Ćwiczenie 18: Generator pseudolosowy i wskaźniki na funkcje

Instrukcja laboratorium

Mariusz Chilmon <mariusz.chilmon@ctm.gdynia.pl>

2024-06-04

> Code never lies, comments sometimes do.
>
> — *Ron Jeffries*

Cel ćwiczenia

Celem ćwiczenia jest zapoznanie się z:

- sposobem działania generatora pseudolosowego,
- metodami pozyskiwania entropii w systemach mikroprocesorowych,
- wykorzystaniem implementacji podstawowych algorytmów z biblioteki standardowej,
- wykorzystaniem wskaźników na funkcje.

Uruchomienie programu wyjściowego

1. Podłącz płytkę *LCD Keypad Shield* do *Arduino Uno*.
2. W lewej części wyświetlacza widocznych jest 6 liczb wybranych przez generator pseudolosowy, symulujący losowanie Lotto.
3. Po kliknięciu przycisku *RIGHT* losowane są nowe liczby.
4. W prawej części wyświetlacza widoczne jest ziarno losowości.

Rysunek 1: Wyjściowy stan wyświetlacza

Program wyjściowy symuluje losowanie liczb w grze Lotto — wybiera w funkcji `Lotto::shuffle()` 6 niepowtarzających się liczb z przedziału [1; 49], dokonując za pomocą algorytmu Fishera-Yatesa permutacji tablicy `numbersPool`, wypełnionej wstępnie kolejnymi liczbami naturalnymi od 1 do 49. Na koniec wybieranych jest 6 pierwszych liczb z tak przemieszanej tablicy.

Algorytm Fishera-Yatesa nie zapewnia losowania sam w sobie, ale wymaga użycia zewnętrznej funkcji losującej. W języku C mamy do dyspozycji funkcję `rand()`, która zwraca wartości z zakresu [0; `RAND_MAX`], gdzie w naszym przypadku `RAND_MAX` wynosi 32767.

Język C++ od wersji C++11 posiada bibliotekę `random`, która zawiera bogaty zestaw narzędzi do generowania wartości pseudolosowych zgodnych z różnymi rozkładami prawdopodobieństwa i z użyciem różnych implementacji generatorów. Ze względu na złożoność nie jest ona dostępna dla platformy AVR.

Należy jednak pamiętać, że żaden generator pseudolosowy, czyli PRNG (*Pseudorandom Number Generator*), nie generuje liczb losowych! Mikroprocesor jest urządzeniem deterministycznym zaprojektowanym z założeniem, że lepiej, by nie działał w ogóle niż działał w sposób nieprzewidywalny[1]. Nie istnieje zatem algorytm, który byłby w stanie wygenerować prawdziwie losowe liczby[2]. Algorytmy PRNG zaprojektowane są w sposób, który zapewnia odpowiedni rozkład generowanych liczb, przypominający losowy, ale wiele z nich jest do tego stopnia deterministycznych, że po zebraniu kilku wygenerowanych liczb można przewidzieć następne. Stąd po każdym włączeniu urządzenia, widoczne są te same wyniki — algorytm zaczyna pracę od tej samej wartości.

Zadanie podstawowe

Celem zadania podstawowego jest inicjalizacja PRNG ziarnem, czyli wartością o charakterze losowym, a przynajmniej bardzo trudnym do przewidzenia i mało powtarzalnym. Idealnym źródłem byłyby zjawiska kwantowe, które z założenia są czysto losowe, co sprawdzono w wielu złożonych eksperymentach, wykazując brak korelacji z jakimkolwiek czynnikiem, włączając w to sygnały sprzed 10 miliardów lat[3]. Przykładami źródeł o takim charakterze są szumy śrutowe złącza p-n czy rozpad promieniotwórczy.

Na nasze potrzeby w zupełności wystarczające jest wykorzystanie jakiegoś procesu o trudnym do przewidzenia przebiegu, np. szumu termicznego, parametrów pogodowych, zjawisk chaotycznych czy interakcji z użytkownikiem. Tym ostatnim rozwiązaniem mógłby być precyzyjny pomiar czasu wciśnięcia przycisku przez użytkownika, ale zakładamy, że chcemy, by urządzenie wyświetlało losowe liczby bez żadnej interakcji. W tym celu użyjemy termometru. Wprawdzie temperatura otoczenia może być bardzo stabilna, jednak na pomiar wykonany z odpowiednią rozdzielczością nałoży się szum termometru i ADC. Analizując więc nie tylko samą temperaturę, ale biorąc pod uwagę jej chaotyczne oscylacje, możemy uzyskać wystarczająco dobre źródło losowości (entropii).

