The following article describes a program for creating Pink Music. Since the music sounds pleasant due to psychological implications, I can't think of a more fitting name than:

PINK FREUD

An Article by
Bob Yannes and
Steve Witham

If you're like me, the sound of random music (such as that produced by passing white noise through a sample/hold) is profoundly uninteresting --even annoying! To me, this type of random music epitomizes the inhuman, mechanical sounds that synthesizers are always being accused of making. You can imagine, then, how I felt when my friend wanted to demonstrate his latest random music program (written for a POLY 88 using his own four-voice music program, TONEGEN, which you may have read about in P.C.C. or Dr. Dobb's). Imagine my surprise when the resultant music was not only tolerable, but even pleasant to listen to! The program, it turned out, was creating what we'll call "pink" music, which, for obscure reasons sounds more like "real" music to most people.

I immediately insisted that we modify the program to work on a 6502 processor, specifically the PAIA 8700. (Fanfare please....) here then is a computer program which will operate on a standard configuration, 8700-driven, polyphonic synthesizer (one QuASH required) and produce four independant, pink voices.

The basis for this program was Martin Gardner's Mathematical Games column in the April 1978 Scientific American. The article, "White and Brown Music, Fractal Curves and One-over-f Fluctuations", describes in detail the various random music relationships and I refer the interested reader to this article. The reasons for pink music (or 1/f music as it is referred to in the article) sounding pleasant are subjective, but essentially the key factor is the amount of correlation between successive random notes. Music based on white noise has no correlation between notes; that is, successive notes can occur anywhere throughout their range with no predictability from one note to the next. This tends to sound too random to be music. At the other extreme, music based on Brownian noise has a high degree of correlation, meaning there is little change from one note to the next. This tends to sound too boring to be music. There are a lot of possibilities between these extremes, however it has been subjectively determined that pink music is the most pleasant to listeners (conversely, most non-listeners prefer silence, but a computer program for producing silence isn't very interesting).

The algorithm for generating pink numbers corresponding to the random frequencies is also given in the Scientific American article and we have adapted it here for computer. The program makes use of a random number generator routine which produces numbers from 0 to 255 (decimal). This routine has appeared in a number of microcomputer magazines (primarily for the 8080 and 8008) and has been previously documented, so we won't cover it here. If you want to impress your friends, you could refer to the pink number generating algorithm as a digital filter as, in a sense, it "filters" the white numbers produced by the random number generator into pink numbers (in the analog world, we would actually use a filter on white noise to produce the desired result). It must be realized, though, that this "filter" only acts like a filter when converting white numbers into pink numbers. More accurately, the algorithm achieves the same results as a digital filter, but is much more efficient for this particular application.

In human terms, the algorithm works like this:
We have 8 bits to work with on a microcomputer and with these 8 bits we can produce $2^8 = 256$ unique numbers, these numbers being 0 to 255 sequentially. Whichever one of these 8-bit numbers we are looking at currently is called the key and associated with each of the 8 bits of the key is a random number (that is, there is one random white number associated with each of the 8 bits in the key for a total of 8 random numbers). The sum of the 8 random numbers associated with the key is the pink number associated with that key (the current pink number is always generated from the current key). There will be a certain pink number associated with each of the 256 values that the key can take on. In order to generate the next pink number, we increment the key by one. The important part of the algorithm concerns how the random numbers associated with the key change when going from one key value to the next (obviously, if the random numbers change, their sum, the new pink number, will also change). When the value of the key is incremented to the next value, some of the bits of the new key value will be different from the bits of the previous value. For every bit that changed, we generate a new random number associated with that bit and replace the old random number. The random numbers associated with bits that didn't change are left the same. Summing the 8 random numbers produces the new pink number.

