define **DIGEST_BUFFER_SIZE** 16
#define **OPTION_FLAG** '-'
#define **MD5_OPTION** "-md5"
#define **HOST_PORT_OPTION** "-hp"
#define **CLIENT_PORT_OPTION** "-cp"
#define **PIN_OPTION** "-pin"
#define **READ_TIMEOUT** 10000
#define **OPEN_SOCKET_ERROR** 1
#define **TCPIP_CONNECT_ERROR** 2
#define **IP_ADRESS_ERROR** 3
#define **TCPIP_SEND_ERROR** 4

Detailed Description

Global definition for program settings.
Tested with Borland C/C++ 5.02 on a SC12 with Bios version 0.64.

Author:
Roland Siegert , Andreas Stricker
Version:

Id:
globals.h,v 1.16 2003/05/15 20:09:03 siegerol Exp

Define Documentation

#define BYTE_SIZE 8

The number of bits of a byte. Used by **open_von_Neuman_src**() (*p.50*) for its byte buffer.

#define CLIENT_PORT_OPTION "-cp"

commandline option for the client port

#define DIGEST_BUFFER_SIZE 16

The default size of the digest buffer. That is the output of the hashing. The MD5 hash function mixes an arbitrary number of bytes to 16 bytes.

#define HOST_PORT_OPTION "-hp"

commandline option for the host port

#define IP_ADRESS_ERROR 3

If there's an error at converting the commandline string to an ip-adress we signalize it with this constant.

#define MD5_OPTION "-md5"

commandline option to switch on MD5-hashing

#define OPEN_SOCKET_ERROR 1

If there's an error at opening a socket we signalize it with this constant

#define OPTION_FLAG '-'

Each option must start with this character

#define PIN_OPTION "-pin"

commandline option for the pin from where the RNG bit stream is to read

#define POOL_BUFFER_SIZE 1024

The pool buffer size. We chose a multiple of 128 bit.

#define READ_TIMEOUT 10000

commandline option for number of readings before the thread is removed

#define RND_BUFFER_SIZE 16

The default size of the random buffer. Should be the same as digest buffer's size.

#define STD_CLIENTPORT 7331

The default client port

#define STD_HOSTPORT 7330

The default client port

#define TCPIP_CONNECT_ERROR 2

If there's an error at connecting we signalize it with this constant

#define TCPIP_SEND_ERROR 4

If there's an error at sending data we signalize it with this constant

histogram.c File Reference

Module histogram: Build histogram of the buffer.

```
#include <stdio.h>
#include <stdlib.h>
#include "histogram.h"
#include "stattest.h"
```

Functions

histogram_t * **build_histogram** (buffer_t buffer, size_t bufsize, histogram_t *histogram)
void **print_histogram** (histogram_t histogram)

Detailed Description

Module histogram: Build histogram of the buffer.
Build a histogram of the buffer. One unit is a block.

Author:
Andreas Stricker , Roland Siegert
Version:

Id:
histogram.c,v 1.3 2003/05/15 07:54:54 stricand Exp

Function Documentation

histogram_t* build_histogram (buffer_t *buffer*, size_t *bufsize*, histogram_t * *histogram*)
Build a histogram from the buffer. Note: if parameter histogram points to NULL than new memory is allocated.

void print_histogram (histogram_t *histogram*)
Print histogram to screen

histogram.h File Reference

Module histogram: Build histogram of the buffer.

```
#include "stattest.h"
```

Defines

#define **HISTOGRAM_BAR_CHAR** "#"
#define **HISTOGRAM_WIDTH** 66

Typedefs

typedef unsigned long **histogram_t** [BLOCK_RANGE]

Functions

histogram_t * **build_histogram** (buffer_t buffer, size_t bufsize, histogram_t *histogram)
void **print_histogram** (histogram_t histogram)

Detailed Description

Module histogram: Build histogram of the buffer.
Build a histogram of the buffer. One unit is a block.

