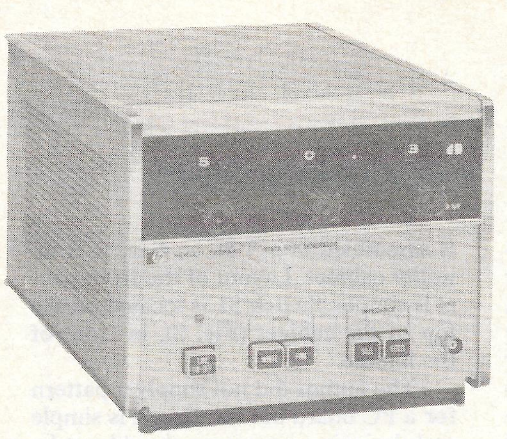
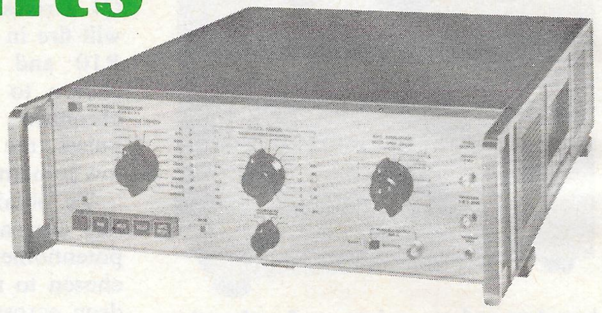


Understanding Pseudo-Random Circuits



Pseudo-random sequences are useful in a number of applications, including music synthesizers. This article explores the circuits that generate these sequences and their applications

by DON LANCASTER

NOISE IS USUALLY THE BAD GUY WITH the black hat in most electronic systems. It's something we want to get rid of, or at least minimize as best we can. But not always! Sometimes noise can be useful. Useful if we want to generate random tone sequences in electronic music. Useful if we want to use it for part of a computer data-security code. Useful to test complex communications gear where all different sorts of signal levels might be present at once. Useful in cryptography for "secret" code generation and transmission. And useful for an elaborate form of communications called signal *correlation,* handy where you use white-hat good-guy noise to extract a signal or some information that may be deeply buried in the bad-guy, black-hat noise you can't control.

The trick is to get noise that *repeats!* Noise that appears random, but repeats in a predictable fashion. Believe it or not, this is very easy to do and takes nothing but a few pieces of TTL or some other logic and an op-amp.

The name of the game is a series of easy-to-generate codes called *Pseudo-Random Sequences.* Besides being interesting and simple to experiment with, you might like to use them in an electronic music system, for audio testing or on a school or fair project.

What do we mean, pseudo-random!

How can noise repeat? What we do is generate a long and involved sequence of digital ones and zeros or analog signal levels. If the sequence is long, the short-term variations *appear* to be completely random and unpredictable and have essentially the same power and statistical distribution properties of "real" random noise. Now, the catch is that over the *long time sequence length,* the *same* "noise" gets put out over and over again. So, short term, you make the signal look like noise. Long term, you make the same thing repeat continuously. You pick the short term to fit the system that needs the noise or a random sequence of inputs. You pick the long term to fit the circuit and your experiment.

For instance, we'll shortly see that if we had a 16-stage digital shift register connected right, we could easily generate a *maximal length* sequence that is 65,535 bits long. Now, take 500, or 1000, or even 5000 bits out of the middle of the sequence and it sure looks like noise. It's an easy matter to convert it to an analog level by low-pass filtering or *integrating* the sequence. Thus, we can go either analog or digital.

The really interesting property of all this is that we can use the same 500, 1000 or 5000 bits over and over again, so that the "noise" *repeats exactly* every time. On a scope display, this means you can get a stable presentation of random events. In electronic music, it means you can get the same melodic or rhythm sequence back anytime you want. With real noise, it would be gone forever. And in security communications and those *correlation* circuits, we have to know what to look for at the other end of the system, so we can generate an exact replica or a delayed replica of what we started with. This process of comparison is called *autocorrelation* and lets us recover our original information.

Types of sequences

Generating long sequences is an easy job for any logic family, particularly MOS or TTL for medium speeds and MECL for faster ones. There are many possible routes to go, but the two most popular ones are called the *Barker* code and the *Maximal Length Pseudo-Random Sequence.* Barker codes are "best" from the standpoint of comparison or autocorrelation. The problem is that they are rather hard to generate and that no one has found any really long ones. So, the maximal length type turns out best for many applications. It's real easy to build, and, as the name implies, we can do no better with a certain amount of parts to attain a given length of sequence.

