Random Number Generator for Microcontrollers

by
Tom Dickens
tom@tomdickens.com

Abstract

The use of random numbers in small microcontrollers can be very useful and fun in robotic programming to yield different behaviors from the same robotic-control program. Due to limited memory, the lack of floating-point math capabilities, and the limited size of integers in some microcontrollers, the practical implementation of a software-based random number generator for a microcontroller is difficult. This paper looks at the generation of random numbers, and compares a variety of implementations on the 68HC11 microcontroller. A good implementation is chosen and the 68HC11 source code is provided that implements it.

Keywords

- Random number generator
- Microcontroller
- 68HC11

Introduction

The use of random numbers in robotics projects (and other microcontroller projects) can be very beneficial. I've seen cases where a wall-following robot gets stuck in a corner at just the right angle. This is due to the algorithm hard-cored into its brain producing the exact same back-up, turn-a-bit, go-forward motion that just oscillates back-and-forth in the corner (what a disgrace!). While those of us who know actually what is going on inside of the robot can understand this behavior, to the general public this makes the robot appear to be quite brainless and dim-witted; the silly thing can't even find its way out of a corner. If the program in the robot used random numbers to slightly alter the back-up time and also the degrees turned in the same algorithm, this robot would not have the exact same back-and-forth behavior that caused it to be stuck in the corner. It would very quickly maneuver out of the corner and continue on its way, making itself (and you) look quite intelligent. The availability of random numbers is also critical for many algorithms incorporating artificial intelligence (AI) techniques such as genetic algorithms and genetic programming, neural networks, and fuzzy logic.

The topic of random number generation (RNG) can be divided into two categories: true random number generation and pseudo random number generation. With true random number generation the next random number generated is not known, and the sequence of random numbers cannot be re-generated. With pseudo random number generation, a sequence of "random numbers" is generated using a known algorithm, and the exact same sequence can be re-generated; hence the classification of pseudo. For software-based systems it is very desirable to use a pseudo random number generator to be able to test the software system with a repeatable set of random numbers; you can re-run a test with

the exact same random numbers being used. Also, with a pseudo random number generator that has been formally evaluated, you can be sure of the attributes of the resulting sequence of random numbers.

Random number generators provided with workstation-class computers return a 32-bit or a 64-bit integer number, or a floating-point number based on that value. In small microcontrollers the desired value from a random number generator is generally an 8-bit or 16-bit number. I will focus my discussion on an 8-bit random number generator.

In my random number generator study I first defined a set of grading criteria as attributes I wanted in a RNG. I then wrote a Java program on a PC to test various RNG algorithms against these criteria, and wrote different algorithms in this program to generate sequences of random numbers to test. I then used the knowledge gained from the successful algorithms and data to write 68HC11-specific assembly language to implement the RNG. Lastly I ran the algorithm on the 68HC11 and verified that it generated the same sequence seen in the Java code.

Hardware Methods

A true random number generator can be implemented in hardware using a white-noise[1] source. In [2] and [3] a hardware circuit is detailed which is used to generate random numbers using a pair of transistors. In [4] another hardware-based random number generator is discussed. While quite useful in many robotic applications, a hardware-based random number generator is out of the scope of the focus of this paper.

Requirements of a “Good” Random Number Generator

Two famous quotes on random numbers help us to set the stage for our discussion:

> ***“Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.”*** *John von Neumann (1903-1957).*

> ***“The generation of random numbers is too important to be left to chance.”*** *Robert R. Coveyou, Oak Ridge National Laboratory in Tennessee (Found in Ivars Peterson, 'The Jungles of Randomness' pp 178).*

In the “art” of generating random numbers with a software-based algorithm, you must determine the attributes you want in a random number generator, and then test candidate random number generators against these grading criteria. It is difficult to define a set of attributes that are simple to define, easily testable, and complete enough to avoid sequences of numbers that follow a simple pattern, such as a simple sequence (0, 1, 2, 3, 4, 5…) or an increment-by-N sequence (0, 21, 42, 63, 84…). Table 1 shows the list of attributes I used in testing candidate random number generators targeted for the 68HC11 microcontroller.