Let's clarify that with an example:
Suppose the value currently in the key is 00000001 (= 1 base 10). There is a random number associated with each of the eight bits of this number which, when added, produce the pink number for that key value. To generate the next pink number, we increment the key value by one, giving us 0000010 (= 2 base 10). In this case, both bit 1 (the LSB) and bit 2 have changed from their previous values, while bits 3 through 8 have remained the same. Following the rules of the algorithm, we generate a new random number for bit 1 (replacing the old one) and a new random number for bit 2 (ditto). Since bits 3-8 didn't change (they all remained zero), the previous random numbers associated with these bits are left unchanged. We now add up the 6 old random numbers (from bits 3-8) and the 2 new random numbers (from bits 1 and 2) and the result is the new pink number (the pink number associated with the key value of 2). Every time we want to generate a new pink number, we increment the current key value, generate new random numbers for each key bit that changed and add up the resulting random numbers.

Eventually, the key value will reach 11111111 (255 base 10) and when incremented, the key value will go to 00000000. Since every bit changes, we generate a new set of 8 random numbers and start over again. It should be noted that, since we are starting at zero with an entirely new set of random numbers, the sequence does not repeat and we will generate an entirely new set of 256 pink numbers. This loop continues indefinitely, generating non-repeating sets of pink numbers ad infinitum.

Although the program described so far will generate pink music, a few modifications are in order to obtain the optimum musical effect. Pink music sounds fine when played monophonically, but when played polyphonically, the combination of pink voices will usually be discordant--hardly a pleasant quality! In order to overcome this, each pink note is "quantized" to a note in a particular scale which contains a minimum of discordant notes. To accomplish this, the pink numbers generated are not used directly in setting the final pitch, but are instead used as a pointer for a scale look-up table. This table contains the notes which sound "harmonious" when played together and it is these notes which determine the pitch. The scale table and its length are variable (3 examples will be given later) and notes can be entered more than once in the table to increase their probability of occurence.

A problem you may have noticed is that due to the limited range of the DAC and the limited size of the scale look-up table, some restrictions must be placed on the random white numbers that go into making up the pink numbers. After all, the random number routine is capable of producing white numbers up to 255 decimal (FF Hex) and we must add eight of these numbers together to get a pink number. Obviously this sum can easily exceed the range of the DAC (00 to 3F Hex), in fact, an overflow of the accumulator is not unlikely! Further restrictions are placed on the final pink number by the length of the scale table. Within the range of the DAC (5 octaves) there is a limited number of notes in each scale. The number varies from scale to scale, but we can adjust the length of the table accordingly. In order to ensure that a pink number will be within the desired range, the sum of the eight random numbers for that pink number must be in that range. In other words, if there are N notes in the table, each random number can be no larger than N/8. No special tricks are used to limit the random numbers--first, the 5 high order bits are truncated, insuring that none of the random numbers will be greater than 7, then each number is compared to see if it is greater than the limit, in which case, we go get another until we find one that is less than or equal to the limit.

The rhythm of the music is also something to be considered. The program plays a note, holding it for a fixed period of time, releases the note (allowing it to decay) and then waits another fixed period before playing the next note. Even when playing monophonically this sounds mechanical, but when played polyphonically, it sounds like "clockwork chords". In order to avoid having the music sounding sequenced, the rhythm must be varied during playing time. There are a number of ways to accomplish this--for example, the rhythm could vary randomly. We chose to compare the new note with the previous note and, if the same, the old note is sustained (rather than decaying) and blended into the new note. The result is that the note is sustained over two (or more) periods. Since there is a good deal of correlation between successive pink notes, it is not unlikely for the next note to be the same as the last. The resultant rhythm is especially effective when playing polyphonically, yielding complex contrapuntal patterns. One consequence of this technique is that we must know the next note while playing the present note (in order to determine whether the present note should be sustained or not). This means that the notes being played are always one step behind the notes being generated.

