HOWTO: A Simple Random Number Generator for the ATmega1280 Microcontroller under C and TinyOS

Patrik Fimml Martin Perner Bernhard Petschina

May 21, 2015 (v2.0)

Contents

1 Introduction **1**
1.1 True randomness vs. pseudo-randomness . . . 1
1.2 Entropy . . . 2

2 Linear Feedback Shift Registers **2**
2.1 Software implementation . . . 3
2.2 Generating random values . . . 3
2.3 Reseeding . . . 3

3 Entropy source **3**

4 Module specification **4**
4.1 RNG module . . . 4
4.2 ADC module for C . . . 4
4.3 ADC module for TinyOS . . . 4
4.4 Concurrency . . . 5

5 Testing your implementation **5**
5.1 Compiling and linking . . . 5
5.2 Running the tests . . . 5

1 Introduction

Various tasks require some ability to make decisions at random, the most important ones probably being simulations using some statistical model, as well as a multitude of applications in the field of cryptography.

It is a common need to do cryptography in embedded systems. Typically, cell phone calls will be encrypted over the air, so a method to get random numbers is required here. (GSM encryption has been broken for some time now though.)

In the application task you need to implement some sort of random number generator to provide non-deterministic behavior to a part of the application's logic. This is not only required in games, but also in network protocols where are randomly chosen timeout is used for arbitration.

1.1 True randomness vs. pseudo-randomness

Usually it is difficult and resource-costly to get truly random numbers with the required rate of randomness. Instead, a *Pseudo-Random Number Generator (PRNG)* can sometimes be good enough. A PRNG provides a sequence of numbers that look as if they were random, as long as one does not know the internal state of the PRNG or the algorithm. As pseudo-random numbers share some statistical properties with truly random numbers, they can often act as a replacement.

After fetching a certain number of random values from a PRNG, the sequence will repeat itself. This is the so-called *sequence length* or *period*. PRNGs have a fixed amount of internal state that determines

their next output value. The state space gives an upper bound for the sequence length that can be achieved by an algorithm. Before the generator can be used, the internal state is initialized by providing a so-called *seed*. For the same seed, the same sequence of output values is generated.

Different applications have differing requirements for (pseudo-)random number generators, such as

- difficulty of predicting past and future values,
- uniform distribution of values,
- long sequence length,
- repeatability, and
- processing speed.

For example, for a simulation one might not care whether future values can be predicted from the history of generated values and knowledge of the algorithm. Instead the repeatability of the sequence is of major importance, i.e., given the seed, you want to be able to get the same result every time. This allows to repeat interesting scenarios under the same circumstances to find bugs, or compare modifications of the simulated system.

On the other hand, cryptographic applications have strong requirements on not being able to predict future or past values from a number of observations. To get such high-quality randomness, additional hardware resources or complex computations might be required.

1.2 Entropy

When discussing random numbers, *entropy* is an important concept. Entropy, or *Shannon entropy* to be precise, can be described as a "measure of randomness" [1]. Low entropy means that values can easily be predicted, while truly random data possesses high entropy.

2 Linear Feedback Shift Registers

A *Linear Feedback Shift Register (LFSR)* provides a simple form of hashing that can be easily implemented in hardware as well as in software. An ideal cryptographic hash function would map input data to an uniformly distributed output value of fixed length. For our use case, LFSRs provide a sufficient approximation of this behaviour. One popular example of real-world LFSR implementation are CRC checksums, used in IP and various other protocols.

When security is a concern, better alternatives are available, as various attacks are possible on LFSRs. Reseeding (see Section 2.3) can be used to inject truly random data into the LFSR and raise the bar for attackers.

There are two different ways of implementing LFSRs. We will only discuss the so-called *Galois configuration*, an example of which is depicted in Figure 1.

In each iteration, one bit of input is consumed and one bit of output is produced. The output bit feeds back to specific positions in the shift register, so-called *taps*. Each of these taps corresponds to a term in the characteristic polynomial $P(x)$. The choice of the taps and hence the polynomial is important, as only some combinations produce a maximum-length sequence. Other polynomials might yield sequences that repeat themselves very quickly and are thus unsuitable for random number generation.

