# Raport z wykonania zadania “Sprzętowy generator liczb losowych na podstawie mikroprocesora RP2040” Laboratorium Bezpieczeństwa Systemów Teleinformatycznych.

**Wykonał:** Krystian Kacprzak

**Data oddania:** 12.04.2021

## Podstawa opracowania:

B. Kristinsson, “The Arduino as a Hardware Random-Number Generator”, 2012, arXiv.org, cat. Computer Science, Cryptography and Security

## Systematyczny przegląd literatury:

1. Baza artykułów naukowych “e-zasoby” Politechniki Poznańskiej
2. Słowa kluczowe “arduino”, “random number generator”, “hardware”
3. Wynik wyszukiwania - “The Arduino as a Hardware Random-Number Generator”
4. Przejście na stronę arXiv.org

## Analiza źródła entropii:

Raspberry Pi Pico (nazywane dalej RPi lub Pico) wyposażone jest w trzy piny analogowe oraz wewnętrzny konwerter analogowo-cyfrowy (ADC). Piny analogowe odczytują wartość napięcia podanego na nie w stosunku do napięcia zasilania i podają ten stosunek jako 16-bitową liczbę całkowitą.

Niepodłączone do niczego piny analogowe zbierają śladowe ilości szumu elektronicznego, co przekłada się na pseudolosowe wartości, zależne od temperatury otoczenia, bliskości urządzeń emitujących pole elektromagnetyczne, czy nawet powierzchni, której dotykają. Bazowymi próbkami do stworzenia wartości prawdziwie losowych będą właśnie pseudolosowe wartości odczytane z nie podłączonych pinów analogowych.

Proces pobierania i zapisywania danych będzie oparty na schemacie “read-save-dump”. Dwa rdzenie procesora będą jednocześnie, w odstępie 4 ms pobierać dane z osobnych pinów analogowych (core0 - ADC0, core1 - ADC1), następnie naprzemiennie zapisywać wartości binarne lub dziesiętne (jest to dowolne) do tablicy. Po wygenerowaniu 1000 wartości tablica zostaje dopisana do pliku .txt na karcie pamięci, wyczyszczona i proces generowania zaczyna się od nowa. W efekcie liczba generowanych próbek może być tylko wielokrotnością liczby 1000, jednak w przypadku utraty zasilania tracimy mniej niż 1000 próbek.

Wewnętrzny ADC opiera się na napięciu zasilania płytki Pico, dostarczanej przez wewnętrzny zasilacz impulsowy, następnie poddanemu filtracji dolnoprzepustowej. W efekcie odczyt pinów analogowych nie jest stuprocentowo dokładny, cytując specyfikację Pico - “We can only do so much filtering, and therefore ADC_AVDD (zasilanie ADC - przyp.) will be somewhat noisy”. W związku z zaszumieniem napięcia odniesienia odczytane na pinach analogowych wartości także mają w sobie znamiona szumu. W związku z tym odczytane napięcie jest zwykle o 30 mV wyższe niż faktyczne (“sampling offset”).

Aby otrzymać stałe, pewne wyniki program generujący próbki uruchamiany jest manualnie z

poziomu aplikacji Thonny, zasilanie jest dostarczane do Pico poprzez kabel USB-A - microUSB, będący także medium przesyłowym danych debugujących pomiędzy komputerem a RPi. Do pomiarów zostały wykorzystany poniższy sprzęt komputerowy:

- laptop Lenovo Thinkpad T460s z procesorem Intel Core i5-6300U, system operacyjny Windows 10 Pro 20H2, kompilacja 19042.867

