U.S.A. $2.50
CANADA $3.00

SOUND & MUSIC

Antic™

The ATARI® Resource

NOVEMBER 1983 VOLUME 2, NUMBER 8

- Atari Service System
- ATARI Sound Editor
- Simple Synthesizer
- "Air Raid" Game
- Antic Pix Gifts

11
74470 12728

SOUND BEYOND SOUND

Breaking the BASIC barrier

by CHRIS CHABRIS

ATARI sound capabilities are very strong, compared to most other small computers, and are completely at the disposal of the programmer. Unfortunately, that access is not well explained in the *Atari BASIC Reference Manual*. You should reread pages 57–58 of the manual to fully benefit from this article, but I intend to take you well beyond the basics, show you some techniques for better use of sound, and finally give you a "sound editor" program to test and select various sounds for use in your programs.

First, let's examine the SOUND command in BASIC, to see how it works, and how we can avoid its limitations.

Sound is generated on ATARI computers by the chip called POKEY (an acronym for POtentiometers and KEYs), a cousin of the GTIA and ANTIC chips. POKEY is capable of routing four individual sound frequencies through the television speaker at any one time. Each frequency is associated with a value ranging from zero to 255, and so can be specified in one byte of memory. These independent frequencies are known as "voices" or "sound channels" because they play one tone at a time.

The frequency generated by each voice is affected by a control value for that voice, also a one-byte number. Packed into the control value are the distortion and volume parameters of the BASIC SOUND command. Distortion, used primarily to cause sound effects, ranges from zero to 14 (even values only), with 10 considered as "pure." Volume ranges from zero to 15, with eight considered normal.

How do these numbers reach POKEY so that it can generate the proper sounds? The POKEY chip itself is controlled by several hardware registers, which are like memory addresses on the chip. Values can be POKEd into them just like normal memory addresses. However, the value obtained by PEEKing into these registers is usually not the same as the one previously POKEd in. Generally, each register has two functions — one when it is read and another when it is written to.

As you may have expected, four of these registers hold the frequency values for the four voices and four of them hold the control values. Incidentally, when PEEKed, these registers will give you the instantaneous values of the eight Paddle Controllers, just as does the PADDLE command in BASIC.

Now let's try generating some sounds by POKEing these registers directly. The command SOUND 0,19,10,8 will produce a high G note of pure distortion and moderate volume on channel number one. How could we duplicate this with POKE commands? Consulting Table 1, we find that the frequency is POKEd into 53760. So type:

POKE 53760,19

The control value poses a greater problem. In order to get the distortion and volume numbers into one byte, use the following formula:

CONTROL = 16*DISTORTION + VOLUME

Therefore, the control value is 16*10 + 8, or 168. The control register for voice zero is 53761, so the second POKE statement is:

POKE 53761,168

If you turned off the original sound with a SOUND 0,0,0,0 command, you should now hear the same tone again.

Try the same experiment with other values, comparing the results obtained with SOUND and POKE. They will be identical.

Now let's consider one of the deficiencies of the Atari BASIC SOUND command, the lack of a duration parameter (I understand that the SOUND command in **Atari Microsoft BASIC** has such a parameter). To get around this deficiency, routines that play music or sound effects are often timed by FOR/NEXT loops. For example:

```
100 SOUND 0,128,10,8
110 FOR L = 1 TO 100:NEXT L
120 SOUND 0,0,0,0
```

This would play a low B note for whatever length of time it took the loop to execute. If we could make a table relating the lengths of empty loops to their execution times, this would be a perfect method of timing sounds. Unfortunately, we can't.

The speed of a FOR/NEXT loop depends on its position in the program; it is slowest at the end of a program and fastest at the beginning. The difference is especially noticeable in long programs. Try it yourself.

Luckily, there is an accurate way of measuring the duration of sounds. Every sixtieth of a second, memory location 540 decrements by one until it reaches zero. Then it is restored to 255, and starts to count down again. Therefore, if we want to play a middle C note for one second, we could use the following code:

```
100 SOUND 0,121,10,8
110 POKE 540,60
120 IF PEEK(540)>0 THEN 120
130 SOUND 0,0,0,0
```

This works very well (you can verify it with a stopwatch), and allows songs to be played with perfectly timed notes and rests.

continued on next page

Christopher Chabris is a high school senior and Junior Programmer for an IBM installation company in New York. He is a candidate Chess Master and has represented the U.S. in international Chess competition.

Location 540 may skip a count under certain conditions, but this is unlikely to occur in a BASIC music-playing or sound effects routine.

There is another problem with using any of the techniques here to play music in a program. If you use a subroutine to play a song, all other action in the program must stop while your routine executes. This can be tens of seconds with a long piece, causing a long interruption.

As usual, there is a solution — the vertical blank interrupt (VBI). This occurs almost precisely every sixtieth of a second, when the electron beam drawing the screen display reaches the lower right corner and must return to the upper left. The screen actually goes blank at this time, although it happens too fast for us to see. Whatever program in whatever language is executing at the time stops, and a different one begins. When its execution is complete, the main program resumes exactly where it stopped without knowing what happened.

This "secondary program" consists of two stages: immediate and deferred. The ATARI has an immediate routine built into the Operating System (OS) which maintains the real-time clock and timers (such as location 540), reads the controller ports, and performs other miscellaneous tasks.

Fortunately, there is no real built-in deferred VBI routine, so we can insert our own. Since the VBI routine is transparent to the main program, it can change the sound registers on POKEY and maintain durations in sixtieths of a second without visibly interfering with the applications program (Wouldn't you like a little background music during a long word processing session?).