Table 1. Attributes for grading a RNG sequence.

Attribute	Parameters	Rational
Number range	Numbers from 0 to 255 are generated.	An 8-bit processor typically works with 8-bit integers.*
Number range average	The average of this number range is 127.5.	Definition used in subsequent attributes. (255-0)/2
Cycle length	The length of the pseudo random sequence shall be at least 16,384 (2^{14}) to 32,768 (2^{15}), but preferably will be 65,536 (2^{16}).	The longer the RNG cycle is the more varied the resulting behavior is.
Number coverage	All numbers in the range will be included the same number of times in the cycle. For example, for a cycle length of 65,536, all numbers 0 through 255 will occur 256 times.	Through the cycle I want all numbers to occur an equal number of times so the average of the cycle is the average of the range, and to ensure all possible values can be encountered.
Large window average	The average of 128 sequential values in all selections from the cycle will vary from the average by at most 20% (i.e. 102 to 153).	To make sure that any sequence of this size will cover a good range of values.
Large window min/max	The average of 128 sequential values in at least 1 selection will vary from the average by at least 10% above and by another selection by at least 10% below (i.e. 114 to 140).	To make sure the window averages are not all the same or too similar.
Medium window average	The average of 32 sequential values in all selections from the cycle will vary from the average by at most 40% (i.e. 75 to 179).	To make sure that any sequence of this size will cover a good range of values.
Medium window min/max	The average of 32 sequential values in at least 1 selection will vary from the average by at least 20% above and by another selection by at least 20% below (i.e. 102 to 153).	To make sure the window averages are not all the same or too similar.
Small window average	The average of 8 sequential values in all selections from the cycle will vary from the average by at most 75% (i.e. 31 to 223).	To make sure that any sequence of this size will cover a good range of values.

* A 16-bit number can be generated using two successive 8-bit values. We could add an attribute to consider the goodness of a sequence of 16-bit values generated this way, but I decided against this.

Attribute	Parameters	Rational
Small window min/max	The average of 8 sequential values in at least 1 selection will vary from the average by at least 35% above and by another selection by at least 35% below (i.e. 79 to 175).	To make sure the window averages are not all the same or too similar.
Repeated values	A specific value may or may not be repeated in the cycle.	I want to discuss this, but do not plan to specify a requirement or test for this.
Repeated sequence	There is not any sequence of 4 or more numbers that is repeated in the cycle.	3 or more repeated numbers may be useful/interesting, but I don't want more than 3.
Value-to-value delta	A sequence made from the differences in the sequence values will meet the same window average and repeated criteria.	To make sure there are no patterns in the deltas.
Operations allowed	The operations used to implement the random number generator are 8-bit add, subtract, shift, and, or, exor, and 8-bit to 16-bit multiply.	These are the commonly offered microcontroller operations, and are found in the 68HC11.
Size of code	The size of the code to implement the algorithm is to be as small as possible-at most 100 bytes long.	Smaller is better with the limited program space available in microcontrollers.
Required RAM	The number of bytes of RAM required to be dedicated to the RNG should be as few as possible and at most 8 bytes.	Smaller is better with the limited RAM space available in microcontrollers.

Approaches

I investigated two different approaches to implementing a simple random number generator for the 68HC11; a table-based method and an algorithm-based method.

Table-based method

For a table of 256 values, there are 256!, or 8.578e+506, possible ways to order the table. My table-based approach uses a table of 128 values, with an algorithm to traverse the table in a number of ways to generate a random number cycle 16,384 numbers in length. I used a table of 128 values and generated the other 128 values in the set of {0 – 255} by EXORing the 128 table values with $FF. An acceptable set of 128 values for this table include the criteria that EXORing these 128 values will generate the remaining 128 values in the set. This gave me an effective table of 256 values using only 128 bytes, plus a small number of bytes for the implementation logic. I then traverse this set of 256 values in 64 different ways to generate a sequence of 16,384 values. Once the driving logic of traversing the table was developed, a good set of table values as seen in Table 2

was determined using a program to try different sets of values* and testing the resulting table against the specified grading criteria. I wrote a Java program to do this.

