True random number generator server with ESP32

Levy G. da S. Galvão[1] e Victor A. Ferraz[2]

***Abstract*— Regarding the importance of true random number generation, this work proposes a network topology of a true random number generator (TRNG) from a ESP32 hardware with wireless connection that can be accessed by Python clients within a wireless local area network (WLAN) and return 32 bits random numbers.**

I. INTRODUCTION

The area of random number generation (RNG) has a wide range of applications. Such as in statistics, cryptography, computer simulations of various kinds, including communication transmissions etc.

RNGs can be divided in two major types: deterministic random number generator or pseudo random number generator (PRNG) and non-deterministic random number generator or true random number generator (TRNG) or physical-RNG [1].

PRNG generates its sequences from deterministic algorithms previous programmed in a machine. Considering the same algorithm is being used, it is more likely to repeat the random sequence. These patterns starts with 'seeds' numbers and the variation of the 'seed' results in different sequences. Once the number of 'seeds' is limited the numbers generated are not truly random [2].

The idea of TRNGs involve the fact of such device that can generate a sequence of numbers that have no relation to each other and such sequence cannot be repeated. Therefore a ideal source is capable of generate long sequences of perfectly independent bits [3].

TRNG rely in measuring physical processes that are unpredictable, such as: thermal noise, noise power level in radio-frequency receiver, photoelectric effect or quantum phenomena [4].

In that way, this works objective is to develop a network topology composed by a TRNG server that can be accessed remotely by any client connected in the same wireless local area network (WLAN) and provide an array of true random numbers of 32 bits for any kind of need.

II. DEVELOPMENT

A. Network topology

The hardware of choice for the TRNG server is a ESP32 from Espressif. The ESP32 is capable of Wi-Fi communication and most important, it has a built in function `esp_random()` for random number generation. The `esp_random()` function access a specialized RNG hardware that when Wi-Fi or Bluetooth are enabled, numbers returned can be considered true random, but when disabled, they are simply pseudo random [5].

Espressif warns that care must be taken when reading random values between the start of the main application and the initialization of Wi-Fi or Bluetooth drivers, since in startup ESP-IDF bootloader seeds the hardware RNG with entropy and the values may not be true random in this window[5].

The client that will be able to access the server and retrieve a vector of true random numbers is a application build in Python 3. This programming language was chosen because its high execution efficiency, clean code and it is a mainstream programming language in latest applications.

The network topology is shown in the figure 1. The Python client and the ESP32 server are both connected to a wireless router in the same WLAN. That can be more than one client, but each one can communicate one at the time with the ESP32.

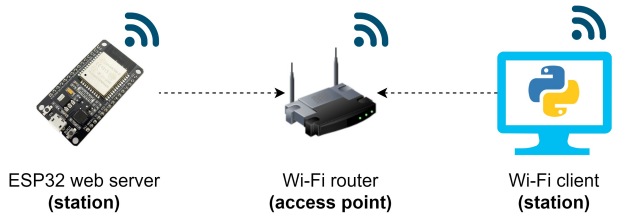

Fig. 1. Application network topology. Source: own.

The requirements to establish communication between server and client is:

1) Server and client(s) must be connected to the same router;
2) The ESP32 server is in idle mode (there is no client connected to the server);
3) The client must know the ESP32 server IP address in the local network and the exact port to connect;

The following sections will explain how the packet exchange is made and how the environments are initialized.

B. ESP32 server implementation

1) Initialization: The server application was implemented in the Arduino IDE and the resulting firmware was uploaded in a ESP-WROOM-32 Dev Module. The only library used was the `WiFi.h` that allows us to handle with server and client objects.

*This work was not supported by any organization

[1] Electrical engineering undergraduate, Universidade Federal do Rio Grande do Norte, Brazil.

[2] Electrical engineering Ph.D, Universidade Federal do Rio Grande do Norte, Brazil.