There is one problem: a VBI routine must be written in machine language. Since most of you probably don't program in machine language, I have provided such a routine that you can incorporate in your own programs. It consists of three parts: the actual deferred VBI routine, which is POKEd into Page Six of memory (a reserved area); a short program, contained in a string, that makes the OS aware of our routine; and a similar program to turn off the VBI routine.

Listing 1 is a BASIC loader program to set up the necessary machine language. It ends with a RETURN command, so you can use it as a subroutine in your own programs.

The music-playing VBI is very easy to use in BASIC. Before using it, execute a GOSUB20000 command to load all of the necessary machine language into memory. Now, you must POKE in several parameters. Table 2 is a list of the various memory locations that hold these parameters. First, there is the number of voices that your music requires: 1, 2, 3, or 4. This is POKEd into memory location 1712. Next, for each voice, you must set the address of the music to be used with the following code:

```
V1A = ADR(V1$)
V1AH = INT(V1A/256)
V1AL = V1A-256*V1AH
POKE XXXX,V1AL
POKE XXXX + 1,V1AH
```

V1$ holds the music table for voice number one and XXXX is the address on Page Six that holds the address of the string containing the music (see Table 2). These lines should be repeated for each voice (V2$, V3$, V4$).

A music string contains values for each note grouped in pairs of *Frequency* and *Duration* (in sixtieths of a second). The frequencies 255 and 254, however, have special meanings and therefore cannot be used to specify notes. A 255 tells the VBI routine to completely turn off the voice whose string contains that byte. A 254 tells the routine to repeat the music for that voice from the beginning. One of the limitations of the routine is that a music string can be a maximum of 255 bytes long. Since each note requires two bytes, a voice may play a maximum of 127 notes before they either repeat or end. Most songs that would be used in programs are shorter than this, so it should not be a big problem.

If you want to turn off a voice, POKE a one (1) into its status address, locations 1725 through 1728 for the four voices. POKE a zero into a status address to restart the voice where it left off.

To completely end all voices and turn off the VBI routine, execute the following code:

```
Q = USR(ADR(RESET$))
FOR L = 0 TO 3
SOUND L,0,0,0
NEXT L
```

Pressing [SYSTEM RESET] will also kill the VBI routine. Remember, this information is summarized in Table 2.

To help you start using this system in your own programs, I have included Listing 2, a demonstration program using one voice to play the theme from a familiar movie. When merged with Listing 1 (through the use of the ENTER command), it will produce a skeleton program that you can modify and incorporate directly into your own code. It is documented by REMark lines and includes all necessary POKE commands. Have fun!

There is one more POKEY register to discuss, and it singlehandedly gives the ATARI greater sound capabilities than most, if not all other personal computers. Location 53768, known as AUDCTL (For AUDio ConTroL), provides you with eight additional options which affect all four sound channels. Table 3 is a summary of the functions available with the AUDCTL register. Each bit of AUDCTL enables an option when set to one and disables it when set to zero. To find the number to POKE into AUDCTL, add up the numbers to the left of each option that you want to enable. These options are explained in greater detail below.

Poly counters provide random pulses used to generate the distortion in the sound channels. By setting bit 7, you can make the patterns set by various distortion values more regular, because long polys do not generate obvious repetition.

Clocking a channel with 1.79 MegaHertz (millions of cycles per second) will simply result in a much higher pitch, because the normal clock base is set at 64 KHz, or 64,000 cycles per second. Bits 6 and 5, when set, change the clock base for voices one and three only.

POKEY normally has a frequency range of zero to 255, allowing for about four octaves. Setting bits 4 and 3 of AUDCTL joins two voices so that their combined frequency ranges from zero to 65535 (known as 16-bit resolution), resulting in a nine-octave range of pitches. The control registers of channels two and four determine the distortion and volume for the joined pairs "one and two" and "three and four." Of course, you cannot use the SOUND command with 16-bit music. The following routine will cycle through the 16-bit frequency range:

```
100 SOUND 0,0,0,0
110 POKE 53768,16
120 POKE 53761,0:POKE 53763,170
130 FOR LO = 0 TO 255
140 POKE 53762,LO
150 FOR LI = 0 TO 255
160 POKE 53760,LI
170 NEXT LI
180 NEXT LO
190 GOTO 100
```

At the end of the loop, when both LI and LO are 255, the output frequency is approximately one Hz, so you hear a low pop every second. Change the POKE value in line 110 to 80 and observe the difference. This switches the channel-one clock to 1.79 MHz and produces higher quality low notes. Remember that you should always execute a SOUND 0,0,0,0 command before the sound routine in your programs, in order to properly initialize POKEY.

Setting bits 2 and 1, enabling high-pass filters, allows only high frequencies to pass through a sound channel. The lowest frequency that can pass is defined as one more than the frequency of the clock channel. The following program uses a high-pass filter to produce an interesting sound, somewhat like a "red alert" on a spaceship:

```
100 SOUND 0,0,0,0
110 POKE 53768,4
120 POKE 53761,170
130 POKE 53764,150
140 FOR L = 255 TO 0 STEP -1
150 POKE 53760,L
160 NEXT L
170 GOTO 140
```

Other special effects can be created by varying the clock frequency while the filtered voice is playing.

The last option available, enabled by setting bit 0, is switching the clock base for all four voices from 64 KHz to 15KHz. With this clock base, the normal tones will be much lower, just as they were higher with bits 6 and 5 set. Go back to the 16-bit resolution music program and change line 110 to the following:

```
110 POKE 53768,17
```

By adding one, you change the main clock base as described above. Now RUN the program. At the end of the loop, the frequency is approximately 250 milliHertz, or one pop every four seconds!

