Entropy on the Arduino
No Guesswork – Just Statistics

This document is a detailed description of my efforts to measure entropy sources easily available on the Arduino. I go through a detailed, step-by-step progression of implementing different entropy sources, testing them, discovering their failures, and analyzing the results. I include all errors, false impressions, and downright stupidities. I did dumb stuff… so you don't have to!

WTF?
Why Entropy On Arduino's?

A while back, one of the Hackaday writers put out a none too subtle call for a one-time pad as a dedicated piece of hardware. That got me thinking (never particularly safe) and I started throwing together some basic design parameters. One thing however, kept nagging me. The problem is that cryptography is such a subtle and unforgiving practice. Catastrophic failures can occur silently, without you ever knowing that they happened.

Instead of jumping all the way into building an OTP device (dodging that particular bullet), I decided to focus on just one part of such a system in greater detail. Specifically, random number generation. One of the first steps toward building such a device (or anything else implementing cryptographic functions), is to find a method of generating excellent random numbers that is inexpensive and relatively easy to implement.

I thought that an unbiased(that's an entropy joke, you should laugh now) analysis of various possible sources of entropy available on the Arduino platform would be of use to the community. I found several written accounts of people who have done similar work, but few compared multiple methods, many didn't publish their test results, and some were incomprehensible to non-experts. Nevertheless, there is good data out there, just not compiled in an easily accessible manner. I've prepared this document with the hope that it can serve as a useful resource for the community by comparing multiple methods, giving test results, and explaining at least some of the reasons why various methods are unsuitable for cryptography. All errors are my own, this information is provided without warranty, your mileage may vary, and if I did make any mistakes… please be gentle. I am not a cryptography specialist or a mathematician so my contribution to this body of knowledge is restricted to using tools that cryptography specialists and mathematicians have created, measuring different sources of entropy, accurately reporting the measurements, and deriving recommendations.

So why the Arduino? Simple. ~~I just want to hang out with the cool kids.~~ They're everywhere. Got a problem? Throw an Arduino at it! They are practically ubiquitous in the community, are inexpensive, and are working their way into all sorts of projects that few ever anticipated. They are already starting to show up in sensitive applications that require secure communications. Therefore, we need to know what sources of entropy can be reasonably implemented on the Arduino (keeping in mind codespace and performance limitations) and whether or not they are trustworthy.

Random Number Generators
“9… 9… 9… 9… 9… 9… 9…”

The Arduino comes with a pseudorandom generator(PRG) which is called using the random() function. A PRG is a mathematical formula, the output of which appears random if you don’t know the number that served as the starting point for the calculation. That starting number is called the seed. Unfortunately, a PRG will produce the same exact pattern of “random” numbers if given the same seed. Therefore, if somebody finds out what number you used to seed a PRG, they can find out every single “random” number you have used. This is not a problem with a true random number generator(RNG) which obtains its random numbers from a real-world source of entropy such as noise in a diode or radioactive decay. Aside from the random() function, there are a few 3rd party libraries for random number generation and a great many suggestions floating around on the internet. That’s nice because nothing is more comforting that unverified internet suggestions on the subject of cryptography.

For practical purposes I needed to answer a few questions. If random() is good enough for use in encryption, then I would need to find a second, very good source of random numbers that I could use as a seed. One that was *not* a PRG, but was a true source of entropy. However, if random() was not good enough, then I needed to find a complete replacement for it.

And just how would I know if it was random enough? If a random number generator consistently passes standard tests for randomness and has resisted attack by knowledgeable cryptanalysts for a considerable length of time, then it is considered suitable for cryptographic applications. The research given here focuses only on standard tests because all of my cryptanalysts were on vacation.

And finally, just because a technique is found to be unsuitable for cryptographic use doesn’t make it useless. When you need an LED to blink randomly, don’t waste the processing time and code space on a really good RNG when a simple one or a convenient one will do the job.