Mikrokontroler ATmega328P wyposażony jest we wbudowany termometr, dzięki czemu nie ma potrzeby uzupełniania urządzenia o dodatkowy sensor.

Wymagania funkcjonalne

1. Po każdym uruchomieniu urządzenie stosuje inne ziarno losowości i wyświetla inne liczby.
2. Użyte ziarno jest wypisywane na wyświetlacz.

[1] *Vide* BOR (*Brown-out Detector*), czyli obwód wyłączający mikrokontroler przy zbyt niskim napięciu zasilania, kiedy część procesora może jeszcze pracować, ale błędnie.

[2] „Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin" — John von Neumann.

[3] Ewentualnie wyniki eksperymentów zostały z góry ustalone u zarania dziejów Wszechświata, ale dla nas jest to nieodróżnialne.

Modyfikacja programu

Konfiguracja ADC

Odczytaj z dokumentacji mikrokontrolera numer kanału ADC oraz wymagane napięcie referencyjne dla termometru wewnętrznego i skonfiguruj odpowiednio rejestr `ADMUX` w metodzie `Lotto::initialize()`.

Potrzebne informacje znajdziesz w sekcji *Temperature Measurement* dokumentacji mikrokontrolera.

Nieprawidłowa konfiguracja ADC niekoniecznie uniemożliwi działanie algorytmu zbierania entropii (bo nieużywane kanały ADC mogą również generować szum), ale proces zbierania entropii może być dużo dłuższy.

Skonfiguruj preskaler ADC i włącz przetwornik:

```
ADCSRA = _BV(ADEN) | _BV(ADPS2) | _BV(ADPS1) | _BV(ADPS0);
```

Teraz możesz uruchomić pomiar, ustawiając bit `ADSC` (*ADC Start Conversion*) i odczytać wynik z pary rejestrów `ADC` po wyczyszczeniu tego bitu przez przetwornik. Ze względu na zmianę napięcia referencyjnego oraz kondensator podłączony do pinu *AREF* pierwszy pomiar będzie znacząco zaniżony. Należy odczekać kilka milisekund i wykonać drugi pomiar:

```
ADCSRA |= _BV(ADSC);
while (bit_is_set(ADCSRA, ADSC)) ;
_delay_ms(10);
ADCSRA |= _BV(ADSC);
while (bit_is_set(ADCSRA, ADSC)) ;
uint16_t seed = ADC;
```

Odczytaną wartość można wyświetlić na wyświetlaczu i porównać z dokumentacją celem oceny poprawności konfiguracji ADC.

Algorytm zbierania entropii

Zaimplementuj algorytm zbierania entropii według poniższego wzoru. Sumujemy tutaj pomiary z ADC tak długo, aż zarejestrujemy 256 zmian wartości temperatury.

Algorithm 1 Algorytm zbierania entropii

$liczbaZmian \leftarrow 0$
$poprzedniADC \leftarrow 0$
repeat
  wykonaj pomiar ADC
  $seed \leftarrow seed + ADC$
  if $ADC \neq poprzedniADC$ **then**
    $poprzedniADC \leftarrow ADC$
    $liczbaZmian \leftarrow liczbaZmian + 1$
  end if
until $liczbaZmian < 256$

Inicjalizacja PRNG

Tak obliczonym ziarnem zainicjalizuj PRNG używając funkcji `srand()`. Wyświetl również wartość ziarna na wyświetlaczu LCD.

Zadanie rozszerzone

Celem zadania rozszerzonego jest sortowanie wylosowanych liczb za pomocą algorytmu z biblioteki standardowej.

Wymagania funkcjonalne

1. Losowane liczby wyświetlane są w kolejności rosnącej.

Modyfikacja programu

Użyj algorytmu `qsort()` z biblioteki standardowej. Znajdź w Internecie dokumentację tej funkcji i użyj jej na tablicy `result.buffer` w metodzie `Lotto::shuffle()` przed zwróceniem danych. Zwróć uwagę, że ostatnim argumentem funkcji `qsort()` jest wskaźnik na funkcję, która odpowiada za porównywanie danych.

Sygnatura tej funkcji powinna przyjmować wskaźnik typu `const void*` (wskaźnik na dowolną wartość), ponieważ `qsort()` może sortować dane różnego rodzaju. Warto na jej początku dokonać konwersji tych wskaźników na `const uint8_t*` i odczytać wartości do zmiennych, na których będzie można prosto operować:

```
int compare(const void* aPointer, const void* bPointer)
{
    uint8_t a = *reinterpret_cast<const uint8_t*>(aPointer);
    uint8_t b = *reinterpret_cast<const uint8_t*>(bPointer);

    // Compare and return.
}
```