Maximum length turns out to be *all but one* of the possible states of a suitable serial shift register circuit. Thus, if we have a 6-stage register, the total possible states from 000000, 000001, 000010, through 111110, 111111 is 63. The maximum length pseudo-random sequence will be 63 bits. A 7-stage register is good for 127, an 8 for 255, up to 65,535 for a 16-stage register. Obviously, with a few more register stages, we can get to astronomical lengths. Thus, with only a few low-cost IC's, you can easily build multi-million bit sequences. Even with six stages, the 63-note sequence is more than enough for a short electronic music melodic sequence. Even shorter sequences are useful for timbre generation of unusual electronic sounds.

Table 1 shows the sequence length we can expect from the various length of registers, provided we have set up the circuit properly to generate a maximal length sequence.

Getting the circuit working

But, all a serial shift register can do is *pass on* ones and zeroes. It gets them from its input and passes them on one and only one stage each time the register is clocked. For instance, if we had a register full of zeros, its states would be.

```
0 0 0 0 0 0
```

Connect the input to a "1" and clock it once, and we get

```
1 0 0 0 0 0
```

Clock it again for

```
1 1 0 0 0 0
```

Now, input a zero and clock it

```
0 1 1 0 0 0
```

And clock it five more times:

```
0 0 1 1 0 0
0 0 0 1 1 0
0 0 0 0 1 1
0 0 0 0 0 1
0 0 0 0 0 0
```

Thus, all a serial shift register consists of is a "bit marcher" that shifts things over one stage each clock command. The particular shift register we are using is a serial-in, shift-right type, the most popular and most common form.

We still have to get our input 1's and 0's from somewhere. The way it's done is with *feedback.* We *logically combine* the proper combination of stage *outputs*

SHORTER SEQUENCE LENGTHS

STAGES	LENGTH	STAGES	LENGTH
2	3	10	1023
3	7	11	2047
4	15	12	4095
5	31	13	8191
6	63	14	16,383
7	127	15	32,767
8	255	16	65,535
9	511	17	131,071

TABLE I—RANDOM SEQUENCE LENGTHS

SEQUENTIAL STATES FOR n-6
(sequence length = 63)

000000 (0)	011100 (28)	000111 (7)	001000 (8)
100000 (32)	101110 (46)	100011 (35)	100100 (36)
110000 (48)	010111 (23)	110001 (49)	110010 (50)
111000 (56)	101011 (43)	011000 (24)	011001 (25)
111100 (60)	110101 (53)	101100 (44)	001100 (12)
111110 (62)	011010 (26)	110110 (54)	100110 (38)
011111 (31)	001101 (13)	011011 (27)	010011 (19)
101111 (47)	000110 (6)	101101 (45)	101001 (41)
110111 (55)	000011 (3)	010110 (22)	010100 (20)
111011 (59)	100001 (33)	001011 (11)	101010 (42)
111101 (61)	010000 (16)	100101 (37)	010101 (21)
011110 (30)	101000 (40)	010010 (18)	001010 (10)
001111 (15)	110100 (52)	001001 (9)	000101 (5)
100111 (39)	111010 (58)	000100 (4)	000010 (2)
110011 (51)	011101 (29)	100010 (34)	000001 (1)
111001 (57)	001110 (14)	010001 (17)	(000000) ((0))

TABLE II—THE SEQUENTIAL STATES for the circuit *n*=6 in Fig. 1. The first (input) register stage is on the left, the last (output) on the right. Numbers in parentheses are the decimal equivalents of the binary words.

REGISTER CONNECTIONS FOR LONGER SEQUENCES

STAGES	SEQUENCE LENGTH	FEED EXCLUSIVE-NOR GATE FROM OUTPUTS
17	131,071	14 and 17
18	262,143	11 and 18
20	1,048,575	17 and 20
21	2,097,151	19 and 21
22	4,194,303	21 and 22
23	8,388,607	18 and 23
24	16,766,977	19 and 24
25	33,554,431	22 and 25
26	67,074,001	21 and 26
27	133,693,177	19 and 27
28	268,435,455	25 and 28
29	536,870,911	27 and 29
30	1,073,215,489	23 and 30
31	2,147,483,647	28 and 31

TABLE III—HOW TO CONNECT FOR SEQUENCES UP TO 31

in a proper circuit to generate a *new* one or zero in unique response to the state the register is *now* in.