The Testing Suites
Down The Rabbit Hole

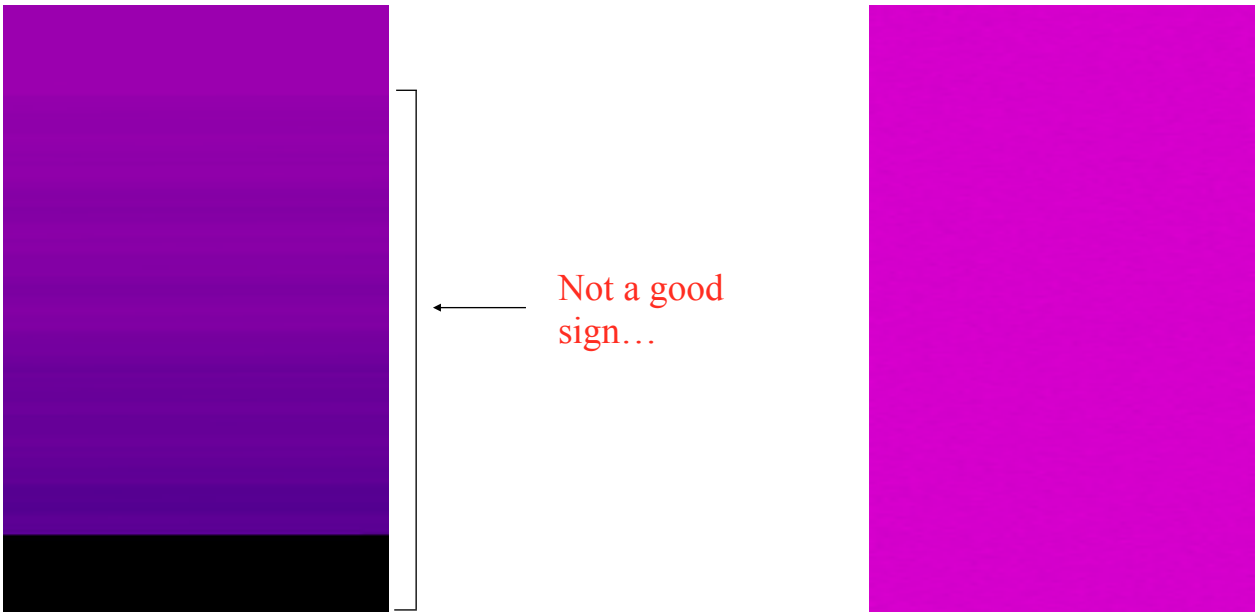
There are several entropy test suites available and each has its proponents and detractors. I applied two different testing suites to the random numbers generated on the Arduino. First is “Ent”. Initially it analyzes any file fed to it as 8-bit numbers and displays a breakdown of the total occurrences and percentage of the 256 possible values. This output can give you an almost instantaneous insight into the trouble with poor entropy. Some values are obviously more common (or less) than others when the entropy source is particularly bad. It also performs six separate tests and gives you a score for each one. Unfortunately, the documentation isn’t clear on what constitutes a good score or a bad score on three of the tests. That led me to read up on recommendations by others who have done work in this area.

From this research I compiled criteria to account for a “Pass” or “Fail” on the various tests. I was very conservative in this regard so it is more likely that good entropy was failing those three tests than bad entropy was passing them.

The second piece of software is the granddaddy of entropy testing. It’s called the Diehard suite and it performs many different tests on whatever file you throw at it. One test continually crashed so it is not reflected in my results. Diehard produces an esoteric and detailed report giving the results. Wading through it is a bit of a task, but I do recommend it for those who really want to thoroughly understand what’s being done to test randomness. The brief summary is that each test gives a probability value (p-value). The p-value of each of the tests is actually the end product of multiple runs though the data. As Diehard runs the test and factors each successive p-value into a final p-value, it will do one of two things. It will either return a number somewhere between 0.0 and 1.0 or it will return a 0 or a 1. A 0 or a 1 is a hard failure of the RNG. *Any other value, no matter how close to 0 or 1, is not a failure*. Some people have erroneously believed that a number within a few decimal points of 1 or 0 is a failure, but the documentation clearly states otherwise. No test exists that can positively confirm randomness. They can only positively confirm a lack of randomness. A 0 or a 1 is confirmation that the output is not random. No p-value other than 0 or 1 indicates a lack of randomness.

The diehard suite recommends a minimum of 10 MB files so I made sure I had double that for each test. The files were run through the two test suites and the output of each suite was then saved for anyone who wants to look at the raw test results. Finally, I made a short summary giving the source of the entropy and a Pass/Fail notation for each of the 24 total tests run on it. One note: The overlapping permutations test in the diehard suite crashed consistently and is not included in my results.

Finally, I used the website binvis.io to produce visualizations of Shannon entropy in the binary files produced by the various entropy sources. The pictures don’t give an in-depth analysis of entropy, but they do give you an easy way to see whether a particular method is worthy of consideration or just plain terrible. It also makes this document a lot less boring because, Pictures! Black regions have low Shannon entropy, bright purple regions have high Shannon entropy. Here’s a sample of a particularly bad method, analogRead() on an unconnected pin. Next to is is the output of a well-regarded PRG called Yarrow (not run on an Arduino). As you can see, Yarrow is the clear winner. Duh.



Bad Things™
(Brought To You By Good Intentions, Inc.)

I started my analysis with the random() function since it is built-in and easy to work with. Initially I researched a bit about its inner workings. The Arduino development environment uses the C library "avr-libc". If you check out the file stdlib.h (standard library) you will find a reference to "RANDOM_MAX" which defines the maximum possible value returned by the random() function as 0x7FFFFFFF. In binary, this translates to a 0 followed by thirty one 1's. The random() function always returns a 32 bit number so this means that there is always a 0 in position 32. I don't know if this is necessary in order to account for a hardware limitation, to maintain backwards compatibility, or some other perfectly legitimate reason, but it means Bad Things™ in the world of cryptanalysis. An adversary knows every thirty-second bit will be a 0. Here is a sample output of a few numbers generated by random() expressed in binary:

00010000110101100011101011110001
01100000101101111010110011011001
00111010101101010000110000101010
01000100001100011011011110000010
00011100000001101101101011001000

Each number has a zero in its most significant bit and that pattern is continued no matter how long you let it run. Obviously, calling random() without correcting this is no good. By default, random() returns an unsigned long, but you don't necessarily have to use the entire 32 bits. Here's one option.

unsigned int myRandomNumber = random();

By assigning the returned value to an unsigned int (16 bits on the Arduino) you strip the leftmost 16 bits off and simply lose them. In the process you also lose that annoying 0. Heres a sample output of the above line of code expressed in binary:

1011011110000010
1101101011001000

1000111011011000
0000100111111110
0010111101000011

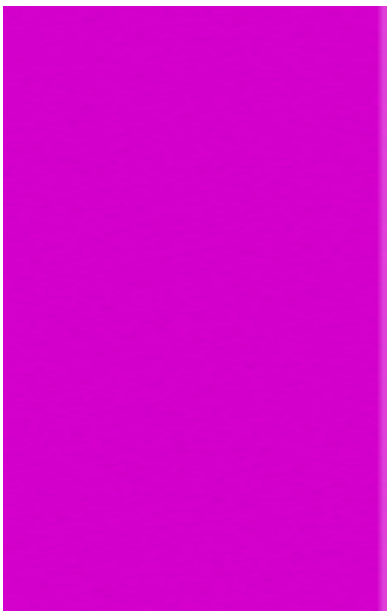
Much better! Now any of the bits can be either a 1 or a 0. Another way to do it that only loses 8 bits rather than 16 is to break up the four bytes and then only use the lower three, like so.

```
unsigned long rand_number = random();
byte byte2 = rand_number >> 16;
byte byte1 = rand_number >> 8;
byte byte0 = rand_number;
```

This gives you three separate bytes corresponding to the lower three bytes of the original unsigned long. I also tried random(255) giving the function a maximum value to calculate that is equal the the maximum value a byte can contain.

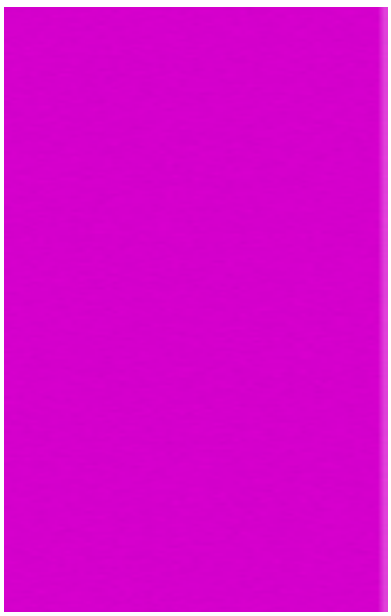
Here's a comparison of random() using all 4 bytes vs. using only the lowest 3 bytes vs. only allowing it to calculate a byte's worth of data per call of the function. Each sample is exactly 20 megabytes in size (20,971,520 bytes).

random() 4 Bytes



random() 3 Bytes

random(255)



From an initial look, there's not much to differentiate them. To get a better idea of their quality I also ran ent and diehard on them. Here is the summary of the results.

	4 Bytes	3 Bytes	(255)
Ent Tests:			
Entropy:	Pass	Pass	Pass
Compression:	Pass	Pass	Pass

Chi-Square:	Fail	Pass	Fail
Arithmetic Mean:	Fail	Pass	Pass
Monte Carlo - Pi:	Fail	Pass	Fail
Serial Correlation:	Fail	Pass	Pass
Diehard Tests:			
Birthday Test:	Fail	Pass	Pass
Binary Rank 31x21 Test:	Fail	Pass	Pass
Binary Rank 32x32 Test:	Fail	Pass	Pass
Binary Rank 6x8 Test:	Fail	Pass	Pass
Bitstream 20-Tuple Test:	Fail	Pass	Pass
Bitstream OPSO:	Fail	Pass	Fail
Bitstream OQSO:	Fail	Pass	Pass
DNA Test:	Fail	Pass	Pass
Count The 1's Stream:	Fail	Pass	Pass
Count The 1's Specific:	Fail	Pass	Pass
Parking Lot Test:	Pass	Pass	Pass
Minimum Distance Test:	Pass	Pass	Pass
3D Spheres Test:	Pass	Pass	Pass
Squeeze Test:	Pass	Pass	Pass
Overlapping Sums Test:	Pass	Pass	Pass
Runs Test:	Pass	Pass	Pass
Craps Wins Test:	Pass	Pass	Pass
Craps Throws/Game Test:	Pass	Pass	Pass

Much to my surprise, the 3-bytes method passed all tests. I actually was not expecting any of them to reach that level of quality. So, from the viewpoint of standard tests, we actually have a decent source of random numbers on the Arduino, the built-in PRG. The only caveat is that one must use it correctly, i.e. omitting the most significant byte. Nevertheless, we must also remember that it is necessary to seed the PRG or it is completely predictable. Therefore, we also need a secure source of seed numbers that come from a true random number generator. We need to do some measurements!