Table 2. 128 values used in my table-based RNG.

```
247, 86,108,103,252, 35,115, 75,202, 70,107, 89, 37, 40,246, 69
111,234, 56, 12,249,146, 19, 80,240,230, 45, 38,197,223, 65,123
 26,208, 74,167,130, 79,253,121,173, 93,136,198, 64,204, 90, 20
 30,233,222, 16, 14,242,182,174,195,105,159,  0, 84,129,250,155
 10, 28,251, 92, 62,201,192,142,104,168,114, 49,172,124, 39, 67
143,254,  7,228,116, 31,154,156, 72, 36,194,161, 71,226, 59, 18
209,177,118,189,221, 98,135,122, 48,153,117,231,238,164, 52,187
 44,145,170,213, 77, 97,128,212, 41,160, 11,149,200, 23, 50,179
```

As this was the first algorithm I got working, I was very attached to it. I then decided to investigate equation-based methods for completeness in my research, but I really liked my table-based solution and thought I would decide to go with it as the best solution.

Equation-based method

An equation-based method uses a seed value. For each random number generated this seed value is used in an algorithm to generate a new seed value and also the random number. In the case of our 8-bit random number generator we want the seed value to be larger than the 8-bit number returned since an 8-bit seed value will allow at most a cycle of 256 values. A classic algorithm for generating random numbers[5] is:

```
unsigned long seed ;
...
seed = 1664525L * seed + 1013904223L ;
```

I implement code using a 16-bit seed value, and looked at different 8-bit selections from the newly-calculated seed as the 8-bit random number to return.

Comparison of Approaches

The table-based algorithm generated a good set of random numbers. The cycle was 16,384 numbers long. The resulting memory-size was 4 bytes of RAM. For program memory it required 62 bytes, plus 128 bytes for the initial table, for a total of 190 bytes.

The equation-based algorithm also generated a good set of random numbers, and the cycle was a full 65,536 numbers. The resulting memory-size was also 4 bytes of RAM. For program memory it required only 25 bytes. The resulting equation was:

```
seed = 181 * seed + 359 ;
```

where seed is a 16-bit number. The returned 8-bit value is the top half of the seed.

* I used the random number generator in Java to populate the table, checking for the EXOR criteria, then tested the resulting table against all of the grading criteria. Running this code found many acceptable solutions for the table data; I choose one for my implementation.

Using a Java program on the PC, I searched for a good set of constants. The numbers 181 and 359 were used in the equation (both prime numbers), which generated a sequence of 65,536 values that satisfied the grading criteria. Based on the size of the code and the complexity of the algorithm, I have to choose the equation-based algorithm as the best random number generator code for my applications with the 68HC11 microcontroller.

Testing

The 65,536 sequence of numbers generated by the equation were tested against the criteria in Table 1. I list the first part of this sequence in Table 3 below. Notice that the number 172 is repeated, which informs us that a number can be repeated in the sequence.

Table 3. The first part of the 65,536 numbers generated by the equation.

```
  1 255 117   4  73 222 125 232  15 167  21 110 230 252  49  27
 35  65 133  50 218 156 132 185 223 239  99 114 197 223  22 226
226 208  81  76  71 229 135 182 203   4 237 226   1 207 119  85
 60 170 216  82 145 187 134 223 211 228 179 190 152 202  84 116
 50 209  28  68 182  28 128  52 248 145 181   6 139 210 172  63
196  68  28  34 183  10 119 181  56   9 242 186 219 229 129 182
243   2 216 237 149 132 106  98 148  78 108 218 134   5 210 217
189  13  80 163  78 137  89  59  13  94  34 102 141  48 159 168
 35  99 131  69 227  27  68  64 161  59  20  94 241 104 232  35
 38   6 115 211  85  56  43 113  81 228  66 194 176 172 172  74
196 244  31  77 162 226  13 206  30  88 171 146 203 251 237  29
254  47 135 179 203  23 236  87   6 153  81 206  66  87 170 156
213 181 171   5 209 217 199  12  10 165  51 118  22 190 227 199
 71 136 138  67 178  39 158 237  43 126  81 138  69  50 152 158
 85 167  38 109 111   0 113 250 103  34 171  11 208 177 201  33
...
```