Author:
Andreas Stricker , Roland Siegert
Version:

Id:
histogram.h,v 1.2 2003/05/15 07:54:54 stricand Exp

Function Documentation

histogram_t* build_histogram (buffer_t *buffer*, size_t *bufsize*, histogram_t * *histogram*)
Build a histogram from the buffer. Note: if parameter histogram points to NULL than new memory is allocated.

void print_histogram (histogram_t *histogram*)
Print histogram to screen

random.c File Reference

Class to use the Random Generators connected to DK40.

```
#include <stdio.h>
#include <string.h>
#include <dos.h>
#include <TCPIPAPI.H>
#include <TCPIP.H>
#include <CONIO.H>
#include "rtos.h"
#include "globals.h"
#include "driver.h"
#include "md5.h"
#include "global.h"
```

Defines

#define **PFE_INT** 0xA2
#define **HAL_INT** 0xA1
#define **TCP_INT** 0xAC
#define **IP_ARG** 1
#define **MAX_IP_SIZE** 17
#define **DEBUG**
#define **SEND_DATA**

Functions

int **str_to_int** (char *str)
void **print_buffer** (char *info, char *buf)
void **print_usage** ()
int **get_ip_string** (char IPStr[], char *arguments[])
void **connection_error** (const int code, void *error, int socket_to_close)
int **get_arguments** (int argc, char *arguments[], char IPStr[], int *clientPort, int *hostPort, int *percentage, int *pin_mask)
int **main** (int argc, char *argv[])
Main Program. Parse Arguments, init sockets, listen to socket for connection and clean up.

Variables

char **rndbuf** [RND_BUFFER_SIZE]
char **poolbuf** [POOL_BUFFER_SIZE]
int **max_index_rndbuf**
int **in_index_pool**
int **out_index_pool**
int **pool_full**
int **md5_hashing**
unsigned **pin_mask**
sd The socket descriptor.int **sd**

Detailed Description

Class to use the Random Generators connected to DK40.
Class to use the Random Generators connected to DK40. Tested with Borland C/C++ 5.02 on a SC12 with Bios version 0.64.

Author:
Roland Siegert , Andreas Stricker
Version:

Id:
random.c,v 1.16 2003/05/15 20:09:03 siegerol Exp

Define Documentation

#define DEBUG
Debug Flag

#define HAL_INT 0xA1
HAL == "Hardware Abstraction Layer"-Interrupt

#define IP_ARG 1
The index at which the ip-adress argument must be placed

#define MAX_IP_SIZE 17
length of a string of an ip-address

#define PFE_INT 0xA2
PFE == "Pin Function Enabler"-Interrupt

#define SEND_DATA
In our Application we want random data to be sent to a PC-System

#define TCP_INT 0xAC
TCP-Interrupt

Function Documentation

void connection_error (const int *code*, void * *error*, int *socket_to_close*)
convenience method: prints information about errors that occurred while connecting to a client and closes the connection
Parameters:
code
error
socket_to_close

int get_arguments (int *argc*, char * *arguments*[], char *IPStr*[], int * *clientPort*, int * *hostPort*, int * *percentage*, int * *pin_mask*)
gets the arguments and options from the commandline
Parameters:

argc copy of the argc argument of the main method
arguments pointer to the argv[] argument of the main method
IPStr where the ip-address-string is put
ClientPort the ClientPort can be specified
HostPort the HostPort can be specified
md5_hashing activates MD5-hashing
percentage sets the size of the of the MD5-hashing input buffer in percentage of the output buffer caution: if MD5-hashing is activated this buffer should be greater or equal to the size of the digest buffer

int get_ip_string (char *IPStr*[], char * *arguments*[])

convenience method, used multiple times in the main method

Parameters:

IPStr the String that is to convert in an ip-address
arguments the pointer to the array of strings that holds the ip-String at the index IP_ARG; thats the "argv" parameter of the main method

int main (int *argc*, char * *argv*[])

Main Program. Parse Arguments, init sockets, listen to socket for connection and clean up.