Issues With Measurement
Measure Once, Cut Twice… Wait, No!

Before I go any further, I have some admissions to make. I made bad, bad, bad errors in my initial efforts that resulted in comically awful results from many of the sources of entropy I tested. First off, a silly error, but one I didn't catch until after I had accumulated a great deal of data for testing. The two functions I had been using to get data off the Arduino were Serial.write() when I was using the serial connection over USB, and SD.write() when I was using an SD card. Both of these functions send a single byte only, no matter what kind of data type you send to them. If you try to send and int or a long then the most significant bits are thrown out and only the lowest 8 bits are transferred. The moral of the story is; collect your entropy as single bytes, or break larger data types down into a series of bytes and then send them individually if you need to use one of these methods of collecting data.

The next error was far more subtle and took a long while to find. Every source of entropy I measured was missing the values 15 and 22. I could generate millions of bytes of data using ANY method imaginable, but I only occasionally got a byte valued at 15 or 22. I posted the odd phenomena and a plea for suggestions on the AVR Freaks forum and the problem turned out to be the way I was getting the data off the Arduino. I had been using Serial.write() to pull data up to my computer via USB, and then used the command "screen" to pull data off the terminal and log it into a file. I had suspected that the screen command might be doing something weird so I tested it on /dev/random, the standard entropy source on most linux/unix systems. This worked just fine, no missing 15's or 22's, so I discarded the theory that screen was the problem and went back to thinking I was going crazy. It never occurred to me that there was another piece of software operating in-between the screen command and the Arduino; the TTY. Apparently the TTY was interpreting those bytes as commands for it to do something as opposed to data that should be passed along. The result was falsified measurements. To solve this, I needed a less intrusive way of getting data off the Arduino, and settled on using an SD card to go back and forth. No more missing values! Of course, I only needed to get the stored random numbers off the Arduino in order to measure them. In most cases you wouldn't need to worry about these problems since one does not normally exfiltrate their own random numbers. That's what adversaries are for!

analogRead() On An Unconnected Pin
And Other Lies I Read On The Internet

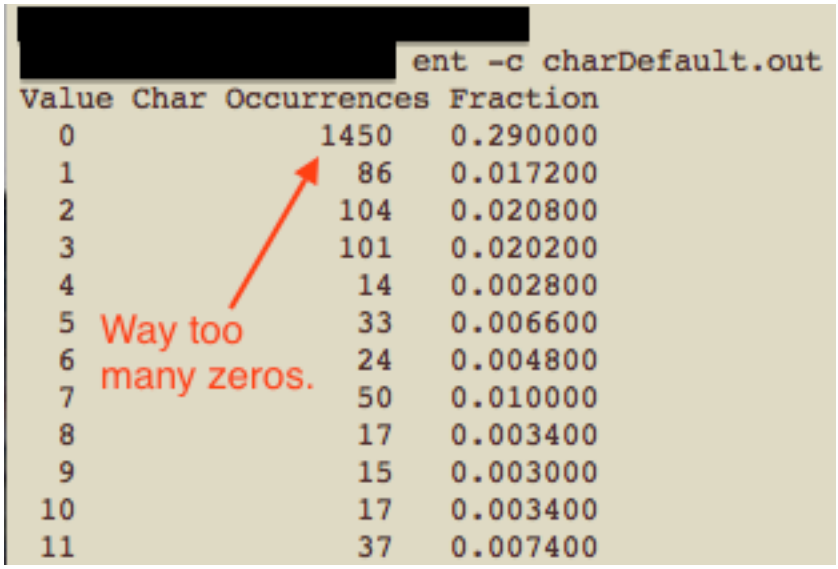
If you look for "Arduino random numbers" on any search engine you will inevitably come across the recommendation to use the analogRead() function on a pin that is left floating with no connection and to use the returned value as a seed for the random() function. It's my sad duty to tell you that every time you do this, somebody somewhere drowns a kitten. Please don't ever do it again, I like kittens. In all seriousness however, this is fine if you are just blinking an LED.

Nevertheless, when you get something making random numbers for you, you should actually measure those numbers correctly. Otherwise, what's the point? If you have some professional hardware RNG generating digitized values then you're all good, but if you need to sample an analog source then you had better know how to use the Analog to Digital Converter. Even if the source isn't very good we should still give it a fair chance by not measuring it wrong… which I was doing for a long while.