A compilation of maximum-length polynomials can be found online [4].

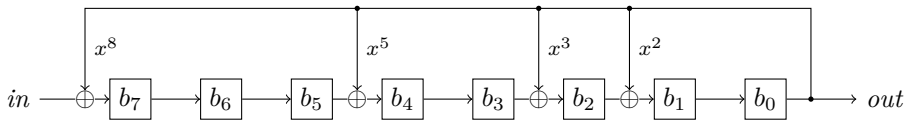

Figure 1. 8-bit LFSR in Galois configuration. b_0 through b_7 can be thought of as flip-flops sharing a common clock line, thus forming an 8-bit shift register. At XOR nodes ($\oplus$) or *taps*, the output feeds back into the register, corresponding to the terms of the characteristic polynomial $P(x) = x^8 + x^5 + x^3 + x^2 + 1$.

2.1 Software implementation

While software CRC implementations usually use lookup tables to process multiple bits at once, we will use a simple bit-shifting implementation.

Consider the pseudocode in Listing 1 and convince yourself that this is equivalent to the diagram in Figure 1. The SHIFT function takes one bit of input, shifts it in from the left and returns the bit shifted out to the right.

Note that the shift register needs to be initialized to a non-zero value.

```
poly ← 96_16
lfsr ← 1
function SHIFT(in)
    out ← LSB of lfsr
    right-shift lsfr
    MSB of lfsr ← in
    if out then
        lfsr ← lfsr ⊕ poly
    end if
    return out
end function
```

Listing 1. Pseudocode for a software implementation of the LFSR in Figure 1.

2.2 Generating random values

When calling SHIFT with an argument of 0, we get one pseudo-random bit as return value. To obtain a random 16-bit integer, we do this repeatedly to obtain more bits.

To obtain an integer in the range $[0; a)$, we generate a 16-bit integer and take the remainder of the division by a. As long as $a \ll 2^{16}$, this gives an approximately uniform distribution.

2.3 Reseeding

Until now, the behaviour of our random number generator is fully deterministic. To get actually random behaviour, we need a way to collect and mix in *entropy*. The process of changing the internal state of the PRNG using an outside random source is called *reseeding*.

Given that we have obtained one bit that is truly random, we call SHIFT to shift this bit into the LFSR. If our only objective is to reseed the PRNG, the return value can be ignored.

One method to obtain truly random bits is discussed in Section 3.

3 Entropy source

While the LFSR alone can give us *pseudo-random* data, we actually want to get truly random data whenever possible. In our scenario, the LFSR serves as an *entropy pool*: we collect actual random data and use it to reseed the LFSR. Simply put, as long as we feed more entropy into the LFSR than we take out of it, we will actually get random results. When we take more random data out than we put in or than the LFSR can hold, the "random" numbers will become predictable. This means our truly random numbers gracefully degrade to pseudo-random numbers if not enough entropy is available.

In cryptographic applications, one wants to avoid this situation. For example, the Linux kernel provides high-quality random data through the device `/dev/random`. To do so, it gathers random data from a variety of sources (e.g., timing information from mouse and keyboard interactions), estimating how much entropy the information contains ("how random it is"), and placing it into the entropy pool of `/dev/random`. When an application requests more entropy than was previously collected, the kernel will block until it gathers more random data.

For our application, we use a very simple method to obtain *somewhat* random data: we use the analog-to-digital converter (ADC) of the microcontroller to sample floating pins, thus essentially measuring noise.

4 Module specification

4.1 RNG module

Implement a 16-bit LFSR with the polynomial $80E3_{16}$ and the initial value 1. Use inline assembler[2] to implement rand_shift similar to Listing 1.

```
/**
 * Shift the LFSR to the right, shifting in the LSB of the parameter. Usually
 * not called directly; use the high-level functions below.
 *
 * Returns: The bit shifted out of the LFSR.
 */
uint8_t rand_shift(uint8_t in);

/**
 * Feed one bit of random data to the LFSR (reseeding).
 */
void rand_feed(uint8_t in);

/**
 * Get one bit of random data from the LFSR.
 */
uint8_t rand1();

/**
 * Generate a random 16-bit number.
 */
uint16_t rand16();
```

Listing 2. RNG module interface (`rand.h`).