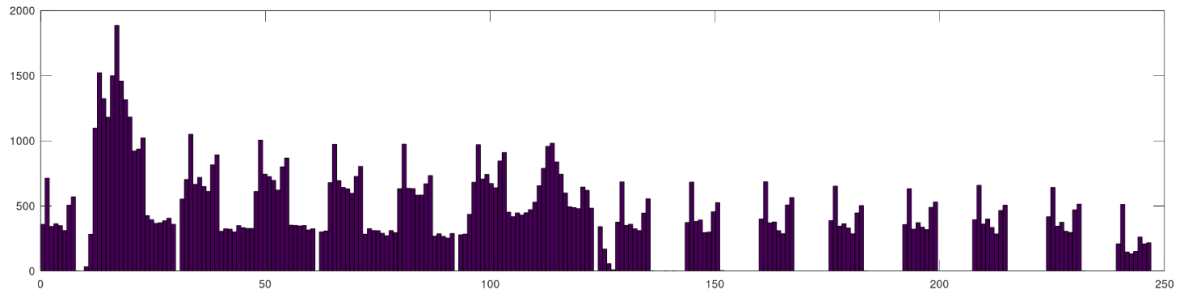
Próbki 16-bitowe zapisywane są do pliku .bin, z którego można następnie odczytywane są w postaci liczb 8-bitowych.

Najlepsze wyniki surowych próbek otrzymano utrzymując Pico w stałej wysokiej temperaturze 60 stopni Celsjusza. Poniżej przedstawiono wynik generowania 100 000 próbek:

- Czas generowania próbek:

1173196 ms = 1173,196 s = 19 m 33 s

- Histogram:




Entropia wyliczona ze wzoru $e = -\sum_i p_i log_2(p_i)$ dla powyższego rozkładu 8-bitowego wynosi **7,3839** bita.

## Metoda poprawy właściwości statystycznych

Jak wykazał poprzedni punkt, otrzymane próbki nie są liczbami całkowicie losowymi. Aby uzyskać liczby prawdziwie losowe należy przeprowadzić post-processing, stosując jeden z wielu dostępnych algorytmów randomizujących.

W przypadku tego projektu zastosowanym algorytmem jest Mersenne Twister. Algorytm ten jest zastosowany w zaimplementowanym w języku MicroPython module “urandom” w postaci uproszczonej względem modułu “random” klasycznego języka Python.

Moduł urandom przyjmuje zadane mu ziarno, następnie na podstawie tego ziarna generuje losową liczbę o zadanej długości bitowej.

W tym projekcie Pico pobiera wygenerowane wcześniej próbki pseudolosowe i używa ich jako ziarna. Każda próbka staje się ziarnem, z którego następnie oba rdzenie RPi jednocześnie generują po 10 liczb 16-bitowych. Wynikowe liczby 16-bitowe są “sklejane” ze sobą do postaci 10 liczb 32-bitowych i zapisywane do oddzielnego pliku. W efekcie każda

16-bitowa próbka pseudolosowa jest ziarnem dla 10 losowych liczb 32-bitowych.

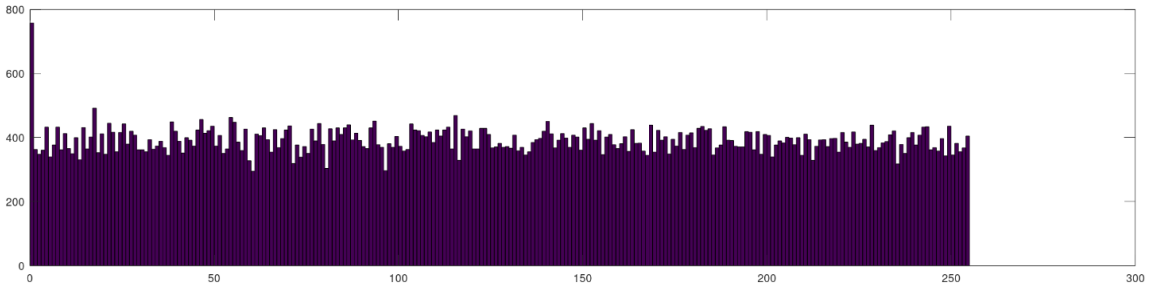
Próbki 32-bitowe zapisywane są do pliku .bin, z którego można następnie odczytywane są w postaci liczb 8-bitowych.

Poniżej przedstawiony jest wynik post-processingu 100 000 próbek 16-bitowych, czego wynikiem jest 1 000 000 próbek 32-bitowych, lub 4 000 000 próbek 8-bitowych.

- Czas post-processingu:

4967,642 s = 1 h 22 m 47 s

- Histogram dla 1 000 000 próbek 8-bitowych:




Entropia wyliczona ze wzoru $e = -\sum_i p_i log_2(p_i)$ dla powyższego rozkładu 8-bitowego wynosi **7,9874** bita.