Clearly, there is tremendous potential for experimentation with all these registers and options. As a bonus program, I have included a short sound-editor as Listing 3 that will let you change the frequency, distortion, and volume values for all four voices. The setting of each AUDCTL bit is displayed and alterable.

Once you have typed in the program and corrected any typos, RUN it. The screen turns orange for a few seconds and then displays a Graphics Mode 0 screen divided into six colored windows (the order of colors may vary slightly the next time you RUN the program).

The top window displays the title block, general status line, and command-reminder line. A "VOICE" indicates which voice you are working on at present, and the two numbers to the right of it are the values of the joined voices. These values are only updated when the appropriate AUDCTL bits are set, so you won't see them change at first. The last line of this window reminds you that the [START], [SELECT], and [OPTION] keys, the stick and trigger are the only inputs used to control the program.

The next four windows display all information for each of the four voices: frequency, volume, and distortion on the first line, and a graphic representation of the frequency on the next three lines.

The last window shows the status of the AUDCTL register, including the value of each bit and the total decimal value (useful for writing POKE commands in your own programs). Using Table 3 in conjunction with this window permits you to selectively enable the various options.

At the beginning of the program, the arrow on the graph for the frequency of voice number one is white and the other arrows are black. This is a reminder that you are working with voice number one now. Plug a joystick into Port One and move it left or right to increase or decrease the frequency value, and push the trigger to increase the frequency value by 25. Select another voice by moving the stick up or down, lighting up a different arrow.

By pushing the [SELECT] and [OPTION] console buttons, you can increase the volume and distortion of the chosen voice. When they reach the maximum value, they "roll over" to zero.

The [START] button is used to change bits of the AUDCTL register. The voice's arrow disappears, and a box appears around bit 7 in the bottom window. Move the stick left or right to place the box over a different bit number. Press the trigger to change the value of that bit from zero to one or vice-versa. To go back and edit the voice again, press [START]. That's all there is to it. When you produce an interesting sound, write down the values and use them in your own program.

While the two utilities I have presented are quite useful, there is always plenty of room for improvement. If you come up with an interesting modification to any of the listings, send it to me c/o ANTIC so other readers can take advantage of it in their own programming.

Some suggestions: Modify the VBI routine to allow variable volume and distortion values for each note, handle songs longer than 127 notes, and use 16-bit resolution music. For

continued on next page

you BASIC programmers, how about putting an option into the sound editor to display the meanings of the AUDCTL option and adding a sound storage feature (to disk or tape).

TABLE ONE: Sound Registers on POKEY

ADDRESS			
Decimal	Hexadecimal	Label	Function
53760	$D200	AUDF1	Voice #1 frequency (pitch)
53761	$D201	AUDC1	Voice #1 control (distortion/volume)
53762	$D202	AUDF2	Voice #2 frequency
53763	$D203	AUDC2	Voice #2 control
53764	$D204	AUDF3	Voice #3 frequency
53765	$D205	AUDC3	Voice #3 control
53766	$D206	AUDF4	Voice #4 frequency
53767	$D207	AUDC4	Voice #4 control

TABLE TWO: VBI Music Routine — Important Memory Locations

ADDRESS			
Decimal	Hexadecimal	Label	Function
1712	$06B0	NUMV	Number of voices to play
1713–1714	$06B1–06B2	V0ADR	Address of Voice #1 music in lobyte/hibyte form
1715–1716	$06B3–06B4	V1ADR	Address of voice #2
1717–1718	$06B5–06B6	V2ADR	Address of voice #3
1719–1720	$06B7–06B8	V3ADR	Address of voice #4
1721–1724	$06B9–06BC	DUR1–4	Duration of voices #1–4 in 60ths of a second
1725–1728	$06BD–06C0	STATUS1–4	Status of voices #1–4: 0 = play voice 1 = voice inactive
1729–1732	$06C1–06C4	COUNT1–4	Position in music string currently being accessed
1733–1736	$06C5–06C8	PAUSE1–4	Pause flag: 3 = pause in progress 0 = playing note
1737	$6C9	AUD	Value to be stored in the AUDCTL register (see text)

TABLE THREE: AUDCTL Register — Bit Options

Decimal Value	Bit Number	Function Enabled (if Bit Set to 1)
128	7	Turn 17-bit poly-counter into 9-bit poly-counter
064	6	Clock channel 1 with 1.79 MHz
032	5	Clock channel 3 with 1.79 MHz
016	4	Join channels 1 + 2 for 16-bit resolution
008	3	Join channels 3 + 4 for 16-bit resolution
004	2	Insert high-pass filter into channel 1 (clocked by channel 3)
002	1	Insert high-pass filter into channel 2 (clocked by channel 4)
001	0	Change main clock base to 15 KHz (normally 64 KHz)
_____		

TOTAL = Value to POKE into 53768 (or 1737 if you're using the VBI music routines)