4.2 ADC module for C

Use differential mode with 200× amplification to sample the voltage difference between pins ADC2 and ADC3 (both of which are left floating) with the highest available prescaler. Feed the LSB of the conversion result to the LFSR as described in Section 2.3. Do this periodically, at least once every 10 ms.

Note that you also need to sample the ADC0 pin for volume control, so you will need to work out a scheme to share the ADC hardware block between these two usages.

While a lower prescaler could be used to gather random data faster, this will also reduce the randomness of the data produced (i.e., 1's and 0's will often be seen in long runs).

The amount of entropy that can be obtained with this method has not yet been evaluated by us. However, the LFSR stage compensates for a potentially low-quality entropy source, and in the worst case (e.g., all 0's from the ADC) degrades to a pure PRNG.

4.3 ADC module for TinyOS

Similar to the above design for C applications, we use differential mode with 200× amplification to sample the voltage difference between pins ADC8 and ADC9 (both of which are left floating) with the highest

available prescaler. Feed the LSB of the conversion result to the LFSR as described in Section 2.3. Do this periodically, at least once every 10 ms.

Note that other modules will also have access to the ADC, so you will need to verify that the access to the ADC happens in a mutual exclusive way. TinyOS will take care of that automatically, if you declare the component `AdcReadClientC()` in the correct fashion.

We do not provide a separate test facility for TinyOS and therefore refer you to the provided test suite for C to develop the module. Afterwards you can just copy and paste the implementation of function, given in Listing 1, into your TinyOS module.

4.4 Concurrency

Make sure that your implementation *rand_shift* can be called from interrupt context without any weird effects, because you probably want to do so when implementing the entropy source. Consider what could go wrong if you do not take special precautions and what precautions should be taken. In a C-program `ATOMIC_BLOCK` may be useful, but of course there are multiple ways to implement this properly. Remember that TinyOS already has the keyword `atomic` to declare an atomic region.

5 Testing your implementation

To help you get started quickly, we provide a library that you can link against your implementation of the `rand` module. It will perform some basic tests on your code and display the results on the LCD. Only the pure PRNG functionality is tested, so you will have to test the reseeding part yourself.

Note that the intent is to help you find simple mistakes, not to exhaustively test your code. Your implementation might have bugs despite passing the test suite.

5.1 Compiling and linking

Link your `rand` module against `librand_test`, which contains testing code including a main function. You should not use any hardware blocks such as the ADC, this belongs in a different module.

Simply copying your implementation into the `librand_test/` directory and using the provided Makefile should work in most cases.

5.2 Running the tests

When executing the resulting program, you will see the test result on the LCD. Status codes are described in Table 1.

Code	Meaning
00	All tests passed. Your code may still have bugs.
01	Your code works, albeit slower than the reference implementation. Maybe you can find a way to make it faster? (You don't have to.)
10	*rand_shift* does not generate the expected output sequence.
11	*rand_shift* appears to always return the same result.
12	The sequence generated by *rand_shift* repeats too quickly.
13	*rand_shift* appears to ignore the input bit.
14	*rand_shift* should handle "special" input values sensibly.
15	*rand_shift* returns something other than 0 or 1.
21	Your code seems to have concurrency bugs.
22	Your code leaves interrupts enabled when they should not be.
23	Your code leaves interrupts disabled when they should not be.
24	Your code is too slow for the concurrency test mechanism, so the test suite went for a coffee instead.
30	*rand_feed*, *rand1* and/or *rand16* do not behave as expected.

Table 1. Status codes printed by the test library and their meaning.

References

[1] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. Wiley, 2010.

[2] Harald Kipp. *AVR-GCC Inline Asssembler Cookbook*. In: *AVR-libc User Manual*. URL: http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/atmega1280/avr-libc-manual/view.

[3] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. 3rd edititon. Addison-Wesley, 1997.

[4] Philip Koopman. *Maximal Length LFSR Feedback Terms*. URL: http://www.ece.cmu.edu/~koopman/lfsr/.