## Uwagi:

1. Najwyższa uzyskana entropia wynosiła **7,9889** i wyliczona została na podstawie 4 000 000 przetworzonych próbek 8-bitowych
2. Temperatura otoczenia ma znaczny wpływ na jakość generowanych danych surowych:
    a. W temperaturze pokojowej 21 stopni Celsjusza entropia danych surowych wyniosła **5,4703**
    b. W temperaturze 4 stopni Celsjusza entropia danych surowych wyniosła **4,3684**
    c. W temperaturze 60 stopni Celsjusza entropia danych surowych wyniosła **7,3839**
3. Zastosowana metoda post processingu jest bardzo skuteczna, nawet w przypadku niskiej entropii danych wejściowych:
    a. Entropia przetworzonych danych z temperatury pokojowej 21 stopni Celsjusza wyniosła **7,9692**
    b. Entropia przetworzonych danych z temperatury 4 stopni Celsjusza wyniosła **7,9426**
    c. Entropia przetworzonych danych z temperatury 60 stopni Celsjusza wyniosła **7,9874**

4. Generowanie danych jak i post processing może być wykonany bez użycia komputera, co umożliwia stworzenie z Pico przenośnego “tokena” do generowania liczb losowych

### Kody programów:

- Kod programu odczytującego 100 000 próbek pseudolosowych z analogowych pinów Raspberry Pi Pico, używając dwóch rdzeni procesora:

```
import machine
import utime
import _thread
import gc
import sdcard
import uos

analog_C0 = machine.ADC(26)
global toWrite_C0

analog_C1 = machine.ADC(27)
global toWrite_C1

rng = []

global core1
global core0
core0 = 1
core1 = 1

led = machine.Pin(25, machine.Pin.OUT)

sd_spi = machine.SPI(1, sck = machine.Pin(10, machine.Pin.OUT), mosi =
machine.Pin(11, machine.Pin.OUT), miso = machine.Pin(12,
machine.Pin.OUT))
sd = sdcard.SDCard(sd_spi, machine.Pin(9))

uos.mount(sd, "/sd")
print("Mounted")

print("Size: {} MB".format(sd.sectors/2048))

print(uos.listdir("/sd"))

rawName = "sd/rng_data.txt"
rawFile = open(rawName, "w")

def readRNGcore0():
```

```
    global toWrite_C0
    global core0
    toWrite_C0 = ""
    read_C0 = analog_C0.read_u16()
    while read_C0 is 0:
        read_C0 = analog_C0.read_u16()
    toWrite_C0 += "{}".format(read_C0)
    toWrite_C0 += "\n"
    core0 += 1

def readRNGcore1():
    global toWrite_C1
    global core1
    toWrite_C1 = ""
    read_C1 = analog_C1.read_u16()
    while read_C1 is 0:
        read_C1 = analog_C0.read_u16()
    toWrite_C1 += "{}".format(read_C1)
    toWrite_C1 += "\n"
    core1 += 1

start = utime.ticks_ms()

for i in range(0, 50001):
    _thread.start_new_thread(readRNGcore1, ())
    utime.sleep_ms(1)
    readRNGcore0()
    rng.append(toWrite_C0)
    rng.append(toWrite_C1)
    if i%100 is 0:
        print(i)
    if i%500 is 0 and i is not 0:
        led.value(1)
        print("Writing to file")
        for x in rng:
            rawFile.write(x)
        rawFile.flush()
        print("Flushed")
        rng = []
        utime.sleep_ms(2)
        led.value(0)
    utime.sleep_ms(3)

print("Randomising done")
print("Generated samples:")
print(i*2)
```

```
print("Time took [s]:")
stop = utime.ticks_ms()
time = "{}".format(utime.ticks_diff(stop, start)/1000)
print(time)
rawFile.close()
gc.collect()
uos.umount("/sd")
for i in range(0, 6):
    led.value(1)
    utime.sleep(1)
    led.value(0)
    utime.sleep(1)
_thread.exit()
```

