Pseudo-Random Number Generation Routine for the MAX765x Microprocessor

This application note gives a function for random number generation using the MAX7651/52 microcontroller with 12-bit analog-to-digital converter (ADC).

Applications such as spread-spectrum communications, security, encryption and modems require the generation of random numbers. The most common way to implement a random number generator is a Linear Feedback Shift Register (LFSR). Codes generated by a LFSR are actually "pseudo" random, because after some time the numbers repeat. The trick is to use a shift register of sufficient length so that the pattern repeats after some extremely long time.

A basic LFSR of length five is shown in Figure 1. The shift register is a series-connected group of flip-flops, with XOR feedback. An XOR gate is use to scramble the input bit.

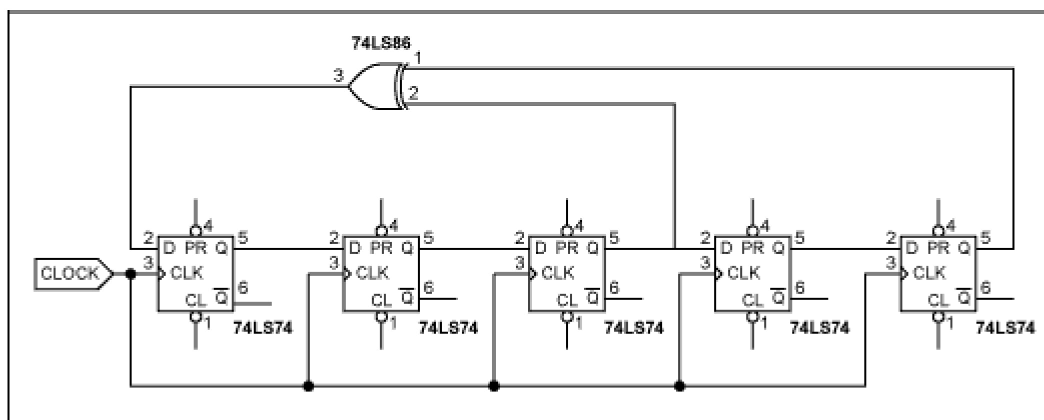

Figure 1. 5-stage linear feedback shift register

There are tables that give the proper feedback tap position for generating sequences that take the maximum number of clocks to repeat. Such a table is shown below:

Table 1. Taps for Maximal-Length LFSRs with 2 to 32 Bits

No. of Bits	Length of Loop	Taps
2	3*	[0,1]
3	7*	[0,2]
4	15	[0,3]
5	31*	[1,4]
6	63	[0,5]
7	127*	[0,6]
8	255	[1,2,3,7]
9	511	[3,8]
10	1023	[2,9]
11	2047	[1,10]
12	4095	[0,3,5,11]
13	8191*	[0,2,3,12]
14	16,383	[0,2,4,13]
15	32,767	[0,14]
16	65,535	[1,2,4,15]
17	131,071*	[2,16]
18	262,143	[6,17]
19	524,287*	[0,1,4,18]
20	1,048,575	[2,19]
21	2,097,151	[1,20]
22	4,194,303	[0,21]
23	8,388,607	[4,22]
24	16,777,215	[0,2,3,23]
25	33,554,431	[7,25]
26	67,108,863	[0,1,5,25]
27	134,217,727	[0,1,4,26]
28	268,435,455	[2,27]
29	536,870,911	[1,28]
30	1,073,741,823	[0,3,5,29]
31	2,147,483,647*	[2,30]
32	4,294,967,295	[1,5,6,31]

* Sequences whose length is a prime number

Be aware that there are multiple solutions for tap positions to generate maximum length sequences.

There is one major issue with using LFSRs: if all stages happen to be '0', the shift register gets "stuck". This is because the XOR of all '0's is still '0'. The XOR feedback does not produce a '1' to get the sequence started again. To prevent this condition, the routine **must** be first loaded with a non-zero seed value. This value can be any number at all, as long as it is not zero. The numbers generated by the LFSR are based on the seed value. You will get the same sequence of numbers, over and over, unless at some point the LSFR is reloaded with a different seed.

Where does this seed value come from? It will depend on what is available in your particular application. For example, if your system has access to a RTC (real-time clock), then a good seed is based off the time. You can read the current time and/or date, mask off portions and use that as a seed. Another example is temperature. If your system can read temperature (assuming it's not constant) then that can make a good seed. The ADC of the MAX765x can be set to read all sorts of things: scales AC power line voltage, some sensor position or even amplified Johnson noise from a Zener diode (a common practice in cryptography).