The logic for maximal length takes nothing but exclusive NOR gates and turns out to be unique. *Any* logical combination of outputs to drive the input will give us *some* sequence length. The problem with the majority of connections is that they only generate a very short (or maybe only a 1-bit!) sequence, and that the states going through the register do not have the random-looking properties that we need.

What we have to do is find the magic combination of logic and feedback that will generate the maximal length sequence for a given stage length. To do this takes a bunch of high-level math, but it has to be done only once. Circuits that will do that are shown in Fig. 1 for register lengths of 2 through 16.

Some more details

Let's actually build a real 63-bit sequence circuit. It's shown in Fig. 2. We use the first six stages of a 74164, one-half of a 7486 quad EXCLUSIVE-OR gate with the two sections cascaded to form an EXCLUSIVE-NOR, the usual 5-volt supply and decoupling, and a variable-speed clock made up from a MC1555 or 555 timer.

Every time the circuit is clocked, it advances one count and generates a pseudo-random sequence of 63 of its 64 possible states. The clock frequency determines how fast the states will change, while the sequence time will be 1/63rd the clock frequency if you run the circuit continuously. We can use the serial bit stream or we can use the digital words that show up in parallel on the register outputs. Or, as we'll shortly see, these are easy to convert to analog "noise" or discrete, randomly varying analog levels.

The two cascaded EXCLUSIVE-OR circuits form an EXCLUSIVE-NOR or comparator. If both inputs are the *same*, a "1" is output. If the inputs are *different*, a "0" is output. Thus our feedback circuit looks at stage 5 and stage 6 to see what they *were* before the new clock arrives. The output of the EXCLUSIVE-NOR then sets up what stage 1 is to *be* after clocking, determined by whether the logic levels on stage 5 and 6 are the same or different.

For instance, in the 63-word sequence of Fig. 1 (n=6), if 5 and 6 are both 0's, a "1" gets entered into the first stage on the next clocking. What was in the first stage goes to the second; the second to the third, and so on. If 5 is a 1 and 6 is a 0, a "0" goes to stage 1 on the next clocking. The same thing happens if 5 is a 0 and 6 is a 1. Finally, if 5 and 6 are both 1's, a 1 is sent to the first stage on the next clocking.

In this manner, the entire pseudo-random sequence is built up. To see how beautifully it works, set up a table like that of Table II for some of the shorter sequences of Fig. 1—say the 15-word sequence of n-3.

All 63 states are shown in Table II. As you can see, any short-term group of bits in the middle jumps around in a very nearly random manner: If you count the number of sequential 1's and 0's you get and work up a distribution curve, it turns out to be a rather chunky approximation to a random probability curve. As we add more and more register stages, the curve smooths out, and the more stages, the better the randomness. For any circuit, the maximum number of *sequential* 1's or 0's we can get has to be equal to or less than the register length. Obviously we get far more short bursts out than long ones. If you go through all the statistical math, you find that you do have very nearly a truly random behavior on a short-term basis, only one that nicely repeats every time we ask it to. In fact, things turn out even better than random noise, since you get the randomness over one sequence length, while "true" noise would theoretically take forever to be truly random. Longer sequences behave even better.

Available sequences

There are usually at least four possible maximal length sequences for any stage length. Circuits to get all four are shown in Fig. 3. If we take the circuit we have and invert the input, we get an inverted sequence in which all the ones are zeros and vice versa. Or, instead of looking at what's going to happen next, we can look to see what *already* did happen and get a backwards sequence. Finally, we can both look backwards and invert to get a backwards sequence with interchanged ones and zeros. All four circuits have essentially the same randomness properties. Which one you use depends on how you like to start the circuit and what output polarity you want. In electronic music, it's handy to use all four, for you can get a sequence, the sequence played backwards, or the sequence played on an inverted scale or on an inverted scale *and* backwards.

The disallowed state

One little detail we have to watch for is the unused state. If you ever get into the 111111 state with the normal circuit, it will stay there forever! The backwards version will also stick in 111111, while either inverted (complementary) sequence sticks in 000000. So, we must never allow this state to occur. It's easy to reset or preset our register or otherwise make sure we also start our sequence on a *known valid* portion of the sequence. By the way, this is true of almost all counters and sequential circuits in general. You have to investigate all the disallowed or "don't care" states to make sure none of them are self-perpetuating.