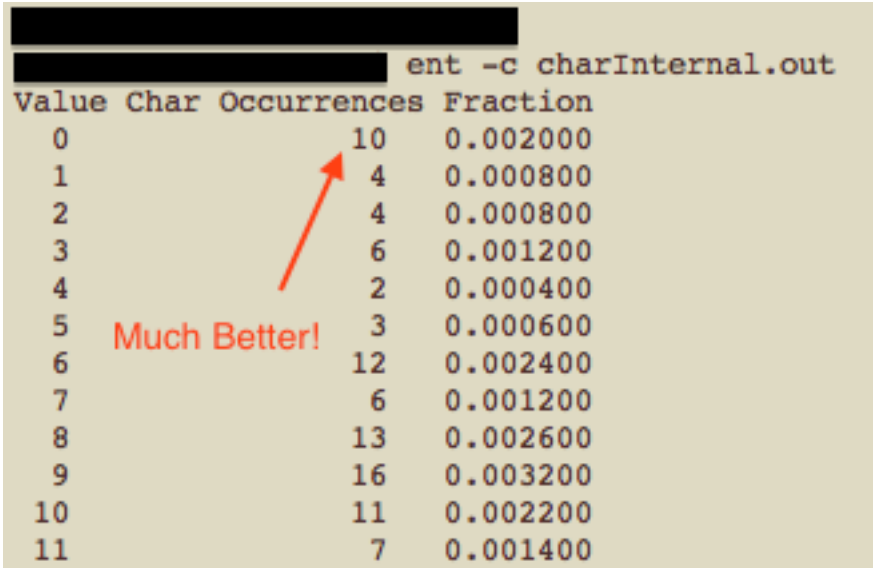
When you take a measurement using analogRead() you are asking the ADC to sample the voltage on the specified pin and report back what that voltage is. By default, it measures from 0 volts to 5 volts on 5V Arduinos or from 0 volts to 3.3 volts on 3.3V Arduinos. Mine is an Arduino Uno and operates at 5 volts. The ADC is a 10-bit device and 10 bits = 1024 possible values it could return for whatever it see's on the pin, numbered from 0 - 1023. Because 5 volts/1024 = 0.00488, the voltage detected on the pin will be measured as being a multiple of 0.00488. For example, if the voltage on the pin is 1.07000 volts, the ADC will measure that the voltage is 1.06872 volts as that is the closest value divisible by 0.00488. Because 1.06872/0.00488 = 219, the number that you actually get back from calling analogRead() is

219 in this case. But what happens when the measured value is less than 0.00488 volts? It reports back a 0. What if it's more than 5 volts? It reports back a 1023. What if it's much more than 5 volts? Nothing, because you broke your Arduino.

Following is the first few lines of the output of "ent -c" on a file containing 5000 bytes of analogRead() on an unconnected pin. As you can see, there are an awful lot of zero's. In fact, 29 percent of all the measurements taken by the ADC were reported back as zero, meaning there was 0.00488 volts or less on the pin.

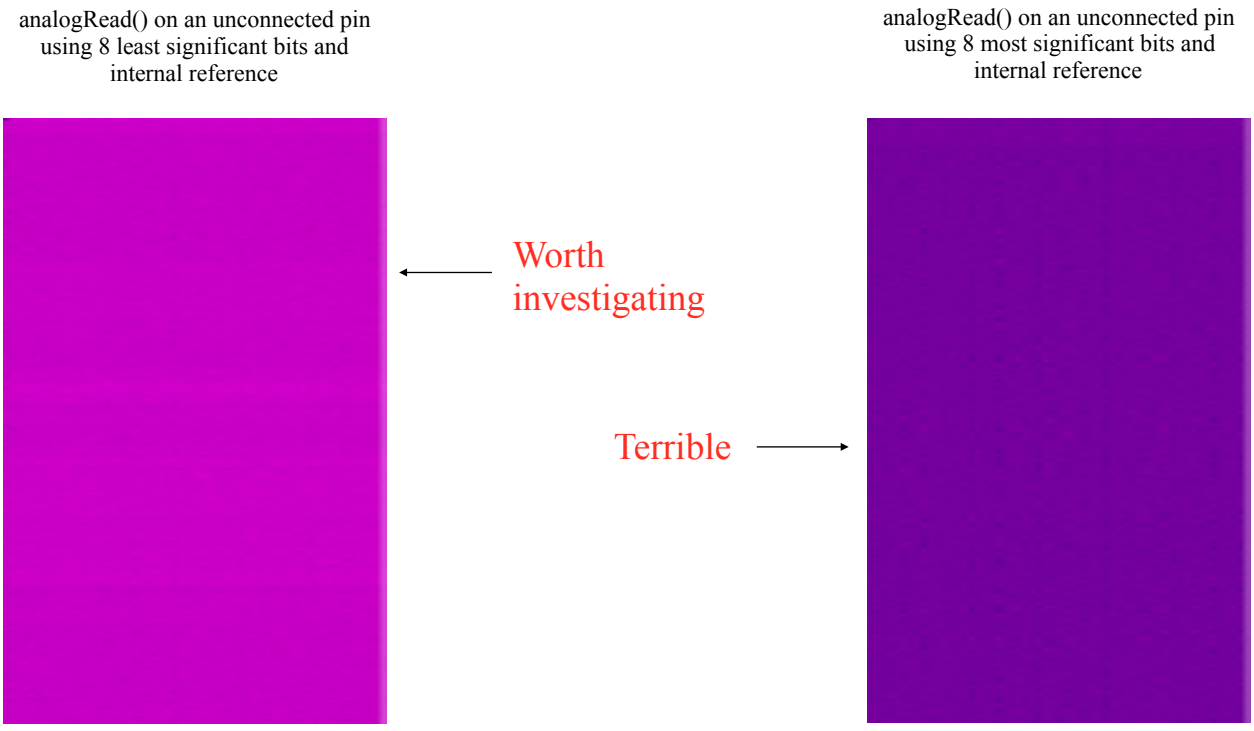


The way to solve this is to change the maximum reference voltage of the ADC. This can be done with the analogReference() function. The two easy options are analogReference(DEFAULT) which uses the system voltage and analogReference(INTERNAL) which uses an internal 1.1V power source (2.56V on some boards). Here's the result using analogReference(INTERNAL).