As a follow-on study of the sequence of random numbers, I plotted them as pairs on a 256x256 grid. In Figure 1 left, three sets of 1024 pairs are plotted as red, blue, and black points. In Figure 1 left, half of the 32,768 pairs of numbers are plotted. Both of these plots visually look random, while the right figure is starting to look a bit grid-like.

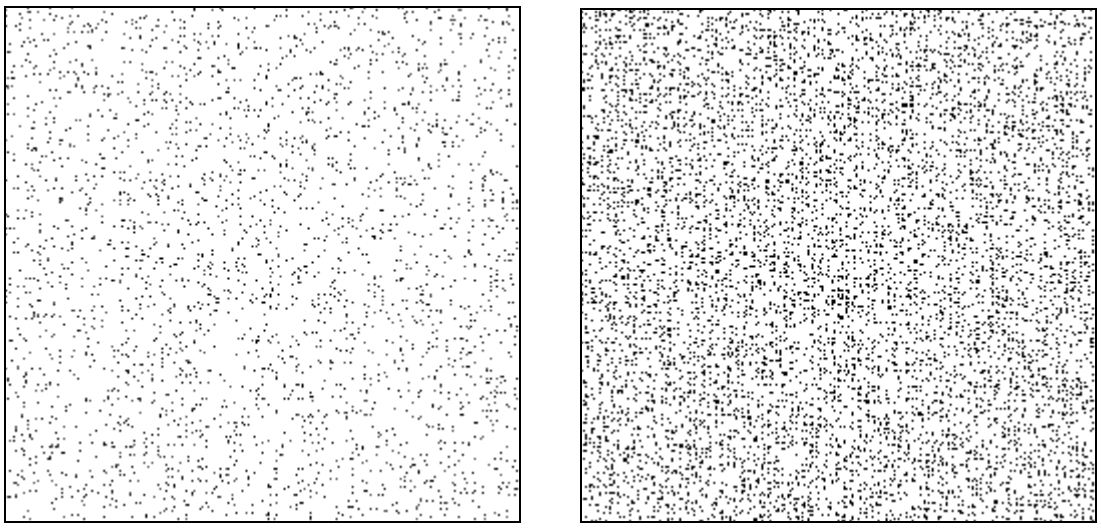

Figure 1. Three sets of 1024 pairs (left), and half of the pairs (right) plotted.

In Figure 2 below all 32,768 pairs of numbers in the 65,536 set of random numbers are plotted. This does produce a definite pattern, but it was the least pattern-like pattern I found in searching many pairs of candidate constants. Another key point of this set is that out of 32,768 pairs of numbers, there were 32,768 unique points plotted; there were no duplicate points from the pairs of numbers.

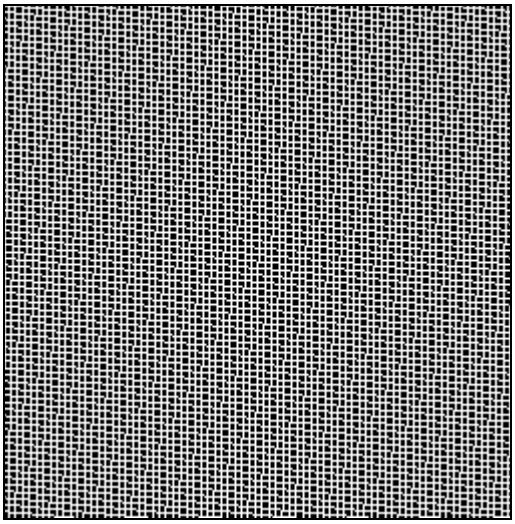

Figure 2. All 32,768 pairs plotted, resulting in a noticeable pattern.