One other little detail is that we obviously must get an additional 1 or 0 (or one less zero or one) since our code is always an odd number of bits long. This missing 1 or 0 will tend to skew the random distribution slightly and will tend to shift the bias on an analog conversion scheme slightly. This is easily avoided by compensating bias resistors, and the longer sequences have almost negligible skew and randomness bias.

Converting to analog

Figure 4 shows two different ways to convert the digits to numbers. In 4-a, we use an integrator or low-pass filter on the serial bit stream, and the output of the integrator is an analog voltage that varies in a random manner. The short-term output is noise that behaves just like white noise up to the filter's cutoff frequency or at least up to a good fraction of it. A cutoff frequency of 1/20 the clock frequency is recommended, particularly for longer sequences. A different possibility is shown in Fig. 4-b.

Here we directly D/A convert the parallel digital words to an analog signal. With this circuit, you get analog levels that jump to some new random value, once every clock cycle.

To find the time the sequence repeats, just divide the clock frequency by the sequence, the repeat time will be 100,- using a 100-kHz clock and a length 16 sequence, the repeat time will be 100,-000/65,535 or about 1.5 times per second. If you are using the 63-note sequence for electronic music at 3 notes per second, it repeats once every 21 seconds. A 127-note sequence would be good for 42 seconds, and so on. As an extreme example, if you used a 48-stage MECL shift register and a 100-MHz clock, it would take around 3 million seconds or over half a year to repeat. At lower clock frequencies, it would take decades or even centuries!

The serial conversion circuit of Fig. 4-a works best when the clock is at least 20 times the filter's cutoff frequency as determined by the capacitor values. Thus, for high-quality audio testing and requirements of this type, you use as long a register length and as fast a clock as you can.

Applications

Let's briefy turn to the things you can do with a pseudo-random sequence generator.

For audio testing and communications, you normally use a very fast clock and a sequence that repeats perhaps 30 times a second, so you can get a stable oscilloscope display. The filtering then gives you random signal variations that duplicate the effect of random combinations of voice or communications data or signal levels. Commercial instruments are available (Hewlett-Packard, among others) that do just this. The net result of the testing is that you simulate the real operating conditions in a realistic man-

