Random Number Generation for IoT Devices

Computer operating systems usually provide functions that generate random numbers for applications but for a small IoT device, the code running on the micro must usually include this capability itself. However, generating even pseudo random numbers in software can present a challenge. A small micro like those typically used in IoT devices may already be tight on code space or RAM so there are limited resources to work with. To compound the problem, for many engineer/programmers, just changing their mindset to write software that produces a different result each time it is run can present a significant hurdle to get over since it is such a major departure from their usual way of thinking.

The security of an encryption algorithm that utilizes random numbers for "keys" is only as good as the degree of randomness in those random numbers. A repeating pattern in the random numbers typically results in repetition in the encryption ciphers, making the algorithm easier to break. Quality random number generation with minimal repetition and predictability is critical for good data security.

The three most important aspects of random number generation in an IoT application are:

1. Incorporating a number of data sources of different types in the random number generation. Time and environmental conditions are two excellent sources for data sources for random number generation.
2. Continuous generation of random numbers improves the degree of randomness and can significantly reduce the time required to generate a key before an encyrpted message can be sent. Instead of only generating a random number when it is needed, modify the random number frequently, every time a data source is read, every tick of a system tick interrupt, every pass through the program's main loop, etc.
3. There should be a high probability of bits being set across the length of the random number and ideally one or more bits within each byte of the random number will change each time a new number is generated. As an example of what not to do, for a 32 bit random number just adding an 8 bit value to obtain the next value can require thousands of addition operations to affect the upper 8 bits of the number. The result may be "random" numbers but using a series of keys with only slight difference between them can compromise the security of the encrypted data.

Having multiple data sources and varying which data sources are used when generating a random number can greatly increase the degree of randomness. Most IoT devices will have a number of existing hardware data sources and easily created firmware data sources that can be utilized for random number generation. Examples of these data sources would include:

- Voltage readings – Battery voltages change over time, over temperature and vary based on load. Even regulated voltages have switching noise, ripple and fluctuations due to load or input voltage changes. These changes may be relatively small but can easily be "amplified" in firmware. With continuous random number generation or if used with several other sources the magnitude of the changes isn't necessarily important.
- Temperature readings – Similar to voltage readings, temperature readings are usually fairly stable within a certain range particularly over short time periods. Continuous random number generation doesn't require large changes and using data modification techniques like those discussed below can help make large changes in the random number.
- Application sensor readings – IoT devices almost by definition have one or more sensors monitoring some physical condition. The readings from these sensors should also be used in random number generation.

- Time from last reset tick counter – Adding a counter to a system tick ISR requires just a few bytes of code and data space but will provide an ever changing value to be used in random number generation. To be more effective, initialize this counter to a large arbitrary number such as 0x20374a19 instead of zero and add a similar value to it instead of just incrementing it by one.
- Real time clock – This is a data source that has significant changes over time. If the RTC is the type that just uses a simple counter then that counter value can be used as is. If the RTC provides actual date and time values then it provides several values that can be used in the random number generation.
- Last network response time – Network response times almost always have some degree of variation at the millisecond or tens of milliseconds level (or seconds when dealing with the Internet). Using the response time of the last network access or adding that time to an accumulator can produce a large number to be used in random number generation. The same can be done with other variable response times in the application.
- System activity accumulator – For every event the device deals with, add a large value such as 0x1032421 to an accumulator or the random number itself. This can be done for timer ticks, each time sensors are read, every network access, etc. If these types of events occur at a much higher frequency than when a random number is needed, this technique by itself can produce numbers with an adequate degree of randomness over a short period of time. Use different values for different types of activity to improve randomness.
- Wireless radio RSSI value or signal strength value – Most wireless radios provide a means to read a signal strength measurement or other link quality measurement. These may fluctuate as objects more around in an area or with interference from other wireless radios or other RF noise sources (motors, microwave ovens, etc).
- Device ID – These can be used as the “seed” starting value for the random number generator to minimize the possibility of several devices of the same type generating the same series of numbers. A number of modern micros have unique 32 or 64 bit devices IDs that could be used for this. A device’s network address could also be used. Modify these numbers based on one or more of the data sources to provide a higher degree of randomness.

Using existing data sources and low overhead firmware data sources makes it easy to add quality random number generation and data encryption capabilities to existing products.

If an IoT device has a limited number of natural data sources or if the values are mostly 8 or 10 bit, there are several tricks that can be used to provide additional data values or increase the width of the data values. Some of these data modification tricks include:

- Reverse the bits in a value – This requires a small code loop but can produce a value numerically very different from the original value, for example 0x25 becomes 0xa4 when the bits are reversed. For 16 or 32 bit values the numeric differences are even greater. For a 32 bit value you can reverse the nibbles/bytes for 8/4 cycles through the code loop instead of 32 for bit reversal.
- Invert the bits in a value – This simple XOR with all 1’s also produces a value numerically very different from the original value with a single instruction on most micros.
- Interleave bits from several values – While it is easy to make a 32-bit value from four 8-bit values, if one or two of those values don’t change significantly then only small changes will occur in the 32-bit value. Interleaving the 8-bit values together will spread the changes in one value over more bits within the 32-bit value. Reversing the bit order of one or two of the values while interleaving the values will usually spread the changes over more bits.
- Multiple shift and add of values – Performing multiple cycles of shifting a value and adding the original value is a quick and simple way to increase the width of a value while producing a value numerically very different from the original value.