Let's get an overview of how the program works:
Since the program is written for four voices, four pink numbers must be generated for each key value. Each pink number for a voice is produced using the previously described algorithm (applying the limit constraint and quantization) and temporarily stored. The program then returns to generate the pink number for the next voice, continuing until it has produced four pink numbers for four voices. After this new set of pink numbers is produced, the previous set of pink numbers is played by sending them to the DAC and their appropriate QuASH channel, after first switching on their gate bits. These notes are held on for a set period of time after which each number is compared to the number it is to be replaced by (the new set of pink numbers). If the number currently being played is the same as the next number to be played, the gate bit of the current number is left on, causing the note to sustain. If the notes are not the same, the gate bit is cleared, allowing the present note to decay. Finally, there is a fixed delay before the new set of notes is played. If a new note is the same as the previous note, the previous note will have been sustained for this delay, and the transistion between the two notes will be seamless, sounding like one long note rather than two. The entire process explained in this paragraph will continue indefinitely until the computer is reset, unplugged or smashed with a hammer (too much pleasant music can get on your nerves!).

One consequence of using this program polyphonically is that quite a bit of memory is needed for data storage. Each voice must maintain its own set of 8 random numbers as well as the present note and the next note. We can't just write over top of a previous voice (say voice #1) with the new voice (voice #2) as we must maintain voice #1's random numbers in order to generate the new voice #1 the next time around.

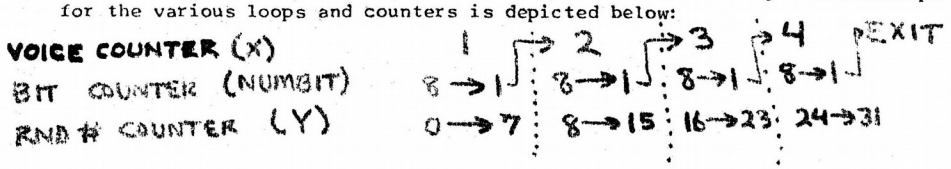
Now that you have an idea of what the program does, let's take a closer look. First, I'd like to clarify a few conventions I'll be using. I'm going to refer to addresses in the assembly listing as line numbers; for example, "Line 6" means ADDR 0006, "line 4D" means ADDR 004D, etc. Also, in the COMMENT column of the assembly listing, I use the symbol "#" not to mean "immediate mode" as it does in the INSTRUCTION column, but to mean "number", which is the more common interpretation anyhow. Okay? Let's go...

Line 0 is the breakpoint trap. Most will realize that in normal use it has no effect, but if you have to debug the program, it allows you to enter the breakpoint vector at 0000 (4C C0 FF) without wiping out the first 3 bytes of the program. When not being used for debugging, it allows you to run the program from 0000. If you left the break vector at 0000 permanently, you'd constantly find yourself forgetting it was there and trying to ~~start~~ run the program at 0000, in which case, even the pink music might not be able to soothe you. Lines 3-6 make sure that everything works the way it's supposed to--the program stack is set to 01FF to place it as far away as possible from the random numbers which are also stored on page 1 (from 0100 to 011F) and the decimal mode is cleared as the 6502 has a rather nasty quirk--you can't test for zero with the Z flag in decimal mode which screws up BNEs and BEQs. (WE WANT ALL ADDITIONS IN HEX ANYWAY) Line 7 initializes the KEY by loading it with $FF (which came from line 3). We start KEY at FF so that when the KEY is incremented to 00, all the bits change and an entire set of 8 new random numbers is generated and stored over top of the garbage that was there before running the program. Since KEY is incremented in line E, the program always starts off with a new set of 8 random numbers and there's never any chance that garbage will be added up, producing an out-of-range number.

The main body of the program is KEYLOOP which extends from line 9 to line 59. KEYLOOP also calls two subroutines, PLAY and RNDGEN. Generation of the pink numbers is handled by VOXLOOP which extends from line 18 to line 41. Within VOXLOOP the KEY bits are tested and the random numbers are generated. There are 8 bits in the KEY that have to be tested, one at a time, this is handled by RNDLOOP. RNDLOOP also keeps a running sum of the random numbers which, after all 8 are summed, is the pink number.