*MicroPython for RP2, v1.14-121 (unstable)*

- Kod programu post-processingowego, wykorzystujący moduł urandom by na dwóch rdzeniach procesora opierając się na ziarnie z próbki pseudolosowej wygenerować 10 losowych liczb 32-bit na każdą próbkę:

```
import machine
import utime
import gc
import sdcard
import uos
import urandom
import _thread

global output_1
global output_2

global pp
pp = []

led = machine.Pin(25, machine.Pin.OUT)

sd_spi = machine.SPI(1, sck = machine.Pin(10, machine.Pin.OUT), mosi =
machine.Pin(11, machine.Pin.OUT), miso = machine.Pin(12,
machine.Pin.OUT))
sd = sdcard.SDCard(sd_spi, machine.Pin(9))

uos.mount(sd, "/sd")
print("Mounted")

print("Size: {} MB".format(sd.sectors/2048))

print(uos.listdir("/sd"))
```

```
rawName = "sd/rng_data_60C_Laptop.txt"
rawFile = open(rawName)
ppName = "sd/final_data.txt"
ppFile = open(ppName, "w")

def core0gen():
    global output_1
    output_1 = urandom.getrandbits(16)

def core1gen():
    global output_2
    output_2 = urandom.getrandbits(16)

def scramble(number):
    global pp
    global output_1
    global output_2
    urandom.seed(number)
    for i in range(0, 10):
        _thread.start_new_thread(core1gen, ())
        core0gen()
        scrambledData = "{:0>16}".format(bin(output_1)[2:18]) +
"{:0>16}".format(bin(output_2)[2:18])
        toSave = str(int(scrambledData, 2))
        pp.append(toSave)

start = utime.ticks_ms()

i = 0
for lines in rawFile:
    scramble(int(lines))
    i += 1
    if i%50 is 0:
        print(i)
        for lines in pp:
            ppFile.write(lines)
            ppFile.write("\n")
        pp = []
        ppFile.flush()

print("Post-processing done")
print("Time took [s]:")
stop = utime.ticks_ms()
time = "{}".format(utime.ticks_diff(stop, start)/1000)
print(time)
```

```
rawFile.close()
ppFile.close()
gc.collect()
uos.umount("/sd")
for i in range(0, 6):
    led.value(1)
    utime.sleep(1)
    led.value(0)
    utime.sleep(1)
```

*MicroPython for RP2, v1.14-121 (unstable)*

- Kod GNU Octave wyznaczający histogramy danych surowych oraz przetworzonych w formie 16- i 32-bitowej:

```
clear;
data_raw = load('rng_data_21C_Laptop.txt');
data_final = load('final_data_21C_Laptop.txt');

subplot(2,1,1)
  hist(data_raw, 100)
subplot(2,1,2)
  hist(data_final, 100)
```

*GNU Octave 6.2.0, configured for “x86_64-w64-mingw32”*

- Kod GNU Octave wyznaczający histogramy danych surowych oraz przetworzonych w formie 8-bitowej:

```
clear;
pkg load communications;
data_raw = load('rng_data_60C_Laptop.txt');

fid_raw = fopen('data_raw_60C.bin','w');
fwrite (fid_raw, data_raw,'uint16');
fclose (fid_raw);

fid_raw = fopen('data_raw_60C.bin','r');
data8_raw = fread (fid_raw, 100000, 'uint8');
fclose (fid_raw);

data_final = load('final_data_60C_Laptop.txt');

fid_final = fopen('data_final_60C.bin','w');
fwrite (fid_final, data_final,'uint32');
fclose (fid_final);
```

```
fid_final = fopen('data_final_60C.bin','r');
data8_final = fread (fid_final, 4000000, 'uint8');
fclose (fid_final);

a=data8_raw;
b=data8_final;
n = 8;
m = 2^n-1;

subplot(2,1,1)
hist(a,m);
subplot(2,1,2)
hist(b,m);

[NN,XX]=hist(a,m);
[MM,YY]=hist(b,m);
NN = NN+1;
MM = MM+1;
p_raw=NN/sum(NN);
p_final=MM/sum(MM);
ent_raw = -sum(p_raw.*log2(p_raw))
ent_final = -sum(p_final.*log2(p_final))
```

*GNU Octave 6.2.0, configured for “x86_64-w64-mingw32”*