Most other constants used for the equation covered far fewer positions when plotted, and generated a much more definite and simple pattern. For example, when using 33 and 171 for the constants, the plot as shown in Figure 3 only covers 9728 out of 32,768 positions.

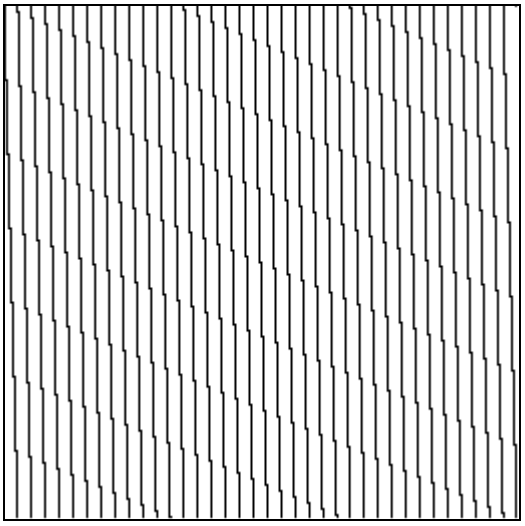

Figure 3. All 32,768 pairs plotted for a different set of constants, resulting in a very noticeable pattern.

After extensive testing and looking at both algorithms and different constant values, I settled on the equation-based with the constants 181 and 359. This is the algorithm implemented in the 68HC11 example code.

The Code

Below is 68HC11 assembly code implement the equation to generate random numbers. All lines starting with '*' are comments, and text following a ';' sign are also comments. This was assembled with the AS11 assembler[*] and loaded in a 68HC11E1 on a BOTBoard using PCBUG11 for testing.

```
*********************************************************************
* Author: Tom Dickens
* Date: 12/10/2002
* Purpose:
* 8-bit random number generator for the 68HC11/68HC12 microcontrollers.
* Method: A 16-bit version of the classic S = S * M + A is used.
* The random number returned is the high byte of the new seed value.
* The constants M and A were chosen to be M=181 and A=359 for
* the following reasons:
*   - They are both prime numbers.
*   - The equation cycles through all 65,536 possible seed values.
*   - The 8-bit random values have the following properties:
*       - Each number from 0 to 255 occurs 256 times in the cycle.
*       - No sequence of 4 or more numbers is repeated.
*       - Every 128-byte window has an average between 104 and 150 (+-20%)
*       - Every 32-byte window has an average between 76 and 178 (+-40)
*       - Every 8-byte window has an average between 32 and 218 (+-75%)
*   - Small RAM and Program space required:
*       - Only 4 bytes of RAM are used
*       - Only 25 bytes of Program memory is used
*   - Fast. Each call to Random takes 64 clock cycles, which
*           is 32 micro-seconds with an 8MHz crystal.
* Usage:
*    JSR Random     ; Register A contains the next random number.
*********************************************************************

*********************************************************************
* RAM Usage: 4 bytes in zero-page RAM, bytes 0 trough 3.
*********************************************************************
       ORG $0000

RandomSeed:
       RMB    2      ; reserve two bytes in RAM for the SEED value
SEED_HIGH     EQU    RandomSeed      ; high byte of the seed
SEED_LOW      EQU    RandomSeed+1  ; low byte of the seed

RandomScratch:
       RMB    2      ; reserve 2 bytes in RAM for scratch space

*********************************************************************
* Constants:
*********************************************************************
MULTIPLIER    EQU    181
ADDER         EQU    359

*********************************************************************
* Program start: Example program to display random numbers on Port C.
*    Change ORG $B600 to $F800 for 'E2 devices.
*********************************************************************
       ORG $B600
       LDAA   #$FF
       STAA   $1007  ; Set port C to output
```