The operation of VOXLOOP is fairly straightforward, but there's a lot of nested stuff to keep track of--I'll get to that in a minute. In line 10, we found which bits changed between the previous KEY and the new KEY by EXORing the two numbers. This leaves us with an 8 bit pattern containing ones where the bits were different (ie, where a bit changed from one key to the next) and zeroes where the bits were the same (ie they didn't change). For example, if the previous value of KEY was 00001101 and the new value was 00001110, the result of the EXOR would be 00000011. This is, for lack of a better name, what will be referred to as the test pattern. The program must now examine this 8-bit pattern one bit at a time and see if a new random number should

be generated for that bit (if the bit is a one) or if the old random number should be kept (if the bit is a zero). The testing is done by rotating the pattern one bit to the left (in line 22) which puts the leftmost bit of the pattern into the CARRY where it can be easily tested with a BCC (line 24). If clear, we don't want a new random number so the program skips to ADDEM and adds the old random number for that bit to the running sum (called TOTAL). If the CARRY was set, we get a new random number from subroutine RNDGEN (line 26), lines 29-2F make sure it's not too big, and the new number is stored over top of the previous random number for the bit being tested. Line 32 (ADDEM) will then add this new random number to the running sum (TOTAL). RNDLOOP must be executed 8 times to test all eight bits of the pattern (by rotating each bit into the CARRY) and when finished, the running sum will be the pink number for the given voice. Since there are four voices, VOXLOOP must be executed four times (with RNDLOOP being executed 8 times for each voice) using the same test pattern, which was preserved through all that rotating by storing it in DELTA. RNDLOOP is a nested loop I was talking about and there are a couple of counters in it which could be confusing. The first is the random number counter (the Y Index Register) and the second is the test pattern bit counter, NUMBIT. NUMBIT is the counter which keeps track of which pattern bit is being looked at and determines whether the voice we're working on is done yet. A voice is finished when we have tested all 8 bits of the pattern, hence NUMBIT counts down from 8 until it reaches zero, at which point we know we are done with that voice. When the voice is finished we go back and repeat the entire VOXLOOP procedure until it's been done once for each voice (in this case, a total of four times). Therefore, NUMBIT counts from 8 to 0 four times for each KEY value. The random numbers associated with the bits of the test pattern (these bits are counted by NUMBIT) pose a slight problem. Each random number for each voice must be maintained separately--we can't write over top the set of 8 numbers for the previous voice with the new set for the present voice. Since there are four voices, the program must keep separate 4 sets of 8 random numbers, that is, 32 numbers in all. The random numbers are counted by the Y Index Register. The main difference between the Y counter and NUMBIT is that the Y counter is not re-initialzed each time VOXLOOP is executed as is NUMBIT. Y, therefore, counts from 0 to 31, 8 counts per voice. Y would keep on counting too, but as soon all four voices are finished, we leave VOXLOOP. The counting relationships for the various loops and counters is depicted below:



The Y counter is used as a memory pointer in line 2F so that the random numbers are stored as a sequential list of 32 numbers from 0100 to 011F. The X (voice) counter is also used as a memory pointer in lines 36-38 to point to a separate running sum for each voice.

The results of executing the program to line 43 are:

A.) 4 sets of 8 random numbers, 1 set per voice, which were used in producing the present voice and will be used in generating the next voice.

B.) 4 pink numbers in TOTAL, 1 per voice.

Now that we have four pink numbers, we want to play them. Well, almost. Remember that the numbers to be played are always one step behind the numbers being generated. In this case, the numbers to be played are stored in a location called OLD and the new numbers are in TOTAL. We haven't talked about OLD yet, because we haven't stored anything there up to this point. We will be storing the note to be played there, but the first time KEYLOOP is executed, OLD will be full of garbage and that means the first note played for each voice will be garbage. This is okay, though, it only happens once out of an infinite sequence of notes and the only time it's noticable is after power up when OLD may contain really odd garbage. If you RESET and start the program again, OLD will contain a leftover set of quantized pink numbers from the previous run which won't sound like garbage.