Requires 16K RAM

Listing 1

```
20000 RESTORE 20010:FOR L=1536 TO 1737
:READ B:POKE L,B:NEXT L
20010 DATA 162,0,160,0,32,155
20020 DATA 6,189,177,6,133,203
20030 DATA 189,178,6,133,204,32
20040 DATA 161,6,189,189,6,208
20050 DATA 118,222,185,6,208,113
20060 DATA 188,193,6,177,203,201
20070 DATA 255,208,22,169,1,157
20080 DATA 189,6,169,0,32,155
20090 DATA 6,157,0,210,157,1
20100 DATA 210,32,161,6,76,143
20110 DATA 6,201,254,208,18,169
20120 DATA 0,157,193,6,169,1
20130 DATA 157,185,6,169,0,157
20140 DATA 197,6,76,143,6,72
20150 DATA 189,197,6,208,23,169
20160 DATA 3,157,185,6,157,197
20170 DATA 6,169,0,32,155,6
20180 DATA 157,0,210,32,161,6
20190 DATA 104,76,143,6,169,0
20200 DATA 157,197,6,104,32,155
20210 DATA 6,157,0,210,169,166
20220 DATA 157,1,210,32,161,6
20230 DATA 200,177,203,157,185,6
20240 DATA 200,152,157,193,6,232
20250 DATA 236,176,6,208,3,76
20260 DATA 167,6,76,4,6,72
20270 DATA 138,10,170,104,96,72
20280 DATA 138,74,170,104,96,173
20290 DATA 201,6,141,8,210,76
20300 DATA 98,228,0,0,0,0
20310 DATA 0,0,0,0,0,1
20320 DATA 1,1,1,0,0,0
20330 DATA 0,0,0,0,0,0
20340 DATA 0,0,0,0
20350 RESTORE 20360:DIM SET$(11):FOR L
=1 TO 11:READ B:SET$(L)=CHR$(B):NEXT L
20360 DATA 104,160,0,162,6,169
20370 DATA 7,32,92,228,96
20380 RESTORE 20390:DIM RESET$(11):FOR
 L=1 TO 11:READ B:RESET$(L)=CHR$(B):NE
XT L
20390 DATA 104,160,98,162,228,169
20400 DATA 7,32,92,228,96
20410 RETURN
```

TYPO TABLE

```
Variable checksum = 98156
   Line num range     Code   Length
   20000   - 20110      FB      325
   20120   - 20230      RJ      294
   20240   - 20350      ON      299
   20360   - 20410      FW      165
```

Listing 2

```
1000 REM - LOAD V1$ WITH MUSIC FROM DA
TA STATEMENTS
1010 DIM V1$(41):RESTORE 1090:FOR L=1
TO 40:READ B:V1$(L)=CHR$(B):NEXT L:V1$
(41)=CHR$(254)
1020 REM - SET UP THE THREE ROUTINES,
POKE IN NUMBER OF VOICES AND ADDRESSES
1030 GOSUB 20000:POKE 1712,1:V1A=ADR(V
1$):V1AH=INT(V1A/256):V1AL=V1A-256*V1A
H:POKE 1713,V1AL:POKE 1714,V1AH
1040 REM - POKE AUDCTL VALUE AND START
 ROUTINE
1050 POKE 1737,0:Q=USR(ADR(SET$))
1060 REM - YOUR PROGRAM COMES NEXT
1070 GOTO 1070
1080 REM - MUSIC DATA FOR VOICE #1 FOL
LOWS
1090 DATA 207,10,207,10,207,10,162,60,
108,60,121,10,128,10,144,10,81,60,108,
30,121,10,128,10,144,10,81,60,108,30
1100 DATA 121,10,128,10,121,10,144,30,
0,30
```

TYPO TABLE

```
Variable checksum = 146487
   Line num range     Code   Length
   1000    - 1090       ZO      575
   1100    - 1100       TD       38
```