In the first section of the code the network credentials (service set identifier a.k.a SSID and its password) of the WLAN are initialized as global constants and the server object is instantiated with a free arbitrary port so the web server object is capable to listen for clients. In the code snippet below the SSID and password must be changed to a valid string.

```
#include <WiFi.h>

const char* ssid = "YOUR_NETWORK_ID";
const char* password = "YOUR_NETWORK_PASSCODE";
const uint16_t port = 80;
WiFiServer wifiServer(port);
```

2) `void setup():` This function is responsible for initialize the serial communication for debugging via the serial monitor:

```
  Serial.begin(115200);
  delay(1000);
```

Also this function connect the ESP32 server with the router and initialize the server. The connection is made via a `while` loop that tests if the WiFi status does not correspond to a valid connection (`WL_CONNECTED`). Inside the loop the networks credentials are used in the function `WiFi.begin` to establish connection. The code snippet below illustrates the whole loop plus the connection try, a serial message and a structural delay:

```
  while (WiFi.status() != WL_CONNECTED) {
    Serial.println("Connecting to WiFi..");
    WiFi.begin(ssid, password);
    delay(1000);
  }
```

Still within the setup function, some serial messages are printed to ensure the connection, including the display of the ESP32 server IP address in the local network (essential so the client can connect with the server) and the server is started with the `wifiServer.begin()` method of the server object.

```
  Serial.println("Connected to the WiFi network");
  Serial.println(WiFi.localIP());
  wifiServer.begin();
```

3) `void loop():` This function is responsible to handle the communication with the client.

The first line of code executes the function `wifiServer.available()` from the server object which returns a client. This `WiFiClient` can be use to evaluate a if-statement to engage packets exchange.

```
  WiFiClient client = wifiServer.available();
```

If the client object is available the code proceeds to communication. Before the actual exchange, two while loops are tested. The outer loop tests if the client still connected with a `client.connected()` flag. The inner loop tests if the client still sending bytes with the function `client.available()` that returns the number of bytes sent by the client and test if it is higher than zero.

Inside both the while loops the packets exchange occurs. First the server reads a string from the client and transforms it in a integer with the function `client.readString().toInt()`. This value is stored in a variable the tracks the number of 32 bits true random numbers that the client is asking for. So inside a for loop the server sends as many randoms as the client asked printing the random number as string in the client object with the command `client.println(esp_random())`. Once all the random numbers are sent, the connection is terminated with the command `client.stop()`. Serial messages and structural delays are also used along the code, and the code snippet of the if-statement returning true is:

```
  if(client){
    Serial.println("Status: Client available. \
    nEngaging connection...");
    while(client.connected()){
      while(client.available()>0){
        uint32_t sizeRnd = client.readString().
    toInt();
        for(unsigned itr=0; itr<sizeRnd; itr++){
    client.println(esp_random());}
      }
      delay(10);
    }
```

The code snippet below returns when the if-statement is false and and its purpose is only cosmetic. After that the function ends and repeat itself until the ESP32 is shutdown.

```
  } else{
    delay(1000);
    Serial.println("Status: IDLE.");
  }
```

C. Python client implementation

The Python application depends majorly in handling with internet sockets with the library *socket*. All the communication with the server is built inside a function that can be called as many times as possible inside the Python script.

Also is mandatory that the code contains variables that stores the ESP32 server IP address in the local network (`HOST` in the code) and the port that the server is listen in (`PORT` in the code). The code snippet below show all initialization that occur inside the function including a if-statement ensuring that the size sent to the server is a positive integer.

```
def TRNG_ESP32(size):
    HOST = 'YOUR_NETWORK_ID'
    PORT = 80
    TIMEOUT-RCV = 1
    TIMEOUT_CON = 5
    POW32 = 2**32-1
    BUFFER = 1024
    if(type(size)==str):
        return []
    else:
        size = abs(int(size))
```