A third option, analogReference(EXTERNAL), is also available and allows you to apply a voltage in-between the Internal and Default voltages to the AREF pin and use that as the reference. I suggest caution in doing so because you can break your Arduino if you do that without following the proper sequence both in the hardware and software. Don't say I didn't warn you! Nevertheless, it may be necessary if you get a bunch of 0's using EXTERNAL, but then get a bunch of 1023's when using INTERNAL. Setting the AREF pin to an intermediate value can correct this situation. Unfortunately, there's no way to know which is more appropriate for your given circuit except to try both and measure them.

Now that we are measuring honestly, there's still the nagging problem of the "extra bits". The ADC is a 10-bit device and calling analogRead() returns an integer, 16-bits. That means you get 6 bits of zeros, and then the actual number that was returned by the function. Needless to say, this is bad if you attempt to use both bytes of data. We only want to keep 8 of those 10 bits so we have two options; throw away the two most-significant of the 10 bits or the two least-significant of the 10 bits. I tried both and here are the results.



The sample using the least significant bits is clearly better which surprised me at first. After thinking about it a bit, it made sense. Because I'm measuring an unconnected pin, it's just reporting back the ambient temperature for the most part, perhaps modified by stray static in the air. That would result in most of the numbers falling into a similar range. If the range of numbers is pretty consistent, then the most significant bits would almost always be the same. By removing those two bits, rather than the two lowest bits (which probably

fluctuate all over the place) you remove the portion of the consecutive values that changes least often. For a closer analysis, here's the ent and diehard results.

	8 Lower Bits	8 Higher Bits
Ent Tests:		
Entropy:	Pass	Fail
Compression:	Fail	Fail
Chi-Square:	Fail	Fail
Arithmetic Mean:	Fail	Fail
Monte Carlo - Pi:	Fail	Fail
Serial Correlation:	Fail	Fail
Diehard Tests:		
Birthday Test:	Fail	Fail
Binary Rank 31x21 Test:	Fail	Fail
Binary Rank 32x32 Test:	Fail	Fail
Binary Rank 6x8 Test:	Fail	Fail
Bitstream 20-Tuple Test:	Fail	Fail
Bitstream OPSO:	Fail	Fail
Bitstream OQSO:	Fail	Fail
DNA Test:	Fail	Fail
Count The 1's Stream:	Fail	Fail
Count The 1's Specific:	Fail	Fail
Parking Lot Test:	Fail	Fail
Minimum Distance Test:	Fail	Fail
3D Spheres Test:	Fail	Fail
Squeeze Test:	Fail	Fail
Overlapping Sums Test:	Fail	Fail
Runs Test:	Fail	Fail
Craps Wins Test:	Fail	Fail
Craps Throws/Game Test:	Fail	Fail

Okay, that's sobering. Clearly the whole unconnected pin/analogRead() thing just isn't working out very well. But there may still be a way…

Bias
Not Just For Mean People Anymore

Once you have random numbers being generated, there is still another thing to consider. No matter how random your entropy source, the data it produces can still suffer from bias. On average, a random system will produce a fairly even quantity of each possible outcome. If you roll a 6-sided dice 600 times, you would expect to get approximately 100 of each number, but because *it is random*, one or more numbers will occur more frequently than others. This bias could be entirely random, or it could be an indication of some underlying pattern demonstrating a weakness in the entropy source. Techniques designed to remove bias from a stream of random numbers are called (speaking of bias…) "whitening" algorithms.

They can never increase the overall entropy of a data stream, but they can remove bias. Whitening algorithms also slow the output of an RNG because they throw out some numbers from the stream as they do their job. It's possible that using a whitening algorithm could improve analogRead() enough to be useful. Or maybe not. I decided to check it out.

There are two methods that are pretty simple to implement but appear to be very effective at removing bias. The first is simple XOR. I started with XOR since you can apply the method to a bit, byte, or any size variable really. The other method can only be applied to a single bit at a time so that makes it a completely different method of measurement than I have been dealing with thus far. To apply XOR, all you do is take one byte from your stream of random number, then take a second byte and do a bitwise XOR on them. The result of this operation becomes your final random number. Of course, generating the random numbers will take twice as long since you have to take two measurements in order to get one output.