However, in some cases you will just have to use 01H or some other number, and accept the fact that the sequence will repeat eventually and in a predetermined pattern.

The routine presented uses a 25-bit sequence, which repeats after being called 33 million times. Even if you cannot produce a unique seed each time, the length is such that in most applications, the "randomness" is more than sufficient.

The MAX765x listing is shown below. The routine uses four 8-bit memory locations labeled RN1-RN4. The lower 3 bytes RN1-RN3 are used for 24 bits, and the MSB of RN4 is the 25th bit. The algorithm uses XOR feedback (using the processor's XRL instruction) from "stages" 25 (the Carry bit) and stage 7 (the MSB of RN1). Since all of the resisters are simply RAM locations, you can form up to 32-bit wide random numbers. For this example, an 8-bit number RANNUM is stored in RAM at the end of the routine.

To get a true Gaussian distribution function of the random numbers, you can do further processing. Adding some arbitrary number of consecutive samples and taking the average (say 4) will create a Gaussian distribution.

One programming 'trick' which is used in the algorithm are "byte swaps" to simulate a "shifting by 8 clocks". This is to save CPU clock cycles. For example, if the original byte order was ABCD, after the byte-swaps the order is BCDA. This prevents the code from having to do "housekeeping" from shifting a byte's MSB into the next byte's LSB. It doesn't matter if the random numbers are calculated every clock or every 8 clocks: they are still random. Since the total length of the LFSR is a product of 3 prime numbers (31, 601 and 1801) the sequence is still 33,554,431 subroutine calls until the sequence repeats! Of course, since we are looking at 8 of the bits for our example, the values are restricted to 00H to 0FFH, and so the same value will be returned multiple times.

Another cycle-saving 'trick' used is when the XOR of the 2 bits of interest is executed, the entire contents of RN1 is altered (see the comment below).

```
; SUBROUTINE RANDOM
;
; GENERATES RANDOM NUMBERS USING 25 BIT SHIFT REGISTER
; WITH XOR FEEDBACK ON STAGES 7 AND 25 (MAX CYCLE LENGTH)
;
; IN ORDER FOR THIS ROUTINE TO WORK, THERE ARE 2 THINGS TO CONSIDER:
; A) SOME TYPE OF NON-ZERO VALUE 'SEED' MUST FIRST BE LOADED INTO RN1
; B) THE SEQUENCE WILL REPEAT AFTER APPROX 33 MILLION CALLS TO THE
ROUTINE
;
; CPU RESOURCES REQUIRED:
;
; 6 RAM LOCATIONS AS FOLLOWS
;
; RANNUM: THE CALCULATED RANDOM NUMBER
; RN1-RN1: FORMS THE SHIFT REGISTER
; TEMP1: A TEMPORARY SCRATCH-PAD REGISTER
;
; THE ROUTINE DESTROYS THE A REGISTER AND THE CARRY BIT
;
;
```

```
RANDOM: MOV  A,RN4       ;BIT 25 TO CARRY
        RRC  A
        MOV  RN4,RN3
        MOV  RN3,RN2
        MOV  RN2,RN1     ; SHIFT ALL BYTES LEFT
        MOV  A,RN4       ; WAS RN3
        RRC  A           ; GET BIT 25 (IN CARRY) INTO OLD RN3
        MOV  TEMP1,A     ; SAVE IT FOR LATER
        MOV  A,RN1
        RLC  A           ; BIT 1 TO 7 IN POSITION
        MOV  RN1,A       ; PUT BACK
        MOV  A,TEMP1     ; RESTORE A
        XRL  A,RN1       ; MOD 2 SUM OF ALL BITS IN RN1 AND THE
                         CARRY BIT
        MOV  TEMP1,A     ; SAVE IT
        MOV  A,RN4
        RRC  A
        MOV  RN4,A       ; ROTATE RN4
        MOV  A,RN3
        RRC  A
        MOV  RN3,A       ; ROTATE RN3
        MOV  A,RN2
        RRC  A
        MOV  RN2,A       ; ROTATE RN2
        MOV  A,TEMP1     ; RESTORE A
        RLC  A
        MOV  RN1,A       ; xor OVER-WRITES ALL 8 BITS, BUT DOESN'T
                         MATTER
        MOV  RANNUM,A    ; RANDOM NUMBER!!
        RET
```

MORE INFORMATION

MAX7651: QuickView -- Full (PDF) Data Sheet (488k) -- Free Sample

MAX7652: QuickView -- Full (PDF) Data Sheet (488k) -- Free Sample