Parameters:

argc Count of arguments stored in argv
argv Vector of argument strings

Returns:

Zero if no error.

void print_buffer (char * *info*, char * *buf*)

usefull to print debug- or user information out to stdout

Parameters:

info name of the buffer or other meta-information
buf the char buffer

void print_usage ()

prints information about the commandline arguments and options to stdout

int str_to_int (char * *str*)

converts a string into integer this function we need to convert commandline options such as client- & hostport from string to int types

Parameters:

str the string to convert

Variable Documentation

sd The socket descriptor. int sd

extern defined socket descriptor

readrand.c File Reference

Read Random Socket.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <unistd.h>
#include <sys/un.h>
#include <netdb.h>
#include <fcntl.h>
#include <getopt.h>
#include "readrand.h"
```

Functions

void **read_data** (int cfd)
Read data from socket and write it to std out file descriptor. Main loop who read data and write it to given destination.

void **listen_socket** ()
Wait for connection. Listen for a connection and then give the control to ***read_data****() (p.53).*

void **init_socket** ()
Initialize socket and bind it to port.

void **close_socket** ()
Close socket.

void **parse_arguments** (int argc, char *argv[])
Parse command line arguments. Reading parameters from command line using get_opts().

int **main** (int argc, char *argv[])
Main Program. Parse Arguments, init sockets, listen to socket for connection and clean up.

Variables

int **socket_fd**
sockaddr_in **addr**
bool **is_binary_output**
int **port_number**
int **display_column**

Detailed Description

Read Random Socket.

Listen on given port for a connection from target system, then read data from socket and write it to standart output

Author:
Andreas Stricker
Version:

Id:
readrand.c,v 1.6 2003/05/09 11:10:13 stricand Exp

Function Documentation

int main (int *argc*, char * *argv*[])

Main Program. Parse Arguments, init sockets, listen to socket for connection and clean up.

Parameters:
argc Count of arguments stored in argv
argv Vector of argument strings
Returns:
Zero if no error.

void parse_arguments (int *argc*, char * *argv*[])

Parse command line arguments. Reading parameters from command line using get_opts().

Parameters:
argc Count of arguments stored in argv
argv Vector of argument strings

void read_data (int *cfd*)

Read data from socket and write it to std out file descriptor. Main loop who read data and write it to given destination.

Parameters:
cfd socket file descriptor

readrand.h File Reference

Read Random Socket.

Defines

#define **PORT_NUMBER** 7331
#define **READ_BUFFER_SIZE** 512
#define **DISPLAY_COLUMN** 16
#define **MSG_ERR_ARG_NOPORT** "Port number missing\n"

#define **MSG_ERR_ARG_INVALID_PORT_NUMBER** "Invalid port number \"%s\"\n"
#define **MSG_ERR_ARG_WELLKNOWN_PORT** "%s is a wellknown port and you won't run this programm as root?\n"
#define **MSG_ERR_ARG_INVALID_COLUMN_NUMBER** "Invalid display column with \"%s\"\n"
#define **MSG_ERR_ARG_UNKNOWN** "Unknown argument \"%c\"\n"
#define **MSG_ERR_SOCKET_CREATE** "cannot initialize IP-socket\n"
#define **MSG_ERR_SOCKET_BIND** "failed binding socket to address\n"
#define **MSG_ERR_SOCKET_LISTEN** "not able to listen\n"
#define **MSG_ERR_SOCKET_CONNECTION** "not able to wait for connection\n"
#define **MSG_START** "Read Random Numbers over TCP/IP from Beck DK40@CHIP\n"
#define **MSG_LISTENING** "Listening on port %d for connection\n"
#define **MSG_HELP**
#define **true** 1
#define **false** 0

Typedefs

typedef char **bool**

Detailed Description

Read Random Socket.

Listen on given port for a connection from target system, then read data from socket and write it to standart output

Author:
Andreas Stricker
Version:

Id:
readrand.h,v 1.4 2003/04/09 17:10:28 stricand Exp

Define Documentation

#define MSG_HELP

Value:

```
"\nsyntax %s [-b] [-p PORT] [-c COLUMN] [-h]\n"\
"\n"\
"        -p PORT     Set listening port to PORT (default 7331)\n"\
"        -c COLUMN   Set columns a line in hex mode (default 16)\n"\
"        -b          Set to binary mode instead of hex output\n"\
"        -h          Show this help message\n\n"
```

stattest.c File Reference

Main part and user interface for statistic test software.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <getopt.h>
#include "stattest.h"
#include "fileops.h"
#include "bias.h"
#include "histogram.h"
#include "entropie.h"
#include "universal.h"
```

Functions

void **parse_arguments** (int argc, char *argv[], struct settings_struct *settings)
Parse command line arguments. Parsed command line arguments stored in structure settings.