Listing 3

```
1000 REM *** SOUND EDITOR, rev. 1.0
1010 REM *** By Chris Chabris 1983
1020 REM *** For ANTIC Magazine
1030 REM *** Allows modification of 4
1040 REM *** voices and AUDCTL value.
1050 FOR L=53760 TO 53768:POKE L,0:NEX
T L:SOUND 0,0,0,0:VOICE=1
1060 GRAPHICS 0:POKE 710,54:POKE 709,0
:POKE 752,1:POKE 82,1:POKE 83,38:? CHR
$(125):POKE 54286,64:GOSUB 1630
1070 REM - MAIN PROGRAM LOOP
1080 POKE 53279,0
1090 IF STRIG(0)=0 THEN 1340
1100 IF PEEK(53279)=6 THEN 1390
1110 IF PEEK(53279)=5 THEN 1560
1120 IF PEEK(53279)=3 THEN 1590
1130 IF STICK(0)=15 THEN 1090
1140 IF STICK(0)<>14 THEN 1170
1150 POKE 703+VOICE,0:VOICE=VOICE-1:IF
 VOICE=0 THEN VOICE=4
1160 POSITION 8,2:? VOICE;:POKE 703+VO
ICE,12:FOR L=1 TO 100:NEXT L:GOTO 1080
1170 IF STICK(0)<>13 THEN 1200
1180 POKE 703+VOICE,0:VOICE=VOICE+1:IF
 VOICE=5 THEN VOICE=1
1190 POSITION 8,2:? VOICE;:POKE 703+VO
ICE,12:FOR L=1 TO 100:NEXT L:GOTO 1080
1200 IF STICK(0)<>7 THEN 1270
1210 V=VOICE:IF FREQ(V-1)=255 THEN 109
0
1220 FREQ(V-1)=FREQ(V-1)+1:POSITION 19
,V*4:? FREQ(V-1);" ";
1230 POKE 53247+V,58+INT(FREQ(V-1)/2)
1240 POKE 53758+V*2,FREQ(V-1):IF AUDCT
L(4)=1 AND V<3 THEN POSITION 22,2:? "
     ";FREQ(0)+256*FREQ(1);
1250 IF AUDCTL(3)=0 OR V<3 THEN 1200
1260 POSITION 33,2:? "      ";FREQ
(2)+256*FREQ(3);:GOTO 1200
1270 IF STICK(0)<>11 THEN 1090
1280 V=VOICE:IF FREQ(V-1)=0 THEN 1090
1290 FREQ(V-1)=FREQ(V-1)-1:POSITION 19
,V*4:? FREQ(V-1);" ";
1300 POKE 53247+V,58+INT(FREQ(V-1)/2)
1310 POKE 53758+V*2,FREQ(V-1):IF AUDCT
L(4)=1 AND V<3 THEN POSITION 22,2:? "
     ";FREQ(0)+256*FREQ(1);
1320 IF AUDCTL(3)=0 OR V<3 THEN 1270
1330 POSITION 33,2:? "      ";FREQ
(2)+256*FREQ(3);:GOTO 1270
1340 V=VOICE:FREQ(V-1)=FREQ(V-1)+25:IF
 FREQ(V-1)>255 THEN FREQ(V-1)=0
1350 POKE 53247+V,58+INT(FREQ(V-1)/2):
POKE 53758+2*V,FREQ(V-1):POSITION 19,4
*V:? "    ";FREQ(V-1);
1360 IF AUDCTL(4)=1 AND V<3 THEN POSIT
ION 22,2:? "      ";FREQ(0)+256*FR
EQ(1);
1370 IF AUDCTL(3)<>1 OR V<3 THEN 1100
1380 POSITION 33,2:? "      ";FREQ
(2)+256*FREQ(3);:GOTO 1100
1390 Q=USR(ADR(SPRAY$),PMMEM+1024+(VOI
CE-1)*256,256):POSITION 1,3:? "  Togg
le AUDCTL bits with TRIGGER.  "
1400 POKE 53247+VOICE,0:RESTORE 1410:F
OR L=0 TO 11:READ B:POKE PMMEM+1024+(V
OICE-1)*256+198+L,B:NEXT L
1410 DATA 126,126,66,66,66,66,66,66,66
,66,126,126
1420 POKE 53247+VOICE,74:A=7:FOR L=1 T
O 50:NEXT L
1430 IF STRIG(0)=0 THEN 1480
1440 IF STICK(0)=11 THEN A=A+1:A=A-(8*
(A=8)):POKE 53247+VOICE,74+16*(7-A):FO
R L=1 TO 50:NEXT L:GOTO 1430
1450 IF STICK(0)=7 THEN A=A-1:A=A+(8*(
A=-1)):POKE 53247+VOICE,74+16*(7-A):FO
R L=1 TO 50:NEXT L:GOTO 1430
1460 IF PEEK(53279)=6 THEN 1520
1470 GOTO 1430
1480 POKE 53279,0:AUDCTL(A)=ABS(AUDCTL
(A)-1):SUM=0:FOR L=0 TO 7:IF AUDCTL(L)
=1 THEN SUM=SUM+INT(2^L+0.5)
1490 NEXT L:POKE 53768,SUM
```

continued on next page

```
1500 POSITION 7+4*(7-A),23:? AUDCTL(A)
;:POSITION 34,20:? "    ";SUM;
1510 POKE 53279,0:GOTO 1440
1520 Q=USR(ADR(SPRAY$),PMMEM+1024+256*
(VOICE-1)):POKE 53247+VOICE,0
1530 RESTORE 1680:FOR L=0 TO 7:READ B:
POKE PMMEM+1024+256*(VOICE-1)+(VOICE-1
)*32+72+L,B:NEXT L
1540 POKE 53247+VOICE,58+INT(FREQ(VOIC
E-1)/2)
1550 POSITION 1,3:? "  START-SELECT-OP
TION  STICK-TRIGGER ":GOTO 1080
1560 VOL(VOICE-1)=VOL(VOICE-1)+1:IF VO
L(VOICE-1)=16 THEN VOL(VOICE-1)=0
1570 POSITION 27,VOICE*4:? VOL(VOICE-1
);" ";:POKE 53759+VOICE*2,DIST(VOICE-1
)*16+VOL(VOICE-1):GOTO 1090
1580 FOR L=1 TO 50:NEXT L:GOTO 1090
1590 DIST(VOICE-1)=DIST(VOICE-1)+2:IF
DIST(VOICE-1)=16 THEN DIST(VOICE-1)=0
1600 POSITION 35,VOICE*4:? DIST(VOICE-
1);" ";:POKE 53759+VOICE*2,DIST(VOICE-
1)*16+VOL(VOICE-1)
1610 FOR L=1 TO 50:NEXT L:GOTO 1090
1620 REM - INITIALIZATION ROUTINE
1630 PMP=PEEK(106)-16:PMMEM=PMP*256
1640 DIM SPRAY$(42):RESTORE 1650:FOR L
=1 TO 42:READ B:SPRAY$(L)=CHR$(B):NEXT
 L:Q=USR(ADR(SPRAY$),PMMEM,2048)
1650 DATA 104,104,133,204,104,133,203,
104,133,206,104,133,205,166,206,160,0,
169,0,145,203,136
1660 DATA 208,251,230,204,202,48,6,208
,244,164,205,208,240,198,204,160,0,145
,203,96
1670 FOR L1=0 TO 3:RESTORE 1680:FOR L2
=0 TO 7:READ B:POKE PMMEM+1024+L1*256
+L1*32+72+L2,B:NEXT L2:NEXT L1
1680 DATA 8,8,8,8,8,62,28,8
1690 RESTORE 1700:FOR L=0 TO 28:READ B
:POKE 1536+L,B:NEXT L:POKE 207,0:POKE
512,0:POKE 513,6
1700 DATA 8,72,138,72,141,10,212,166,2
07,189,32,6,141,24,208,232,224,5,208,2
,162,0,134,207,104,170,104,40,64
1710 POKE 1568,86:POKE 1569,118:POKE 1
570,166:POKE 1571,198:POKE 1572,246
1720 DL=PEEK(560)+256*PEEK(561):FOR L
=8 TO 24 STEP 4:POKE DL+L,130:NEXT L
1730 FOR L=704 TO 707:POKE L,0:NEXT L:
POKE 54279,PMP:POKE 53277,3:POKE 623,1
:POKE 559,62:POKE 54286,192
1740 POSITION 1,0:? "  **********  SOUN
D EDITOR  **********":? "   *****  By Ch
ris Chabris  1983  *****"
1750 ? :? "  START-SELECT-OPTION  STIC
K-TRIGGER "
1760 FOR L=4 TO 16 STEP 4:POSITION 1,L
:? "  VOICE #";L/4;":    FREQ=0    VOL=6
 DIST=10":POSITION 1,L+2
1770 ? "  ++++++++++++++++++++++++++++
++++":? "  000       064       128       192
      255":NEXT L
1780 POSITION 1,20:? "AUDCTL SETTING:
    (TOTAL VALUE=0   )":? "BIT | 7 | 6
| 5 | 4 | 3 | 2 | 1 | 0 |"
1790 ? "  +---+---+---+---+---+---+---+---
---+":? "IS  | 0 | 0 | 0 | 0 | 0 |
0 | 0 | 0 |";
1800 FOR L=53248 TO 53251:POKE L,58:NE
XT L
1810 POSITION 2,2:? "VOICE=1          V1
+2=0      V3+4=0     "
1820 DIM FREQ(3),VOL(3),DIST(3),AUDCTL
(7):FOR L=0 TO 3:FREQ(L)=0:VOL(L)=6:DI
ST(L)=10:POKE 53761+2*L,166:NEXT L
1830 FOR L=0 TO 7:AUDCTL(L)=0:NEXT L:P
OKE 704,12:RETURN
```