In order to play the notes in OLD, we use subroutine PLAY twice. The first time through, we play all notes by forcing their gate and glide bits on and hold them on for a length of time determined by a counter called TEMPO. The second time through, we leave the gate and glide on for only the notes that are to be sustained and all others are turned off, allowing them to decay. Before getting the next set of numbers, we again delay for an amount of time determined by TEMPO. PLAY is first jumped to in line 45. PLAY uses an argument supplied by the main program and held in the Accumulator and it is this argument which tells whether to allow voices to decay or not. The argument supplied by line 45 is $C0 (which was loaded in line 43). The pattern $C0 looks like 11000000 in binary and, if this pattern represents a note to be sent to the DAC, this note would have both its gate and glide turned on. $C0 is stored in NOTESTAT in PLAY--NOTESTAT is the status of the note being played (ie whether gate and glide are to be on). Lines 63-67 use the new pink number as a pointer to point to a number in SCALE, which becomes the quantized number for the next note, and compares this to the note to be played. If the old note and the new note are equal, line 6C forces the gate and glide on for the note to be played and line 6E has no effect. If the notes aren't the same, we skip right to line 6E which ORs the note to be played with NOTESTAT. Since this is the first time through PLAY, NOTESTAT is still $C0, which is the same as ONBITS, the number we use to force the gate and glide on no matter what. The result is that when PLAY is executed the first time, it doesn't matter what the value of the note is, it will always be played (ie the gate will always be turned on). This is exactly what we want since the first time PLAY is executed it means we have a new set of notes to play and none of them are ready to decay yet (how can a note decay if it hasn't been played yet?). Lines 70-79 actually play the note by sending it to the DAC/QuASH and allowing the voltage to settle. This procedure, called UPDATE, is executed four times, once for each voice and finally, the entire TEMPLOOP procedure is executed $20 times to create the tempo delay while continuously keeping the QuASH updated.

After all notes have been played for the fixed TEMPO delay, control is passed back to the main program, line 48. Here, the argument to be passed to PLAY is loaded as 00, then the program jumps to PLAY again. The procedure is the same as before, if the old note and the new note are the same, gate and glide are forced on by line 6C, however, if the notes are not the same, line 6E leaves the gate and glide off (as NOTESTAT is 00). When the notes are played, the results are that if the two notes were the same, the note being played is sustained so it can blend into the next note of the same value. If the notes were different, the note being played is allowed to decay.

by turning off its gate. Again, the entire procedure is repeated to update all four voices and create a tempo delay.

After returning from PLAY for the second time, we are now ready to store something in OLD and that something is the quantized pink number that we just compared with the note being played in PLAY. lines 4F-54 get the unquantized pink number out of TOTAL, use it to point to a number in SCALE and store the resulting quantized number in OLD. This is repeated in order to store all four voices. Following this, line 59 sends us back to the very beginning (KEYLOOP) where we get the next KEY value, go through the whole mess again and play the numbers that we stored in OLD before jumping back to get the new KEY. And so it goes.

That's how the program works, but there are a number of things you have under your control to tailor the music generated to your liking. For example, the tempo of the music is set by ADDR 005E and you're free to change it. Try changing ADDR 005E to 30 for a less bubbly sort of music. If you're into environment effects, you can slow it down a lot and use slow attack and decay rates on the synthesizer, or for special effects, you can make it really fast.* Just remember, the maximum amount of tempo delay you can get is 255 delays by entering 00 at ADDR 005E and the shortest amount of delay (zero) is arrived at by entering 01. This may seem a bit odd, but COUNTEMP, the tempo counter, is decremented before it is tested for zero, therefore, if TEMPO =0, TEMPO-1 =255. Likewise, if TEMPO =01, TEMPO-1 =0.
The other music characteristic you have directly under your control is the SCALE, which is the most important part of the music. If you don't believe me, try running the program without entering a scale and see how awful it sounds. You're free to enter any scale you want and there are a vast number of scales you can choose. For example, the scale need not cover the entire range of the DAC, you can change the LIMIT at 002C to operate over, say, a one octave range or repeat certain notes in the scale so that they will occur more often. There are three scales which we give as an example. The first is the ever popular pentatonic scale in which any combination of notes sounds harmonious. The second is a C-Major scale and the third is a C-Minor scale which is good for making people feel uneasy or depressed.