int **main** (int argc, char *argv[])

Detailed Description

Main part and user interface for statistic test software.
This utility test a random source to its randomness. The central test is the "universal statistical test" Some important function like bias, entropy and histogram also included

Author:
Andreas Stricker , Roland Siegert
Version:

Id:
stattest.c,v 1.12 2003/05/16 14:07:33 stricand Exp

Function Documentation

int main (int *argc*, char * *argv*[])

Main function run the choosen tests

void parse_arguments (int *argc*, char * *argv*[], struct settings_struct * *settings*)

Parse command line arguments. Parsed command line arguments stored in structure settings.

Parameters:

argc Count of arguments from main
argv Vector of arguments from main
settings Pointer to settings structure where the results are stored.

stattest.h File Reference

Main part and user interface for statistic test software.

```
#include <unistd.h>
```

Data Structures

struct **settings_struct**

Defines

#define **MSG_START** "Statistical test for randomness, Andreas Stricker, Roland Siegert\n"
#define **MSG_QUOTE**
#define **MSG_ERR_ARG_REJECTRATE_NO_NUM** "Rejection rate option need a floating point argument\n"
#define **MSG_ERR_ARG_REJECTRATE_OUT_OF_RANGE** "Rejection rate out of range]0.0 - 0.5[\n"
#define **MSG_ERR_ARG_BLOCKSIZE_NO_NUM** "Blocksize L option need a number argument\n"
#define **MSG_ERR_ARG_BLOCKSIZE_OUT_OF_RANGE** "Blocksize is not in {8, 16}\n"
#define **MSG_ERR_ARG_MAXSIZE_NO_NUM** "maxsize option need a number argument\n"
#define **MSG_ERR_ARG_MAXSIZE_OUT_OF_RANGE** "maxsize out of range [1 - %ul]\n"
#define **MSG_ERR_ARG_UNKNOWN** "Unknown argument \"%c\"\n"
#define **MSG_ERR_ARG_NOFILE** "No file specified\n"
#define **MSG_ERR_FILE_NOT_ACCESSIBLE** "File \"%s\" not accessible, check for existence and permissions\n"
#define **MSG_ERR_NO_ENOUGH_MEMORY** "No enough memory to allocate the buffer.\n"
#define **MSG_ERR_CANT_READ_WHOLE_FILE** "Cannot read whole file. %d Bytes still to read.\n"
#define **MSG_ERR_CANT_OPEN_FILE** "Cannot open file. Please check permissions\n"
#define **MSG_INFO_FILE_TOO_BIG** "File is bigger than %1$d Bytes. Testing only first %1$d Bytes\n"
#define **MSG_HELP**
#define **true** 1
#define **false** 0
#define **BLOCK_LENGTH** 8
#define **BLOCK_RANGE** (0x01 << BLOCK_LENGTH)
#define **MAX_BUF_SIZE** 16*1024*1024
#define **DEFAULT_REJECTION_RATE** 0.01
#define **DEFAULT_BLOCKSIZE_L** 8
#define **DEFAULT_MAX_SIZE** 0

Typedefs

typedef char **bool**
typedef unsigned char **block_t**
typedef block_t * **buffer_t**
typedef unsigned char **UINT8**
typedef unsigned short int **UINT16**
typedef unsigned int **UINT32**

Detailed Description

Main part and user interface for statistic test software.
This utility test a random source to its randomness. The central test is the "universal statistical test" Some important function like bias, entropy and histogram also included

Author:
Andreas Stricker , Roland Siegert
Version:

Id:
stattest.h,v 1.12 2003/05/16 14:07:33 stricand Exp

Define Documentation

#define MSG_HELP

Value:

```
"\nsyntax: %s [OPTION] [-h] FILE\n"\
"PARAMETERS:\n"\
"        FILE                 File to test for randomness\n"\
"OPTIONS:\n"\
"      --entropy    -e       Only return entropy\n"\
"      --bias       -b       Only return bias\n"\
"      --histogram  -i       Only return histogram\n"\
"      --universal  -u       Only return universal statistical test\n"\
"      --rejectrate RATE     Set rejection rate to RATE used by\n"\
"                   -r RATE   universal statistical test (Default 0.001)\n"\
"      --blocksize  -L       Set blocksize L for universal statistical test\n"\
"      --fixedq     -Q       Set initialisation length to fixed size: 10*2^L\n"\
"      --maxsize SIZE        Only read first SIZE Bytes of file. It possible\n"\
"                   -m SIZE   to use 'k' for kilobytes and 'M' for megabytes\n"\
"      --help       -h       Show this help\n"\
"\n"
```