I tried applying this to the least awful analogRead() method I've yet found, analogRead() on an unconnected pin using the 8 least significant bits only… but I won't bore you with the details. It was even worse. Way worse then doing nothing to the bit stream in the first place. It only took me a few minutes to realize the problem. The whole reason reading an unconnected pin isn't very good is because it generally doesn't change much from one reading to the next. What happens when you XOR two very similar numbers together? You get a whole lot of zero's. Example: 00101011 XOR 00100111 = 00001100. This will tend to give you byte values which are predominantly composed of binary 0's. In other words, the "bias reducing" technique was doing exactly the opposite of its intended purpose. In all fairness, with a bit more research I found that I was grossly misapplying XOR as a whitening tool. It is actually intended to be used with two different sources of entropy rather than the same source serially. Live and learn.

I dispensed with XOR as a whitening algorithm for use with analogRead() on an unconnected pin and turned to the other simple and common technique, Von Neumann whitening. To implement this you need to do a whole lot more reads from the analog source because you only use 1 bit at a time. Basically, you take a reading to get just the least significant bit, then take another reading to get another bit and you compare the two bits. If they are both 0's or both 1's, you immediately discard both bits and start over. If the first bit is a 0 and the second is a 1, then you output a 0. If they are reversed, 1 then 0, you output a 1. You link these final 0's and 1's together until you have a full byte (or whatever sized variable you are trying to get). As you can imagine, with the many, many calls to analogRead() that this would entail, it takes a very long time. To collect 20 MB took just over 10 hours. The result was significantly improved but still just not good enough. Here's Von Neumann applied to the least significant byte method compared to the original.

	Low Byte	Low Byte With Von Neumann
Ent Tests:		
Entropy:	Pass	Pass
Compression:	Fail	Pass
Chi-Square:	Fail	Fail
Arithmetic Mean:	Fail	Pass

Monte Carlo - Pi:	Fail	Fail
Serial Correlation:	Fail	Fail
Diehard Tests:		
Birthday Test:	Fail	Fail
Binary Rank 31x21 Test:	Fail	Pass
Binary Rank 32x32 Test:	Fail	Pass
Binary Rank 6x8 Test:	Fail	Fail
Bitstream 20-Tuple Test:	Fail	Fail
Bitstream OPSO:	Fail	Fail
Bitstream OQSO:	Fail	Fail
DNA Test:	Fail	Fail
Count The 1's Stream:	Fail	Fail
Count The 1's Specific:	Fail	Fail
Parking Lot Test:	Fail	Fail
Minimum Distance Test:	Fail	Fail
3D Spheres Test:	Fail	Pass
Squeeze Test:	Fail	Fail
Overlapping Sums Test:	Fail	Fail
Runs Test:	Fail	Pass
Craps Wins Test:	Fail	Fail
Craps Throws/Game Test:	Fail	Fail

Adding Von Neumann whitening got it to pass three of the ent tests and four of the diehard tests. Better, but nowhere near good enough to trust for anything more dangerous than a blinking LED. And not even that if it's a really important LED. Just saying.

"All my brilliant plans foiled by thermodynamics.
Damn you, Entropy"!

In my final tests of analogRead() on an unconnected pin, I ran across yet another problem that makes this such a completely unreliable technique. The XOR sample was taken in the evening, the Von Neumann sample in the morning. Between that evening and the following morning, the environment had changed enough to require changing analogReference() from INTERNAL to DEFAULT in order to get sensible readings. In other words, even if you determine the best reference voltage to use, it could be entirely wrong just hours later. Yet another reason never to trust the floating pin.

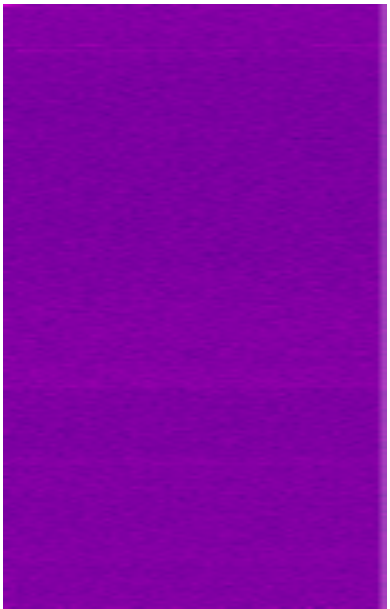
I know that I really ran analogRead() into the ground, but the fact is that I *wanted* to find a way to make it work. I already determined that there is a way to make the random() function good enough for most purposes, maybe even good enough to use in secure applications. It would have been fantastic to find a way to seed random() without having to resort to additional hardware or 3rd party libraries that take up more of my precious codespace. I guess it just wasn't meant to be.