Almost as important as the notes being generated is how you apply these voltages. If you have four complete synthesizer voices, this is ideal. I recommend that you set up a nice bass patch on one, tune the next voice an octave higher and set up a mellow mid-range patch, tune the next one another octave higher and set up a shimmery high-range patch such as violins, etc. The fourth voice could be tuned an octave higher still, but would probably be a bit too high, so better to use it to fill in and enrich another one of the voices. The effect will not be as good if you only have a monophonic synthesizer and try to drive four oscillators, since the independent rhythms will be lost (although the voices will continue to change according to their own rhythm, you can only use one note to control the ADSR and VCA.). As usual, though, experiment! Try controlling filters or volume or anything else you can find--you've got four outputs, you might as well use them and if any of you come up with something interesting, write to POLYPHONY! One other thing, remember the glide is also activated so you can get any glide you want by turning up the glide knobs.

* The tempo can be slowed even further by increasing the delay at ADDR 0077

SCALE TABLES:

Load one of these tables starting at ADDR $00BD.

Pentatonic	C-Major	C-Minor
00	04	04
02	06	06
05	08	07
07	09	09
0A	0B	0B
0A	0D	0C
0C	0F	0F
0E	10	10
11	10	10
13	12	12
16	14	13
16	15	15
18	17	17
1A	19	18
1D	1B	1B
1F	1C	1C
22	1C	1C
22	1E	1E
24	20	1F
26	21	21
29	23	23
2B	25	24
2E	27	27
2E	28	28
30	28	28
32	2A	2A
35	2C	2B
37	2D	2D
3A	2F	2F
3A	31	30
3C	33	33
3E	34	34

Addendum: There are a lot of things that you can experiment with in this program. You could load all three tables and have the program change scales randomly between them using indexed indirect addressing. There is a lot of room for improving the rhythm control--having voices operate at different speeds, etc. Our plans for super pink music are beginning to take shape now and we would be greatly interested in any neat tricks that you readers may come up with.

Also, due to the pseudorandom nature of the random number generator, if the RAM locations for BYT1 to BYT4 come up with the same garbage on power-up each time, then the first time you run it after power-up, you will always get the same sequence. If this bothers you, you can load your own "random numbers" in locations 00AC-00AF before running it the first time, or you can run the program for a few notes, then RESET and start it again.

Finally, although I said that the random number generator (RNDGEN) is totally independent of the program and can be used separately, you must realize that, like the rest of the program, it is written for zero page addressing, therefore whatever program you use RNDGEN with, you must either rewrite RNDGEN for absolute addressing, or make sure there is room on page zero.

Oops, one more. If you want to write your own scales, see PAIA's table showing the hex equivalents for the note numbers (QuASH manual) at the end of this document.