#define MSG_QUOTE

Value:

```
"A sequence is random, if the smallest description of it is the sequence"\
  " itself\n"
```

stddeviation.c File Reference

Standard deviation interpolation from table.

```
#include <stdio.h>
#include <math.h>
#include "stddeviation.h"
```

Defines

#define **TABLE_SIZE** 4000
#define **INDEX_DIVISOR** 1000

Functions

double **standard_deviation** (double x)
Interpolate standard deviation N(x) for x in [0..4] Lookup nearst values from x in the table and interpolates linear between them.

double **inv_standard_deviation** (double p)
Search the inverse of the function N(x). Binary search a y where 1.0 - N(y) = p/2.

Variables

double **standard_deviation_0_4** []

Detailed Description

Standard deviation interpolation from table.
Calculate standard deviation with a table an additional linear interpolation between values

Author:
Andreas Stricker , Roland Siegert
Version:

Id:
stddeviation.c,v 1.7 2003/05/15 07:54:54 stricand Exp

Function Documentation

double inv_standard_deviation (double *p*)

Search the inverse of the function N(x). Binary search a y where 1.0 - N(y) = p/2.

Parameters:
p the area under the normal distribution function
Returns:
the pos x where the area is = p/2
See also:

standard_deviation() (*p.59*)

double standard_deviation (double *x*)

Interpolate standard deviation N(x) for x in [0..4] Lookup nearst values from x in the table and interpolates linear between them.

Parameters:

x Value of Integrals N(x) between -infinity and x, must be in range [0..4]

Returns:

-1.0 if x is out of Range, else the value of the integral

See also:

inv_standard_deviation() (*p.59*)

Variable Documentation

double standard_deviation_0_4[]

Table with values from integral

$$N(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2}\, dt$$

for x = [0..4.0] with step 0.01 (4004 Values)

stddeviation.h File Reference

Standard deviation interpolation from table.

Functions

double **standard_deviation** (double x)

Interpolate standard deviation N(x) for x in [0..4] Lookup nearst values from x in the table and interpolates linear between them.

double **inv_standard_deviation** (double p)

Search the inverse of the function N(x). Binary search a y where 1.0 - N(y) = p/2.

Detailed Description

Standard deviation interpolation from table.

Calculate standard deviation with a table an additional linear interpolation between values

Author:

Andreas Stricker , Roland Siegert

Version:

Id:

stddeviation.h,v 1.4 2003/05/15 07:54:54 stricand Exp

Function Documentation

double inv_standard_deviation (double *p*)

Search the inverse of the function N(x). Binary search a y where 1.0 - N(y) = p/2.

Parameters:

p the area under the normal distribution function

Returns:

the pos x where the area is = p/2

See also:

standard_deviation() (*p.59*)

double standard_deviation (double *x*)

Interpolate standard deviation N(x) for x in [0..4] Lookup nearst values from x in the table and interpolates linear between them.

Parameters:

x Value of Integrals N(x) between -infinity and x, must be in range [0..4]

Returns:

-1.0 if x is out of Range, else the value of the integral

See also:

inv_standard_deviation() (*p.59*)