The Tom Sawyer Technique
Get Someone Else To Do The Work For You

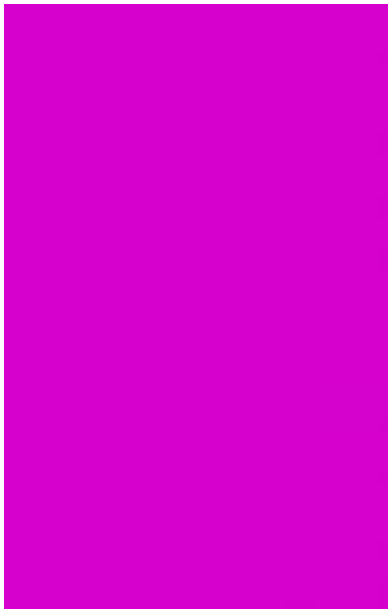
Going in search of a way to seed random() I turned to 3rd party libraries. The first one I tested was the TrueRandom library which generates it's random numbers by "setting up a noisy voltage on Analog pin 0, measuring it, and then discarding all but the least significant bit of the measured value". Hmmm, looks like my favorite function again. I was unable to figure out how it gets a "noisy voltage" but I chose not to worry about it and just tested the values it produced. Much like my quick hack of a Von Neumann whitening algorithm, this library took a considerable amount of time to produce the necessary 20 MB,12 1/2 hours. The library allows for the return of different data types and even some specially formatted types like MAC address or UUID. I stuck to TrueRandom.randomByte().

The second library I tested was the aptly named Entropy library. This one uses a really interesting source of entropy that is completely different than all the others so far examined. The other sources of random (sort of) data relied on analogRead() but this one uses a technique known as timer jitter. Basically, there are four different clocks on an Arduino and they operate at different frequencies, the three "timer" clocks being significantly faster than the watchdog clock. Because no clock is truly perfect, there can be some variation of how many times each of the faster timers will tick during one tick of the watchdog clock. The Entropy library watches these changes and then converts the differences between one watchdog tick and the next into 0's and 1's. After passing this through a whitening algorithm, it returns the random data. A very neat concept, but unbelievably slow. In order to accumulate 20MB to test, I had to run my Arduino Uno for **31 DAYS**. For this reason, the author suggests using it to seed a PRG, not as a primary source of entropy. Luckily, that's exactly what I want to use it for and randomSeed() only requires 4 bytes. On the other hand, if it is a good enough source of entropy and you only occasionally need a small amount of random numbers, it could work just fine on its own. I used the randomByte() function to collect my sample. Without further ado, here's the comparison.

TrueRandom Library
Truerandom.randomByte()



Entropy Library
randomByte()



	TrueRandom	Entropy Library
Ent Tests:		
Entropy:	Fail	Pass
Compression:	Fail	Pass
Chi-Square:	Fail	Pass
Arithmetic Mean:	Fail	Pass
Monte Carlo - Pi:	Fail	Pass
Serial Correlation:	Fail	Pass
Diehard Tests:		
Birthday Test:	Fail	Pass
Binary Rank 31x21 Test:	Pass	Pass
Binary Rank 32x32 Test:	Pass	Pass
Binary Rank 6x8 Test:	Fail	Pass
Bitstream 20-Tuple Test:	Fail	Pass
Bitstream OPSO:	Fail	Pass
Bitstream OQSO:	Fail	Pass
DNA Test:	Fail	Pass
Count The 1's Stream:	Fail	Pass
Count The 1's Specific:	Fail	Pass
Parking Lot Test:	Fail	Pass
Minimum Distance Test:	Fail	Pass
3D Spheres Test:	Fail	Pass
Squeeze Test:	Fail	Pass
Overlapping Sums Test:	Fail	Pass
Runs Test:	Fail	Pass
Craps Wins Test:	Fail	Pass
Craps Throws/Game Test:	Fail	Pass

Damn, that's harsh. I actually really liked the way the TrueRandom library was put together. A lot of thought and effort went into making it as useful as possible for the end user. However, the numbers don't lie, it just isn't using a good source of entropy. The Entropy library, on the other hand, doesn't provide you much more than your choice of a 32-bit random number, a 16-bit random number, or an 8-bit random number, but it is clearly using a good source of entropy. The Entropy library is the clear victor and appears to be fully capable of serving as a source of seed values. Or, if you don't need a lot of random numbers quickly it can simply deliver them directly.

Conclusion
Caveat Emptor

All this can be pretty well summed up in the following way: If you need good random numbers, pick some numbers from the Entropy library and use them to seed the random() function. Then chop up the output and take only the lowest three bytes. Use those bytes to build up whatever length chunk of random data you need. Reseed frequently.

I did a lot of research to get this document to where it is, but that doesn't mean it is complete. I only tested a single 20MB file from each source of entropy. If you want to be really sure, you should test many more, each taken under different circumstances, with the rest of your code running on the test platform to simulate true use cases. Also, there's a whole world of techniques I didn't even try to apply. For example, passing your entropy source through a hashing algorithm as a whitener. Some hashes have been designed for exactly this purpose. Or what about passing an even or odd parity bit to a whitening algorithm instead of using least significant bits? I've heard of it being done, but didn't test the idea. In other words, I've only scratched the surface.

Is this cryptographically secure? Who knows! But it's a heck of a lot better than what many of us have been doing thus far. Maybe someone out there will extend this research and improve it… or prove it totally wrong. In the meantime, I think that the above advice will serve us well.