Pink Music

A program for the PAIA 8700
by Bob Yannes and Steve Witham

```
*= $0000

ADDR  CODE       LABEL     INSTRUCTION      COMMENTS

0000  4C 03 00   BPOINT    JMP PINK         BREAKPOINT TRAP.
0003  A2 FF      PINK      LDX #$FF         INITIALIZE
0005  9A                   TXS              STACK POINTER.
0006  D8                   CLD              SET HEX MODE.
0007  86 B1                STX KEY          INITIALIZE KEY.
0009  A5 B1      KEYLOOP   LDA KEY          GET KEY...
000B  8D 20 08             STA DISP         AND SHOW IT.
000E  E6 B1                INC KEY          NEXT KEY.
0010  45 B1                EOR KEY          GET PATTERN OF CHANGED KEY BITS...
0012  85 B2                STA DELTA        AND STORE IT.
0014  A2 04                LDX #VOXNUM      GET # OF VOICES.
0016  A0 00                LDY #0           INITIALIZE RND # COUNTER.
0018  A9 00      VOXLOOP   LDA #0           CLEAR THE
001A  95 B4                STA TOTAL-1,X    RUNNING SUM FOR VOICE #X.
001C  A9 08                LDA #8           INITIALIZE BIT COUNTER...
001E  85 B3                STA NUMBIT       AND STORE IT.
0020  A5 B2                LDA DELTA        GET CHANGED BIT PATTERN.
0022  2A        RNDLOOP   ROL A            PREPARE TO TEST LEFTMOST BIT.
0023  48                   PHA              SAVE SHIFTED PATTERN.
0024  90 0C                BCC ADDEM        BIT DIDN'T CHANGE--KEEP OLD RND #.
0026  20 83 00   PICK      JSR RNDGEN       BIT DID CHANGE--GET NEW RND #.
0029  29 07                AND #7           TRUNCATE IT.
002B  C9 05                CMP #LIMIT+1     # OUT OF RANGE?
002D  B0 F7                BCS PICK         YES, GO GET ANOTHER.
002F  99 00 01             STA RANDOM,Y     NO, STORE IT IN LOCATION Y.
0032  B9 00 01   ADDEM     LDA RANDOM,Y     GET THE RANDOM # FROM LOCATION Y.
0035  18                   CLC              PREPARE TO ADD.
0036  75 B4                ADC TOTAL-1,X    ADD IT TO THE RUNNING SUM FOR VOICE #
0038  95 B4                STA TOTAL-1,X    STORE THE RUNNING SUM (PINK #) FOR VO
003A  C8                   INY              NEXT RND #.
003B  68                   PLA              GET SHIFTED PATTERN BACK.
003C  C6 B3                DEC NUMBIT       NEXT BIT.
003E  D0 E2                BNE RNDLOOP      DONE ALL BITS?  IF NO, GO DO NEXT.
0040  CA                   DEX              NEXT VOICE.
0041  D0 D5                BNE VOXLOOP      DONE ALL VOICES?  IF NO, GO DO NEXT.
0043  A9 C0                LDA #ONBITS      PLAY NOTES WITH
0045  20 5B 00             JSR PLAY         GATE & GLIDE FORCED ON.
0048  A9 00                LDA #0           PLAY GATE & GLIDE ONLY
004A  20 5B 00             JSR PLAY         ON SUSTAINED NOTES.
004D  A2 04                LDX #VOXNUM      GET # OF VOICES.
004F  B4 B4      SCALOOP   LDY TOTAL-1,X    GET PINK # FOR VOICE #X.
0051  B9 BD 00             LDA SCALE,Y      USE IT AS POINTER FOR SCALE TABLE.
0054  95 B8                STA OLD-1,X      STORE QUANTIZED NOTE FOR VOICE #X.
0056  CA                   DEX              NEXT VOICE.
0057  D0 F6                BNE SCALOOP      DONE ALL VOICES?  IF NO, GO DO NEXT.
0059  F0 AE                BEQ KEYLOOP      BRANCH ALWAYS.

005B  85 B4      PLAY      STA NOTESTAT     STORE GATE & GLIDE STATUS.
005D  A9 20                LDA #TEMPO       INITIALIZE
005F  85 DD                STA COUNTEMP     TEMPO COUNTER.
0061  A2 04      TEMPLOOP  LDX #VOXNUM      GET # OF VOICES.
0063  B5 B8      UPDATE    LDA OLD-1,X      GET NOTE TO BE PLAYED.(OLD NOTE) VOICE
0065  B4 B4                LDY TOTAL-1,X    GET NEW PINK # (POINTER) FOR VOICE #X.
0067  D9 BD 00             CMP SCALE,Y      OLD NOTE = NEW NOTE?
006A  D0 02                BNE CHOP         NO, GO CLEAR.
006C  09 C0                ORA #ONBITS      YES, FORCE GATE & GLIDE ON.
006E  05 B4      CHOP      ORA NOTESTAT     CLEAR GATE & GLIDE UNLESS FORCED.
0070  8D 00 09             STA DAC          LET DAC SETTLE...
0073  9D FB 09             STA QUASH-1,X    SEND IT TO QUASH CHANNEL #X...
0076  A0 25                LDY #$25         AND DELAY
0078  88        SETTLE    DEY              UNTIL QUASH
0079  D0 FD                BNE SETTLE       SETTLES.
007B  CA                   DEX              NEXT VOICE.
007C  D0 E5                BNE UPDATE       DONE ALL VOICES?  IF NO, GO DO NEXT.
007E  C6 DD                DEC COUNTEMP     1 LESS TEMPO DELAY.
0080  D0 DF                BNE TEMPLOOP     DONE TEMPO DELAY?  IF NO, DO IT ALL AG
0082  60                   RTS              RETURN

0083  98        RNDGEN    TYA              SAVE
0084  48                   PHA              Y.
0085  A4 AC                LDY BYT1
0087  A5 AD                LDA BYT2
0089  84 AD                STY BYT2         BYT2=OLD BYT1.
008B  A4 AE                LDY BYT3
008D  85 AE                STA BYT3         BYT3=OLD BYT2.
008F  A5 AF                LDA BYT4         ACCUM.=OLD BYT4.
0091  84 AF                STY BYT4         BYT4=OLD BYT3.
0093  84 B0                STY SHREG        SHREG=OLD BYT3.
0095  06 B0                ASL SHREG        SHIFTED ONCE
0097  2A                   ROL A            INTO A.
0098  85 AC                STA BYT1
009A  06 B0                ASL SHREG        TWICE.
009C  2A                   ROL A
009D  06 B0                ASL SHREG        THRICE.
009F  2A                   ROL A
00A0  06 B0                ASL SHREG        FOURCE (FOURCE?).
00A2  2A                   ROL A
00A3  45 AC                EOR BYT1         EXOR WITH ONCE-SHIFTED BITS.
00A5  85 AC                STA BYT1         BACK INTO BYT1 AS NEW RANDOM #.
00A7  68                   PLA              GET Y
00A8  A8                   TAY              BACK.
00A9  A5 AC                LDA BYT1         LOAD NEW RANDOM NUMBER.
00AB  60                   RTS              RETURN.
```