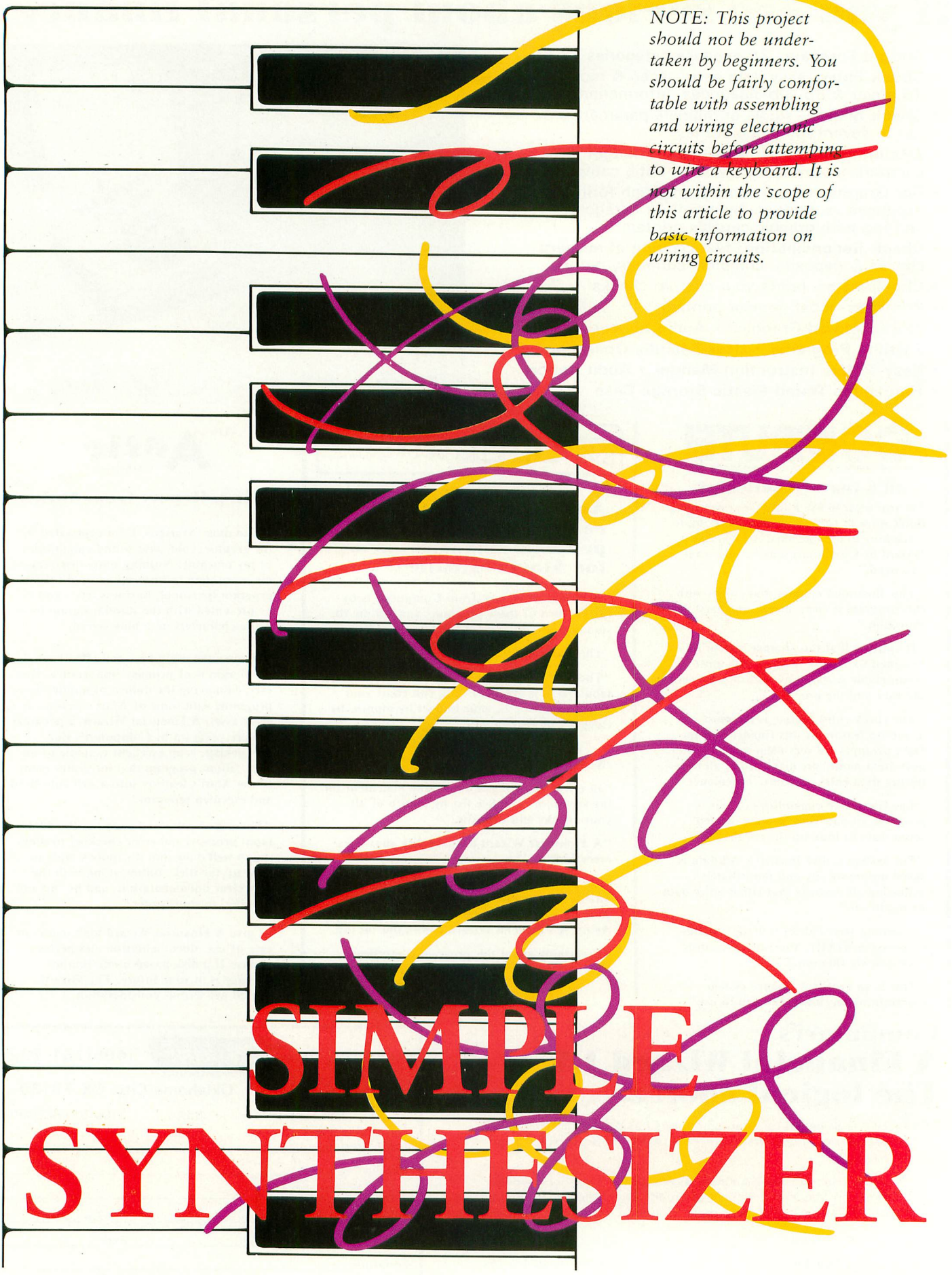
TYPO TABLE

```
Variable checksum = 399456
    Line num range     Code    Length
      1000  - 1100       PF       500
      1110  - 1210       ST       514
      1220  - 1290       KH       527
      1300  - 1350       TZ       513
      1360  - 1420       QQ       538
      1430  - 1500       UW       576
      1510  - 1570       SW       513
      1580  - 1650       RT       503
      1660  - 1710       PU       517
      1720  - 1770       LE       563
      1780  - 1830       AD       503
```