- Use signed changes in readings – With values that change little between consecutive samples (such as voltage and temperature), using signed changes in the readings can produce big value changes as an ADC reading goes up/down by even just 1 or 2 LSB.
- Expand small values – Similar to interleaving bits from several values, an 8-bit value can easily be converted to a 16-bit value using every other bit or a 32-bit value by using every fourth bit.
- Accumulate small or infrequent changes - For values with infrequent significant changes (such as voltage or temperature), accumulate the readings and add the accumulated value to the random number instead of using the actual values.
- Create additional values – Add or subtract the values from different data sources to create another value. This is particularly effective when one of the data sources changes frequently compared to the other. If the micro has a hardware multiplier then multiplying two values can produce much larger values as quickly as addition or subtraction.

If having truly random numbers is particularly important in a specific application, there are several ways to provide fairly inexpensive hardware assists to the random number generation. Several examples would include:

- Sine-wave or triangle-wave – Using an oscillator or the PWM output from a micro (with an RC circuit to provide smoother rise/fall) and feed the signal into an ADC. When using a PWM to generate the signal, the signal doesn't need to be particularly clean and vestiges of the original PWM signal riding on the produced signal can serve to increase randomness. The signal should run continuously when the micro is awake so it can be sampled at any time to provide a truly random value.
- Use a micro GPIO to charge/discharge a cap through a resistor and read with an ADC – The signal rise/fall time will be fairly consistent (determined by the resistor and cap values) so the charge/discharge should run asynchronously to the code execution so it can be sampled at any time. Alternatively, use fairly short rise/fall times and a random value to delay from starting the charge/discharge to reading the ADC value instead of running the circuit constantly.
- Amplify power supply noise/ripple – A small op-amp circuit can be used to produce a signal that amplifies short term power supply noise and other fluctuations to be read with an ADC. With a switching regulator based power supply, this approach has the added benefit that the switching regulator naturally operates asynchronously from the micro and its code loops.

With these added data sources, using the actual reading is obvious but also using the change from the previous reading or accumulating a value creates additional values to be used in the random number generation.

Below are C code snippet examples for a continuous random number generation scheme. These examples assume the code will run on a 32-bit micro with a barrel shifter and hardware multiplier but the same concepts can be applied to many lower-end micros as well.

In a tick timer ISR:

```
free_run_ctr += 0x25610d43;
```

Each time through the program main loop:

```
free_run_ctr += 0x16042159;
```

When reading an ADC value:

```
shift_cnt = free_run_ctr & 0x0000000f;
rand_value += ADC_value << shift_cnt;
```

```
free_run_ctr += 0x12c56703;
if ( reading_temp )
{
	last_temp_reading = ADC_value;
	last_temp_accum += last_temp_reading;
}
```

Each time a message is sent or received:

```
free_run_ctr += 0x81650e74;
rand_value = rand_value << 3;
```

In the random generation function:

```
rand_value = rand_value << 1;
rand_value += free_run_ctr;
x = free_run_ctr & 0x00000030;
y = free_run_ctr & 0x00000007;
if ( x == 0 )
{
	rand_value += last_volt_reading << y;
	rand_value += last_temp_accum << y+5;
	rand_value += free_run_ctr & 0x000003ff;
	rand_value += free_run_ctr & 0x05fd0000;
}
else if ( x == 0x10 )
{
	rand_value += last_temp_accum << y;
	rand_value += last_volt_reading << y+14;
	rand_value += free_run_ctr & 0xffc00000;
	rand_value += free_run_ctr & 0x00007fe0;
}
else if ( x == 0x20 )
{
	rand_value += (last_temp_reading + last_volt_reading) << y;
	rand_value += (last_volt_reading - last_temp_reading) << y+17;
	rand_value += last_volt_reading * last_temp_reading;
}
else if ( x == 0x30 )
{
	rand_value += (free_run_ctr >> 24) & 0x000000ff;
	rand_value += (free_run_ctr >> 8) & 0x0000ff00;
	rand_value += (free_run_ctr << 8) & 0x00ff0000;
	rand_value += (free_run_ctr << 24) & 0xff000000;
}
free_run_ctr += 0x5d16a491;
rand_value ^= free_run_ctr;
```

Using the types of data sources and techniques discussed here, it is possible to achieve a high degree of randomness using little code space and with little execution time overhead. Continuous random number generation is particularly important for IoT applications since it can increase the degree of randomness while not adding a long latency for the random number generation.