universal.c File Reference

Universal statistical test T_U.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "stattest.h"
#include "universal.h"
#include "stddeviation.h"
```

Data Structures

struct **stat_values_t**

Functions

void **universal_statistical_test_8** (buffer_t buffer, size_t block_size, struct **test_params_t** *test_results, bool fixed_Q)

void **universal_statistical_test_16** (buffer_t buffer, size_t block_size, struct **test_params_t** *test_results, bool fixed_Q)

test_params_t * **universal_statistical_test** (buffer_t buffer, size_t block_size, struct **test_params_t** *test_results, bool fixed_Q, int block_size_L)

The universal statistical test is performed here. The function tests the whole buffer with block_size size and write the results to the structure test_results.

double **ust_c_L_K** (int L, size_t K)

bool **is_universal_statistical_test_accepted** (struct **test_params_t** *test_result, double rejection_rate)

Test if the values are accepted. The test result follow normal distribution within a small range, given by the standard deviation that is specified by the rejection rate. This function test if the previous calculated test result is between the two borders t1 and t2 and return true if test is accepted or false if not.

void **print_universal_statistical_test** (struct **test_params_t** *test_result)

Variables

stat_values_t ust_value_table []

Detailed Description

Universal statistical test T_U.

The universal statistical test, developped by Ueli M.Maurer and first appeard in Journal of Cryptology, vol. 5, no. 2, 1992, pp. 89-105 test a sequence U^N of its randomness. The test is closly related to the entropy of the sequence and therefore able to show the redundance. The original Pascal sample code converted to C and enhanced to a full quality test.

Author:

Ueli M. Maurer , Andreas Stricker , Roland Siegert

Version:

Id:
universal.c,v 1.14 2003/05/16 14:07:33 stricand Exp

Function Documentation

bool is_universal_statistical_test_accepted (struct test_params_t * *test_result*, double *rejection_rate*)

Test if the values are accepted. The test result follow normal distribution within a small range, given by the standard deviation that is specified by the rejection rate. This function test if the previous calculated test result is between the two borders t1 and t2 and return true if test is accepted or false if not.

Parameters:

test_result conterns the result of test calculated by universal_statistical_test
rejection_rate specifies the rejection rate. The rate should be between]0, 0.5]

Returns:

true if test is accepted (and also set the field accepted in test_result)

See also:

universal_statistical_test() (*p.62*), **print_universal_statistical_test**() (*p.62*)

void print_universal_statistical_test (struct test_params_t * *test_result*)

Print result of universal statistical test readable to screen.

Parameters:

test_result a pointer to a structure **test_params_t** (*p.34*) with results of the test

struct test_params_t* universal_statistical_test (buffer_t *buffer*, size_t *block_size*, struct test_params_t * *test_results*, bool *fixed_Q*, int *block_size_L*)

The universal statistical test is performed here. The function tests the whole buffer with block_size size and write the results to the structure test_results.

Parameters:

buffer Buffer with random data of size block_size
block_size size of buffer
test_results pointer to struct of type **test_params_t** (*p.34*), filled with the result of test and also returned by this function
block_size_L Size of a block (8 or 16)

Returns:

result of test pointer to the same structure as parameter test_result

See also:

is_universal_statistical_test_accepted() (*p.62*), **print_universal_statistical_test**() (*p.62*)

double ust_c_L_K (int *L*, size_t *K*) `[inline]`

A helper function for factor c(L, K) c(L, K) = 0.7 - 0.8 / L + (4 + 32/L) K^(-3/L) / 15

Variable Documentation

struct stat_values_t ust_value_table[]

Initial value:

```
{

   { 0.7326495,    0.690    },
   { 1.5374383,    1.338    },
   { 2.4016068,    1.901    },
   { 3.3112247,    2.358    },
   { 4.2534266,    2.705    },
   { 5.2177052,    2.954    },
   { 6.1962507,    3.125    },
   { 7.1836656,    3.238    },
   { 8.1764248,    3.311    },
   { 9.1723243,    3.356    },
   { 10.170032,    3.384    },
   { 11.168765,    3.401    },
   { 12.168070,    3.410    },
   { 13.167693,    3.416    },
   { 14.167488,    3.419    },
   { 15.167379,    3.421    }
}
```

Table with pre-calculatet values for different block-sizes (see Maurer)

universal.h File Reference

Universal statistical test T_U.