The code snippet below shows the socket object creation and the connection with the server. It is important to set a timeout so if the connection does not occur the code flow is not interrupted.

```
    s = socket.socket()
    s.settimeout(TIMEOUT_CON)
    try:
```

```
        s.connect((HOST, PORT))
    except:
        print('Timeout: cannot connect to server!')
        return []
```

The following command send to the server the number of random numbers that it must return to the client. Once the server will receive a string, the client ensures that the size sent is a string with each character corresponding to a byte.

```
    s.sendall(bytes(str(size),'utf-8'))
```

Thereafter the client is configured to receive the data inside a buffer with arbitrary size. The data received is in string format, so it is concatenated in one single variable and each number is separated by a line break since the server used a `println` function to write data into the client. Another timeout is used to check when there is no incoming data in the input buffer, so it can quit the while loop and proceed to close the connection with the server and post-process the received data into a list of number in the range of 0-1 with 32 bits resolution rather than a character string, ending the function.

```
    dataString = ""
    while True:
        dataString += s.recv(BUFFER).decode('utf-8'
    )
        ready = select.select([s], [], [], TIMEOUT)
        if not ready[0]:
            break
    s.close()
    data = list(map(int, dataString.split('\r\n')
    [0:-1]))
    return [x/POW32 for x in data]
```

It is important that inside the while loop, only commands related to receive and storing data are executed to preserve efficiency. If the data is early processed to numbers inside the loop, the communication can be delayed and others clients may experience troubles.

III. RESULTS

Three kinds of tests were made to evaluate the system performance. The first test was a elapsed time taken to generate a random sequence of arbitrary size; the second test was a series of randomness tests; and the third was a simply noise contamination of a sinusoid to check graphically the noise behaviour.

A. Elapsed time test

The results were extracted in Python simulations and were compared with the numpy PRNG function `numpy.random.rand()` that has a probability density function with uniform distribution.

The first test was composed of time duration for different quantities of random numbers to be generated. The table I show the results for this tests.

Due to the overhead of communication the TRNG server has a approximately constant time response until 10^3 and rise right after. PRNG is taking a little time to process, showing a best option regarding time consumption.

TABLE I

DURATION TO EXECUTE PYTHON BUILT-IN PRNG VS. ESP32 TRNG SERVER.

	Time duration (s)	
Quantity of numbers generated	TRNG duration (s)	PRNG duration (s)
10^0	3.0995	0.0
10^1	3.268	0.0
10^2	3.1463	0.00099
10^3	4.0195	0.0
10^4	6.2279	0.000993
10^5	7.3193	0.000989
10^6	13.8328	0.00598

B. Randomness tests

Then a series of randomness tests where used with the randomness test suite from stevenang that implement the NIST Test Suite for Random Number Generators [7] [**?**]. It was chosen 14 tests and they were replicated for both Python numpy PNRG and for the ESP32 TRNG, the results are shown in the figures 2 and 3.

Type of Test	P-Value	Conclusion
01. Frequency Test (Monobit)	0.03383773804326312	Random
02. Frequency Test within a Block	0.8699195144337334	Random
03. Run Test	0.3715888622647564	Random
04. Longest Run of Ones in a Block	0.37320962993202195	Random
05. Binary Matrix Rank Test	0.10010470503544915	Random
06. Discrete Fourier Transform (Spectral) Test	0.8905174796760388	Random
07. Non-Overlapping Template Matching Test	0.6387284559270616	Random
08. Overlapping Template Matching Test	0.13444360063455313	Random
09. Maurer's Universal Statistical test	0.7121000517258922	Random
10. Linear Complexity Test	0.9155630081826577	Random
11. Serial test:		
	0.387227041808463	Random
	0.5324470670556894	Random
12. Approximate Entropy Test	0.11791158743836992	Random
13. Cummulative Sums (Forward) Test	0.054767999343384464	Random
14. Cummulative Sums (Reverse) Test	0.009338739791689291	Non-Random

Fig. 2. NIST test suite applied to PNRG. Source: own.