[*] Version 2.1, dated 10-Aug-91, by Randy Sargent.

```
       LDAA   #$00
       STAA   $1003         ; Clears port C to $00
       BSR RandomInit       ; Seed = 0
TopLoop:
       BSR Random           ; A = random number
       STAA $1003           ; A -> Port C
       LDX #$FFFF
       BSR DelayX           ; wait a bit to see the number
       BRA TopLoop          ; do it again...

*************************************************************************
* Initialize the random number generator to the start of the sequence
* by loading the seed value with 0. You can initialize the seed to any
* value you choose, which will start the random number sequence somewhere
* else in the cycle of 65,536 seed values.
*************************************************************************
RandomInit:
       CLR    SEED_HIGH
       CLR    SEED_LOW
       RTS

*************************************************************************
* Return the next 8-bit pseudo random number in the sequence
* of 65,536 numbers, generated from the following equation:
* SEED = SEED * 181 + 359
* The random value returned is the high byte of the new SEED.
* Return value: The A register.
*                            (bytes, cycles) = (25,64)
*************************************************************************
Random:
       PSHB                       ; (1,3) Remember the current value of B
* scratch = seed * multiplier
       LDAA   #MULTIPLIER         ; (2,2) A = #181
       LDAB   SEED_LOW            ; (2,3) B = the low byte of the seed
       MUL                        ; (1,10) D = A x B
       STD    RandomScratch       ; (1,4) scratch = D
       LDAA   #MULTIPLIER         ; (2,2) A = #181
       LDAB   SEED_HIGH           ; (2,3) B = the high byte of the seed
       MUL                        ; (1,10) D = A x B
* low byte of MUL result is added to the high byte of scratch
       ADDB   RandomScratch       ; (2,3) B = B + scratch_high
       STAB   RandomScratch       ; (2,3) scratch = seed * 181
*
       LDD    RandomScratch       ; (2,4) D = scratch
       ADDD   #ADDER              ; (3,4) D = D + 359
       STD    RandomSeed          ; (2,4) remember new seed value
* (A = SEED_HIGH from ADDD instruction)
       PULB                       ; (1,4) Restore the value of B
       RTS                        ; (1,5) A holds the new 8-bit random number

*************************************************************************
*      DelayX:  Delay based on the value in.
*************************************************************************
DelayX:
       PSHX
LoopX:
       DEX
       BNE    LoopX
       PULX
       RTS
```

Conclusion

I was able to develop a good random number generator for the 68HC11 that was quite small (25 bytes) and which met all of my goodness criteria. This was an equation-based algorithm using the equation S = S * 181 + 359, with the returned random number the top 8-bits of the 16-bit S value. A table-based algorithm was also developed, which also generated a good sequence of random numbers, but the algorithm and table required 190 bytes of program space, much larger than the 25 bytes of the equation-based algorithm. The resulting 68HC11 code was successfully tested on an HC11E1 system.

About The Author

Tom Dickens is an engineer and Associate Technical Fellow at The Boeing Company, specializing in the design and implementation of hardware and software systems. He also teaches evenings in the Seattle area at Boeing, the University of Washington, the University of Phoenix (Washington campus), and Henry Cogswell College. Tom has been an active member of the Seattle Robotics Society for 15 years, has been the editor of the SRS newsletter (the Encoder), has served as the SRS secretary, and has served for a numbers of years on the SRS board of directors and the SRS Robothon committee. Tom maintains a web-site dedicated to the 68HC11 microcontroller at:

http://tomdickens.com/6811/intro.html

References

[1] Whatis.com: “white noise”, http://whatis.techtarget.com/definition/0,,sid9_gci213526,00.html

[2] ISA Card Random Number Generator, http://www.cryogenius.com/hardware/isarng/

[3] Hardware Random Number Generator, http://www.cryogenius.com/hardware/rng/

[4] A True Random Number Generator, http://www.vaxman.de/projects/rng/rng.html, ulmann@vaxman.de.

[5] Press, William H. et al, “Numerical Recipes in C, second edition,” Cambridge University Press, 1992, page 284. http://www.library.cornell.edu/nr/bookcpdf.html