WHAT IS A TYPO TABLE

Newcomers to ANTIC may wonder about the "Typo Table" that appears at the end of most of our basic listings. TYPO is a program that helps you find typing errors made when entering programs that appear in ANTIC. TYPO will produce a table of values which can be used to pinpoint where an error was made. The TYPO program and instructions originally appeared in Volume 1, Number 3 of ANTIC, and was reprinted in Volume 2, Number 1. The latter issue is still available as a back issue, and the TYPO program itself is included in ANTIC UTILITIES DISK #2.

NOTE: When comparing your TYPO TABLE with the one we publish, first look at the length column. For a given line number range, if your length is only off by one or two, it may be due to spacing. Missing or extra spaces generally occur between quotes or in a REM statement. Spaces must be accurately placed for TYPO to work, so first experiment with the spacing.



NOTE: This project should not be undertaken by beginners. You should be fairly comfortable with assembling and wiring electronic circuits before attemping to wire a keyboard. It is not within the scope of this article to provide basic information on wiring circuits.

SIMPLE SYNTHESIZER

Make a musical peripheral for your ATARI

by VERN MASTEL

This article shows you how to build and connect a true piano-type keyboard that you can use with your ATARI computer as a simple music synthesizer. Included in the article is the software that allows you to play music.

The method is very simple from a hardware standpoint, requiring only some signal diodes and microswitches. The software will run on any ATARI computer from 16K on up with no modifications. Best of all, the program uses only BASIC — no machine language subroutines are used.

To begin, a discussion of the ATARI parallel data bus and joystick mechanics is necessary. "Beg pardon," you say, "the ATARI doesn't have a parallel data bus." Well, strictly speaking, it doesn't; but what it does have comes awfully close in this application. The ATARI joystick is a wonderfully simple device, consisting of only five switches and a housing. Four switches sense the major positions, and the fifth one senses the trigger (see Figures 1 and 2). Moving the stick in any given direction never closes more than two of the four switches at one time. Take a look at Program 1. Most will recognize it as a simple program to read the joystick position and print out the number corresponding to it.

Program 1

```
10 A=STICK(0):REM CAN BE ANY OF THE FOUR
   STICKS
20 PRINT A
30 GOTO 10
```

Note how the numbers run 5, 6, 7, 9, 10, 11, 13, 14, 15. What happened to 0, 1, 2, 3, 4, 8, 12? Now take the top off the joystick, RUN Program 1 and press two or more switches at once. Presto, the missing numbers appear! For a diagram of the switch patterns and the corresponding number returned by the STICK command, see Figure 3. What all of this boils down to is that each controller port on the ATARI can be treated as a four-bit parallel input port. Four bits are half of a byte and can be used to count in base 16. By selectively grounding combinations of the four input pins, the numbers from zero to 15 will be generated. This is the key to our Simple Synthesizer.

First, a keyboard is required. It can be anything from a dime store toy piano to a grand piano as long as it has the right complement of keys. I got mine from a junked Wurlitzer organ at a local music store for only $20. One octave (C through C) is all that is needed, so you should not have any problem finding something to suit. One octave is only 13 notes and there are 16 combinations available on one input port. If the note values are stored in a matrix in the same sequence they follow on the keyboard, then pressing a key will play the associated value stored in the matrix. For example,

Program 2

```
10 DIM M(3):M(1)=243 :M(2)=217: M(3)=193
20 FOR X=1 TO 3:SOUND 0,M(X),10,10
30 FOR D=1 TO 100:NEXT D:REM DELAY
40 NEXT X:SOUND 0,0,0,0
```

This program will play three notes, C, D and E, in succession. The SOUND command does not care where the arithmetic value which specifies the pitch comes from, so what happens if a joystick is used to generate the values? Try Program 3 and find out.

Program 3

```
 10 DIM M(13)
 20 FOR X=1 TO 13
 30 READ A
 40 M(X)=A
 50 NEXT X:REM GET THE NEXT VALUE
 60 A=STICK(0):IF A=0 THEN 60
 70 SOUND 0,M(A)-4,10,10
 80 GOTO 60
 90 DATA 243,230,217,204,193,182
100 DATA 173,162,153,144,136
110 DATA 128,121
```

What this program does is to put 13 note values (one octave) into matrix M and then use the joystick to change the pitch. Are you beginning to get the idea of how the Simple Synthesizer will work?

One problem remains to be tackled. To generate each note with a keypress on a keyboard at least two switches would be needed if you counted from three to 15. If you count from one to 13 then at least three would be needed. The problem comes when you try to mount three microswitches under one key — there simply isn't enough room. The solution lies in the use of a diode matrix as shown in Figure 4. Using this, only one switch is needed to handle each number from one to 13. The diodes pass DC current in only one direction and thus isolate each port pin from the others.