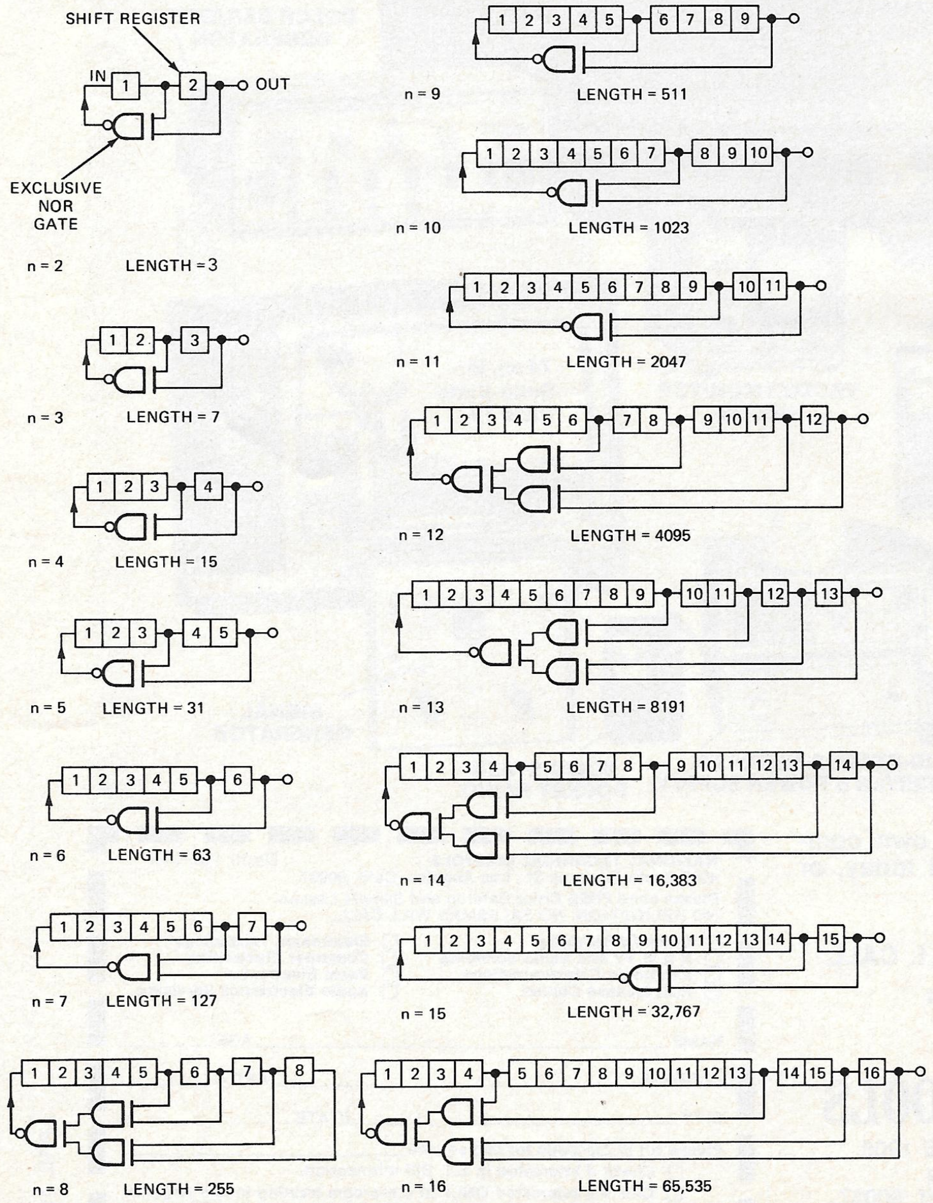
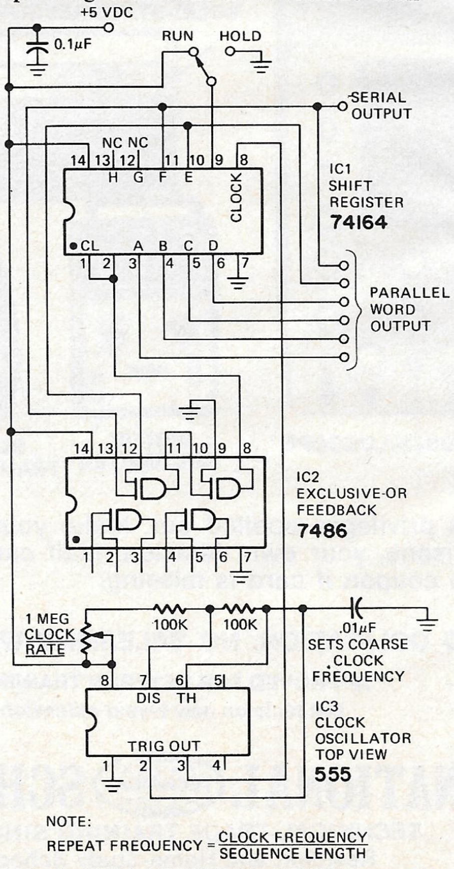
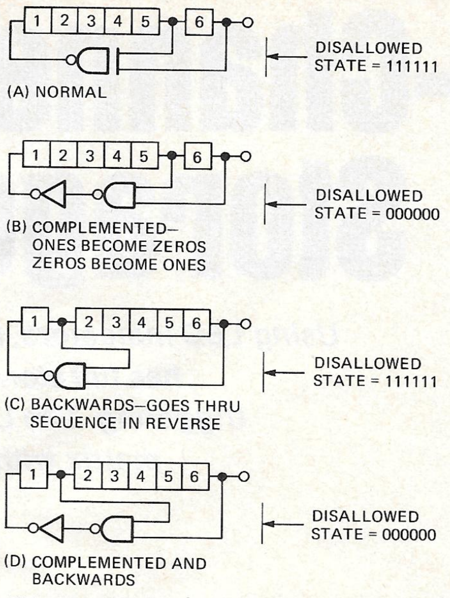


FIG. 1—PSEUDO-RANDOM CIRCUITS that produce sequences from 3 to 65,535 words.