```
#include <stdlib.h>
#include "stattest.h"
```

Data Structures

struct **test_params_t**

Functions

test_params_t * **universal_statistical_test** (buffer_t buffer, size_t block_size, struct **test_params_t** *test_results, bool fixed_Q, int block_size_L)

The universal statistical test is performed here. The function tests the whole buffer with block_size size and write the results to the structure test_results.

bool **is_universal_statistical_test_accepted** (struct **test_params_t** *test_result, double rejection_rate)

Test if the values are accepted. The test result follow normal distribution within a small range, given by the standard deviation that is specified by the rejection rate. This function test if the previous calculated test result is between the two borders t1 and t2 and return true if test is accepted or false if not.

void **print_universal_statistical_test** (struct **test_params_t** *test_result)

Detailed Description

Universal statistical test T_U.
Original Pascal Code converted to C

Author:

Ueli M. Maurer , Andreas Stricker , Roland Siegert

Version:

Id:

universal.h,v 1.10 2003/05/16 14:07:33 stricand Exp

Function Documentation

bool is_universal_statistical_test_accepted (struct test_params_t * *test_result*, double *rejection_rate*)

Test if the values are accepted. The test result follow normal distribution within a small range, given by the standard deviation that is specified by the rejection rate. This function test if the previous calculated test result is between the two borders t1 and t2 and return true if test is accepted or false if not.

Parameters:

test_result conterns the result of test calculated by universal_statistical_test
rejection_rate specifies the rejection rate. The rate should be between]0, 0.5]

Returns:

true if test is accepted (and also set the field accepted in test_result)

See also:

universal_statistical_test() (*p.62*), **print_universal_statistical_test**() (*p.62*)

void print_universal_statistical_test (struct test_params_t * *test_result*)

Print result of universal statistical test readable to screen.

Parameters:

test_result a pointer to a structure **test_params_t** (*p.34*) with results of the test

struct test_params_t* universal_statistical_test (buffer_t *buffer*, size_t *block_size*, struct test_params_t * *test_results*, bool *fixed_Q*, int *block_size_L*)

The universal statistical test is performed here. The function tests the whole buffer with block_size size and write the results to the structure test_results.

Parameters:

buffer Buffer with random data of size block_size
block_size size of buffer
test_results pointer to struct of type **test_params_t** (*p.34*), filled with the result of test and also returned by this function
block_size_L Size of a block (8 or 16)

Returns:

result of test pointer to the same structure as parameter test_result

See also:

is_universal_statistical_test_accepted() (*p.62*), **print_universal_statistical_test**() (*p.62*)

ust.c File Reference

Blockwise universal statistical test.