Data Storage

ADDRESS	NAME	ADDRESS	NAME
00AC	BYT1	00B3	NUMBIT
00AD	BYT2	00B4	NOTESTAT
00AE	BYT3	00B5-00B8	TOTAL
00AF	BYT4	00B9-00BC	OLD
00B0	SHREG	00BD-00DC	SCALE
00B1	KEY	00DD	COUNTEMP
00B2	DELTA	0100-011F	RANDOM

SUMMARY

SIMPLIFIED QuASH
ADDRESS MAP
BASE ADDRESS 0900

QuASH#	CHANNEL			
	A	B	C	D
1	FF	FE	FD	FC
2	FB	FA	F9	F8
3	F7	F6	F5	F4
4	F3	F2	F1	F0*
5	EF*	EE*	ED*	EC*
6	EB*	EA*	E9*	E8*
7	E7*	E6*	E5*	E4*
8	E3*	E2*	E1*	E0*

* In systems using more than 15 channels, each 8781 BS pin must be tied to address line 5 (A4 - J7 pin 7) ON ALL 8781 MODULES IN THE SYSTEM.

To determine the address of a specific S/H channel, add the address of the channel as shown above to the base address 0900, eg. the address of channel B of the first QuASH is 09FE.

DATA (in HEX) AND CORRESPONDING NOTE NAMES

NOTE	OCTAVE 0	1	2	3	4	5
G#	00	0C	18	24	30	3C
A	01	0D	19	25	31	3D
A#	02	0E	1A	26	32	3E
B	03	0F	1B	27	33	3F
C	04	10	1C	28	34	
C#	05	11	1D	29	35	
D	06	12	1E	2A	36	
D#	07	13	1F	2B	37	
E	08	14	20	2C	38	
F	09	15	21	2D	39	
F#	0A	16	22	2E	3A	
G	0B	17	23	2F	3B	

Shaded area represents PAIA 8782 Keyboard active area.

NOTE: Moving across horizontal rows transposes by octaves.