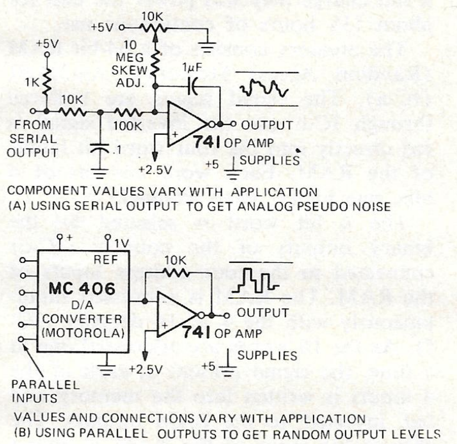
+5 VDC
0.1µF
RUN
HOLD
SERIAL OUTPUT
NC NC
H G F E
CLOCK
CL A B C D
IC1 SHIFT REGISTER 74164
PARALLEL WORD OUTPUT
IC2 EXCLUSIVE-OR FEEDBACK 7486
1 MEG CLOCK RATE
100K
100K
.01µF SETS COARSE CLOCK FREQUENCY
DIS TH
TRIG OUT
IC3 CLOCK OSCILLATOR TOP VIEW 555
NOTE:
REPEAT FREQUENCY = CLOCK FREQUENCY / SEQUENCE LENGTH

FIG. 2—THIS LENGTH-63 CIRCUIT uses 6 stages (*n*=6).



DISALLOWED STATE = 111111
(A) NORMAL
DISALLOWED STATE = 000000
(B) COMPLEMENTED– ONES BECOME ZEROS ZEROS BECOME ONES
DISALLOWED STATE = 111111
(C) BACKWARDS–GOES THRU SEQUENCE IN REVERSE
DISALLOWED STATE = 000000
(D) COMPLEMENTED AND BACKWARDS

FIG. 3 — FOUR POSSIBLE SEQUENCES for any *n* can be made by rearranging the circuit.



10K
10 MEG SKEW ADJ.
1K
10K
100K
FROM SERIAL OUTPUT
OUTPUT
741 OP AMP
SUPPLIES
COMPONENT VALUES VARY WITH APPLICATION
(A) USING SERIAL OUTPUT TO GET ANALOG PSEUDO NOISE
MC 406 D/A CONVERTER (MOTOROLA)
REF
PARALLEL INPUTS
OUTPUT
741 OP AMP
SUPPLIES
VALUES AND CONNECTIONS VARY WITH APPLICATION
(B) USING PARALLEL OUTPUTS TO GET RANDOM OUTPUT LEVELS

FIG. 4—ANALOG OUTPUT CIRCUITS.

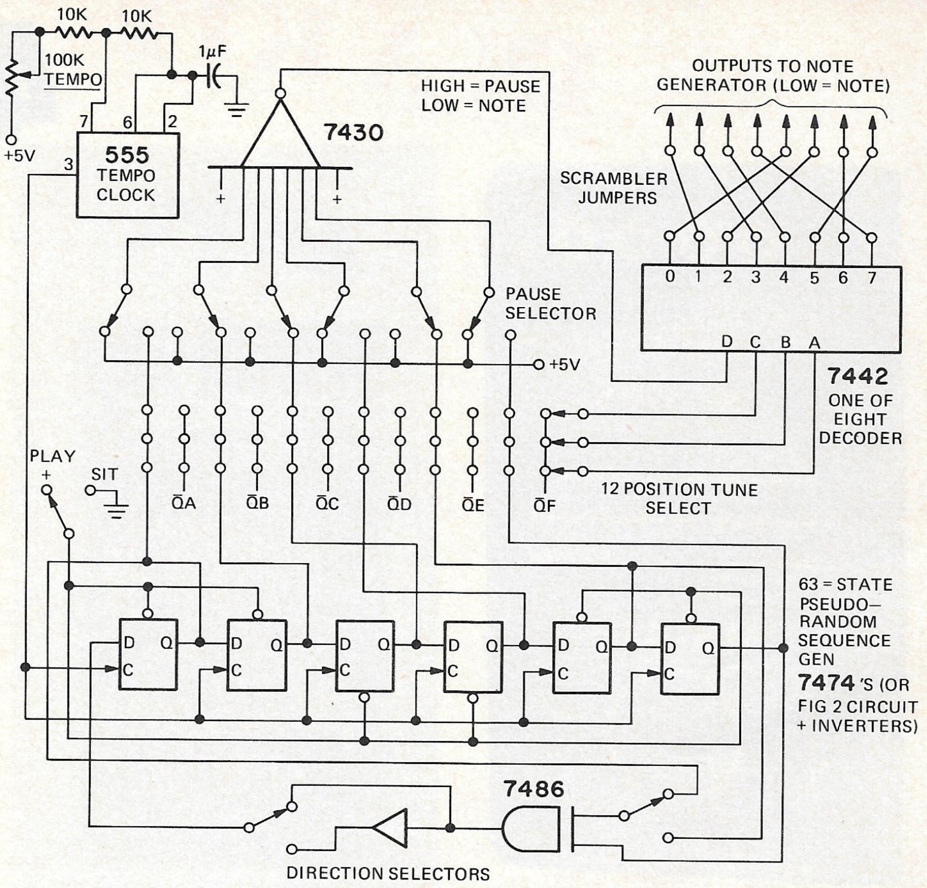


FIG. 5—POSSIBLE MUSIC COMPOSER.