Type of Test	P-Value	Conclusion
01. Frequency Test (Monobit)	0.8336676730351155	Random
02. Frequency Test within a Block	0.6936855664717952	Random
03. Run Test	0.649860553741109	Random
04. Longest Run of Ones in a Block	0.9432463010810312	Random
05. Binary Matrix Rank Test	0.3440726425983841	Random
06. Discrete Fourier Transform (Spectral) Test	0.91959576955207	Random
07. Non-Overlapping Template Matching Test	0.3296577468325298	Random
08. Overlapping Template Matching Test	0.3841228358397272	Random
09. Maurer's Universal Statistical test	0.4951462511803555	Random
10. Linear Complexity Test	0.05168639722695011	Random
11. Serial test:		
	0.5764511828472517	Random
	0.9352113998774924	Random
12. Approximate Entropy Test	0.17051069427578777	Random
13. Cummulative Sums (Forward) Test	0.8546298708096746	Random
14. Cummulative Sums (Reverse) Test	0.9204594255326393	Random

Fig. 3. NIST test suite applied to TNRG. Source: own.

The results clearly show that the TRNG sequence is in fact more random than the PRNG.

C. Time curves

The figure 4 show a good pattern of noise contamination for the TRNG sequence.

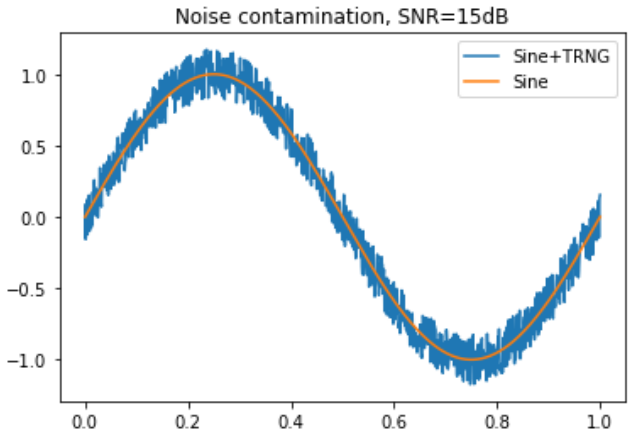

Fig. 4. TRNG noise contamination in a sine wave. Source: own.

Once that the random numbers are adjusted to the range $(0, 1]$, its values can be modified to meet any pattern of interest, including adjusting its amplitude to meet any signal-to-noise ratio requirement.

IV. CONCLUSIONS

The application have downsides that only one client can access the server at the time and the time taken to obtain the random vector may be substantial depending on the objective. But the results with the NIST test suite was very favorable, since it has shown that the TRNG sequence generated had a higher level of randomness than the PRNG sequence.

In this way this application should be used only in the strict case when true random numbers have high priority, since it take too long to give a proper response.

In future works the server could be improve to deal with multiplies client at the time, reducing the communication overhead between sections.

REFERENCES

[1] NIST. Random Bit Generation. Random Bit Generation Team, https://csrc.nist.gov/projects/random-bit-generation

[2] . Jason M. Rubin. Can a computer generate a truly random number?. MIT School of Engineering, https://engineering.mit.edu/engage/ask-an-engineer/can-a-computer-generate-a-truly-random-number/.

[3] Callegari, Sergio, Riccardo Rovatti, and Gianluca Setti. "Embeddable ADC-based true random number generator for cryptographic applications exploiting nonlinear signal processing and chaos." IEEE transactions on signal processing 53.2 (2005): 793-805.

[4] Wijesinghe, W. A. S., M. K. Jayananda, and D. U. J. Sonnadara. "Hardware implementation of random number generators." (2006).

[5] Espressif. Miscellaneous System APIs . https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/system.html

[6] Rukhin, Andrew, et al. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Booz-allen and hamilton inc mclean va, 2001.

[7] stevenang. randomness_testsuite. https://github.com/stevenang/randomness_testsuite