```
#include <math.h>
#include "ust.h"
```

Defines

#define **BLOCK_SIZE_L** 8
#define **BLOCK_RANGE_V** (0x01 << BLOCK_SIZE_L)
#define **INITIAL_STEPS_Q** (10*BLOCK_RANGE_V)
#define **EXPECTED_VALUE** 7.1836656f
#define **CONSTANT_C** ((float)BLOCK_SIZE_L - EXPECTED_VALUE)
#define **INVALID_SUM** 0xFF
#define **LN_2** 0.6931471806f
#define **LOOKUP_TABLE_SIZE** 1024

Functions

float **lookup_ln** (unsigned int x)
Calculate ln(x) for unsigned integer x with fast lookup-table. If x is smaller than 1024 the table is used, else the slowly function log() is called to calculate the value.

void **ust_init** (int *input_hash_size, int output_hash_size)
Initialize the universal statistical test. Only need to run once at start.

void **ust_add_block** (buffer_t buffer, size_t buf_size)
Add data blockwise to test. Calculate universal statistical test blockwise. Test starts new all NUMBER_OF_STEPS_Q bytes to make sure the test ist fast enough to changes of binary input source.

int **ust_get_hash_input_size** (int hash_size)
Calculate the amount of input bytes we need to hash. Calculate the amount of input bytes we need to hash to a hash function with hash_size size of digest.

Variables

size_t **tab** [BLOCK_RANGE_V]
table with 2^L entries.

float **sum** [2]
one valid sum and one currently used sum.

size_t **pos_n**
position (index n) in data.

size_t **initial_steps_Q**

initial steps (init. with INITIAL_STEPS_Q).

unsigned char **current_sum**
index of currently used sum.

unsigned char **valid_sum**
index of valid sum.

char **initialisation**
is initialisation phase?

int * **in_hash_size_ptr**
int **out_hash_size**
float **ln_lookup_table** [LOOKUP_TABLE_SIZE]

Detailed Description

Blockwise universal statistical test.
This test is able to calculate the test blockwise (stream) instead of buffer-wise. The function **ust_get_hash_input_size**() (*p.66*) return the size of hash needed to input if we need full cryptographic key size.

Author:
Andreas Stricker , Roland Siegert
Version:

Id:
ust.c,v 1.2 2003/05/16 12:22:03 stricand Exp

Function Documentation

float lookup_ln (unsigned int *x*)

Calculate ln(x) for unsigned integer x with fast lookup-table. If x is smaller than 1024 the table is used, else the slowly function log() is called to calculate the value.

Parameters:
x Value to calculate logarithm to base 2 from.
Returns:
logarithm from x to base 2.

void ust_add_block (buffer_t *buffer*, size_t *buf_size*)

Add data blockwise to test. Calculate universal statistical test blockwise. Test starts new all NUMBER_OF_STEPS_Q bytes to make sure the test ist fast enough to changes of binary input source.

Parameters:
buffer a block of data stream with size buf_size
buf_size size of buffer

See also:

ust_init() (*p.66*), **ust_get_hash_input_size**() (*p.66*)

int ust_get_hash_input_size (int *hash_size*)

Calculate the amount of input bytes we need to hash. Calculate the amount of input bytes we need to hash to a hash function with hash_size size of digest.

Parameters:

hash_size size of the digest produced by the hash function.

Returns:

Size of Digest.

See also:

PARANOID, **ust_init**() (*p.66*), **ust_add_block**() (*p.66*)

void ust_init (int * *input_hash_size*, int *output_hash_size*)

Initialize the universal statistical test. Only need to run once at start.

Parameters:

input_hash_size pointer to a variable that is filled with the needed amount of hashing input buffer size according to output_hash_size

output_hash_size Size of used digest. Needed to calculate input_hash_size

ust.h File Reference

Blockwise universal statistical test.

Defines

#define **NULL** 0
#define **NUMBER_OF_STEPS_K** 0xFFFF

Typedefs

typedef long int **size_t**
typedef unsigned char * **buffer_t**

Functions

void **ust_init** (int *input_hash_size, int output_hash_size)
Initialize the universal statistical test. Only need to run once at start.

void **ust_add_block** (buffer_t buffer, size_t buf_size)
Add data blockwise to test. Calculate universal statistical test blockwise. Test starts new all NUMBER_OF_STEPS_Q bytes to make sure the test ist fast enough to changes of binary input source.

int **ust_get_hash_input_size** (int hash_size)
Calculate the amount of input bytes we need to hash. Calculate the amount of input bytes we need

to hash to a hash function with hash_size size of digest.

Detailed Description

Blockwise universal statistical test.
This test is able to calculate the test blockwise (stream) instead of buffer-wise. The function **ust_get_hash_input_size**() (*p.66*) return the size of hash needed to input if we need full cryptographic key size.

Author:
Andreas Stricker , Roland Siegert
Version:

Id:
ust.h,v 1.1 2003/05/15 15:34:46 stricand Exp

Define Documentation

#define NUMBER_OF_STEPS_K 0xFFFF

This parameter specifies how many bytes are passed throuh one test until the test is restarted. More means more accurancy, but slower reaction. I recommend to use a value between 0x8000 and 0xFFFF

Function Documentation

void ust_add_block (buffer_t *buffer*, size_t *buf_size*)

Add data blockwise to test. Calculate universal statistical test blockwise. Test starts new all NUMBER_OF_STEPS_Q bytes to make sure the test ist fast enough to changes of binary input source.

Parameters:
buffer a block of data stream with size buf_size
buf_size size of buffer
See also:
ust_init() (*p.66*), **ust_get_hash_input_size**() (*p.66*)

int ust_get_hash_input_size (int *hash_size*)

Calculate the amount of input bytes we need to hash. Calculate the amount of input bytes we need to hash to a hash function with hash_size size of digest.

Parameters:
hash_size size of the digest produced by the hash function.
Returns:
Size of Digest.
See also:
PARANOID, **ust_init**() (*p.66*), **ust_add_block**() (*p.66*)

void ust_init (int * *input_hash_size*, int *output_hash_size*)

Initialize the universal statistical test. Only need to run once at start.

Parameters:

input_hash_size pointer to a variable that is filled with the needed amount of hashing input buffer size according to output_hash_size

output_hash_size Size of used digest. Needed to calculate input_hash_size