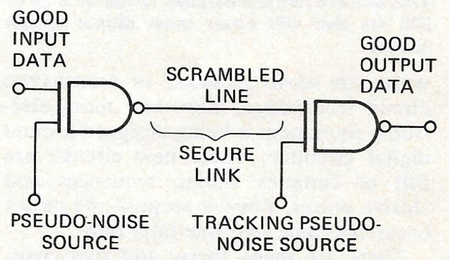


FIG. 6—A PSEUDO-RANDOM CODER or data scrambler for privacy or cryptography.

ner but also in a way that lets you see the results as a stable display.

Electronic music uses are relatively obvious. By interchanging the outputs in a programmable manner, you can use one basic sequence generator to build a fantastic number of tunes and can obtain them frontwards, backwards, normal scale or inverted. Some of these combinations will be dull and others will be simply phase-shifted replicas of others, but the number of unique and interesting variations remaining are still a bunch. Figure 5 shows one possible electronic music composer. By adding random rhythm and pause combinations, you can end up with an essentially infinite number of variations. You can also use pseudo-random sequences to generate timbre waveforms for electronic music.

Secure computer communications encode the data to be transmitted onto a pseudo-random sequence that is locked to a replica at the far end. Usually, the sequence length is very long, days, months, even years. Cryptography and other security schemes are other applications of this type. As with *any* code, regardless of its complexity, it *can* be broken. The object of any code game is to make cracking the code so complex, make it take so long, or make it so expensive that the cost of cracking exceeds or at least severely diminishes the value of the information to be gained. So, as with all codes, the pseudo-random technique is a reasonably effective deterrent, not a fail-safe and foolproof route to security.

To encode or decode data, two EXCLUSIVE-NOR gates are used, one at each end of our secure line. Remember that the output of an EXCLUSIVE-NOR is the same if the inputs are identical and different if they differ. So, if our pseudo-random generator happens to be in a "1" state, input data 1's stay ones and 0's stay zeros, e.g. they are transmitted without "error." On the other hand, if our pseudo-random generator happens to be in a "0" state, the 0's become 1's and 1's become 0's; we say the data is *complemented*. Since the line now consists of a random mixture of good and bad data, it appears to be garbage to anyone monitoring in the middle. At the other end, we simply add a new pseudo-random generator *identical in length and sequence* to the original; once again, it inverts when zero and passes when one; and all the data straightens back out again. Figure 6 shows the circuit.

Autocorrelation is a very complex subject, but it dramatically illustrates the power of pseudo-random sequences. Suppose we have a sequence length of 63 and that a 1 is +1 volt and 0 is −1 volt. If we multiply the code by itself on a bit-by-bit basis, we would get +63 volts out. On the other hand, if we multiply the code by a delayed replica of itself or a random string of ones and zeros, we will probably get a very low value, maybe +1 or −1 out. Thus when the code matches itself, you get a very strong output signal; otherwise you get very little. Only the Barker codes can give you perfect + and −1 *sidelobe* levels; the mismatch and noise level produced in a pseudo-random code is higher, but still has a very useful sidelobe level.

This tremendous build-up of signal buys you a signal-to-noise improvement and the ability to extract a signal deeply buried in uncontrollable noise.

Longer sequences

The schematics for sequences longer than 16 get rather cumbersome to draw, so they are shown in table form in Table III. The lengths are shown up to 31 stages, which is a sequence length of 2,147,483,647. That should be long enough for just about anything. Note that this sequence can be built up with only four of the 74164 shift registers. Sequence 19 is omitted because it takes more than one exclusive OR gate to build it. There are likewise *other* possible maximally long or nearly maximal sequences for lengths 17, 21, 22, 23, 25, 27, 28, 29 and 31, but one should be enough for each length. **R-E**