RANDOM NUMBERS AND LORA

LoRa chips bring an unexpected bonus, related to radio frequencies: random numbers generation. Both the SX127x and the SX126x series provide this service, through one or more registers. In the SX127x series, the process is a little more involved (and some LoRa libraries for this chip get it wrong), but offers more control on the data generated (including adding whitening with eg a von Neumann extractor). On the other hand, the SX126x makes it easier, but the process is quite opaque, and you are delivered 4 random bytes without being given any control. Let's have a look.

SX127x

RegRssiWideband

In these older LoRa chips, what you do is read Register 0x2C, the RSSI Wideband Register. This Wideband RSSI measurement can be used to generate a random number, but you have to be careful with the data – if you return the contents of this register and hope to have a number anywhere near random, you'll be disappointed. What you need to do is extract the lowest significant bit, bit 0, which is much more probable to be as close to truly random as possible than its neighbours, and add ito to a pile of other LSB taken successively from this register. This will create a possibly random number between 0 and 255. With the help of a randomness extractor, you can generate a highly random output that appears independent from the source and uniformly distributed.

With the von Neumann extractor two bits are taken at a time (first and second, then third and fourth, and so on). If the two bits match, no output is generated. If the bits differ, the value of the first bit is added to the stream, and the second discarded. This is quite simple to implement in C/C++.

```
uint8_t getLoRandomByte() {
  uint8_t x = 0, b;
  for (uint8_t j = 0; j < 8; j++) {
    b = readRegister(RegRssiWideband) & 0b00000001;
    while (b == readRegister(RegRssiWideband) & 0b00000001) {
      // von Neumann extractor.
      b = readRegister(RegRssiWideband) & 0b00000001;
    }
    x = (x << 1) | b;
    //delay(1);
  }
  return x;
}
```

`readRegister(RegRssiWideband)` reads the RSSI Wideband registers and discards all bits except bit 0. A second read from the register is compared with the first one, and if they are identical, the function loops, until they differ. Once they do, the first bit is added to the 8-bit variable, `x = (x << 1) | b;`, and it starts again, until there are 8 random bits, `for (uint8_t j = 0; j < 8; j++)`.

Once you have this set up, you can add more functionalities: fill a buffer of XX bytes, return a number between X and Y, etc.

Setup

However, there’ s a little more work to do. This only works when the SX127x is in a special working mode, continuous receive mode. In continuous receive mode, the modem scans the channel continuously for a preamble. Each time a preamble is detected the modem tracks it until the packet is received and then carries on waiting for the next preamble. Which is why the RegRssiWideband gets filled non-stop with pseudo-random data, and reading from it can be used to generate random numbers. You also need to set the SX127x chip to SF 6.

```
uint8_t modemconf10;
uint8_t modemconf1;
uint8_t modemconf2;

void setupLoRandom() {
  // First save current settings
  modemconf0 = readRegister(RegOpMode);
  delay(10);
  modemconf1 = readRegister(RegModemConfig1);
  delay(10);
  modemconf2 = readRegister(RegModemConfig2);
  delay(10);

  // 1: LoRa mode
  // 0: Access LoRa registers page 0x0D: 0x3F
  // 00: reserved
  // 1: Low Frequency Mode (access to LF test registers)
  // 101: Receive continuous (RXCONTINUOUS)
  writeRegister(RegOpMode, 0b10001101);
  delay(10);

  // 0111: BW 125
  // 001: CR 4/5
  // 0: Explicit Header mode
  writeRegister(RegModemConfig1, 0b01110010);
  delay(10);

  // 0110: SF 6
  // 1: continuous mode, send multiple packets across the FIFO
  // 0: disable CRC check on payload
  // 00: RX Time-Out MSB
  writeRegister(RegModemConfig2, 0b01101000);
  delay(10);
}
```

The chip is now in a working mode suitable for RNG. Once you are done, you reverse the process:

```
void resetLoRa() {
  // reset settings
  writeRegister(RegOpMode, modemconf0);
  delay(10);
  writeRegister(RegModemConfig1, modemconf1);
  delay(10);
  writeRegister(RegModemConfig2, modemconf2);
  delay(10);
}
```

So as you can see there’ s quite a bit of work, which I simplified by writing a small library, LoRandom. But then came the SX126x series, and I had to start all over again...

SX126x

With the newer chips, the process is much easier. There are 4 registers in sequential order, `0x0819` to `0x81C` that provide 4 bytes of randomly generated data. As is often the case with Semtech, while the docs are often thick and detailed, they tend to skip the interesting bits, and we’ re left with just this: “Can be used to get a 32-bit random number.” Oh, well...

So in my Sx1262LoRandom version of the LoRandom library, I made a simple function, `uint32_t fillRandom()` , that fills up a 256-byte buffer, from which you can draw. In this example, it is a permanently enabled buffer, but it could be easily changed to pass a reference to your own buffer.

```
uint8_t randomStock[256];
uint8_t randomIndex = 0;

void fillRandom() {
  uint8_t regAnaLna = 0, regAnaMixer = 0, cnt = 0;
  regAnaLna = Radio.Read(REG_ANA_LNA);
  Radio.Write(REG_ANA_LNA, regAnaLna & ~(1 << 0));
  regAnaMixer = Radio.Read(REG_ANA_MIXER);
  Radio.Write(REG_ANA_MIXER, regAnaMixer & ~(1 << 7));
  // Set radio in continuous reception
  Radio.Rx(0xFFFFFF);
  // SX126xSetRx(0xFFFFFF); // Rx Continuous
  for (uint8_t i = 0; i < 64; i++) {
    Radio.ReadBuffer(RANDOM_NUMBER_GENERATORBASEADDR, (uint8_t*)(randomStock + c
nt), 4);
    cnt += 4;
  }
  randomIndex = 0;
  Radio.Standby();
```

```
    Radio.Write(REG_ANA_LNA, regAnaLna);
    Radio.Write(REG_ANA_MIXER, regAnaMixer);
}
```

Like in the SX127x version, the radio has to be set up just so, and you need to save settings before setting it up to work for RNG. And when you’ re done, you need to reset the settings. But all in all the process is streamlined to just a few lines.

The example code for the SX127x works with the LoRa library by Sandeep Mistry (which has a big bug for RNG), but is easily adaptable to any library: all you need to do is provide two functions to read and write registers.

```
void writeRegister(uint8_t reg, uint8_t value);
uint8_t readRegister(uint8_t reg);
// Provide your own functions, which will depend on your library
```

The code for the SX126x version is much more integrated to the Wisblock / RAK universe, and will work as is on RAK4631, and mostly without changes on the other LoRa-enabled MCUs in the lineup. It won’ t be very difficult to adapt them to other MCUs / devices, but that’ s way beyond the scope of this article.

As a side note, the nRF52840 inside RAK4631 has another way to generate random numbers, with the CC310 cryptographic chip embedded inside. I cover this in my fork of the Adafruit nRFCrypto library, which adds the AES and Chacha and ciphers. But even the original library has the Random object.