One additional ATARI hardware feature used with the Simple Synthesizer is the game paddles. Each port handles two paddles. For this application, two ports are used. The reason for this is to allow individual control of the volume of each voice of the Synthesizer. Normally, the resistance value

continued on next page

Vern Mastel is the manager of a Team Electronics store in Bismarck, ND, where he sells computers, including the ATARI, and related equipment. His favorite pastime is writing programs.

of the potentiometers used in the paddles is one megohm (1,000,000 ohms). This value gives a count range of zero to 228. To use the paddles to control the volume, a range of only zero to 10 or 12 is needed. If you are using more than three voices, the highest recommended value for volume is eight. If the sum of all volumes exceeds 32, sound is distorted. You could use a one-meg pot and divide the count to limit it to 12, but that wastes time in the program and upsets the smooth operation of the notes. The solution is to use a 10K pot (10,000 ohms). This gives a working range of zero to about seven, which is just right.

Now that the hardware has been covered, it is time for the software. Program 4 is the listing of the program which allows you to use the keyboard. Sound generation is handled by three separate routines — one each for multiple-voice mode, chord mode and piano mode. The modes are what the names imply. In the multiple-voice mode each voice is active, its level or volume determined by the setting of the associated paddle pot. In the chord mode, pressing a key generates the major chord of that key. For example, pressing C also generates the notes E and G.

It is interesting to note that Atari says this cannot be done well from BASIC because BASIC is too slow. As you will see when you RUN the program it is not only possible, it works very well.

The piano mode is the most interesting of the group because it involves dynamics which really do strain BASIC to the limit. When in this mode, a key which is pressed and held will decay to zero volume and not play again till the key is released and pressed again. If the key is released before the decay is finished, the note will be cut off (this is similar to what you would encounter with the rest pedal on a piano). If one note is pressed and then another, before the first has decayed, the second note will then sound and be allowed to decay. The rate of decay is determined by the size of the D loop in line 60 of the program. A larger value will take longer to decay; a smaller value will yield a shorter decay.

In all modes, the frequencies of all of the notes are stored in matrixes (L)ow, (M)edium and (H)igh. These matrixes are automatically loaded with the correct values at the time of mode selection. You will notice that the screen is blank when a play mode is selected. This was done after I noticed that some notes would be missed if I played fast enough with the screen on. Turning off the screen allows the Operating System to read the ports more often and thus keeps the notes from getting lost.

Rather than going into a long line-by-line description of the program, I chose to comment it heavily. These REMarks are strictly for illustration and should be deleted from the program after you have checked it with TYPO. If REM statements are not removed, the piano voice may be adversely affected. This program is by no means the final word in synthesizers, but it is a springboard from which you can proceed.

Use of the program is very simple. When RUN, a menu of six choices is displayed:

1) Multiple voices
2) 1st Octave chords
3) 2nd Octave chords
4) Piano Octave 1
5) Piano Octave 2
6) Piano Octave 3

Pressing the number of the desired mode activates the music keyboard. The screen will go blank at the same time. Pressing any key on the computer keyboard will restore the menu. In the first three modes you can control the volume of the individual voices with the pots connected to the paddle inputs. In the piano mode, the volume is fixed.

Lastly, to all of the musicians out there, please don't attack me over the accuracy of the pitch of all of the notes. The numbers in the program are from the official Atari pitch chart. I know they aren't right on the button but remember this is done for fun, not for the New York Philharmonic.

Figure 1. Synthesizer viewed from top rear. Note plugs for joystick ports.

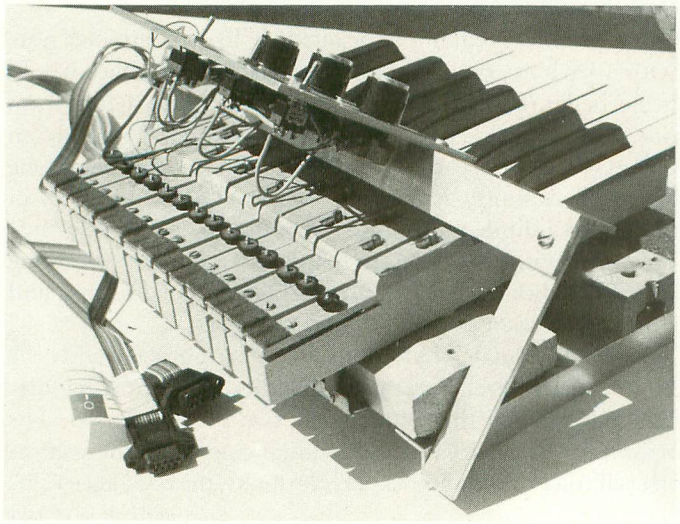
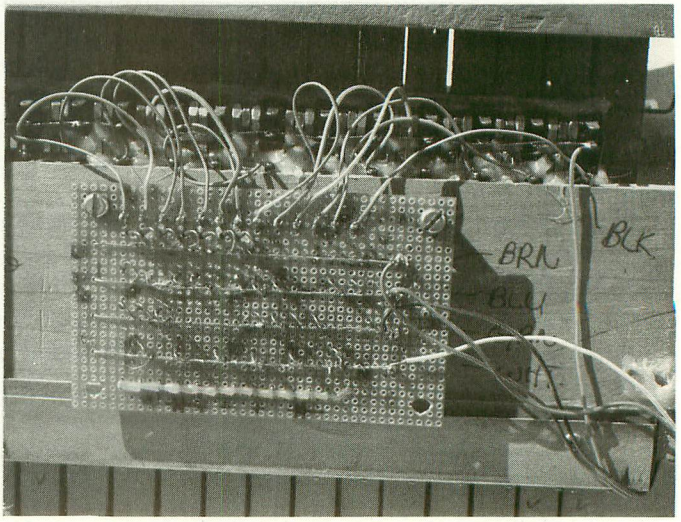


Figure 2. Underside of synthesizer shows wiring from switches to diode matrix.



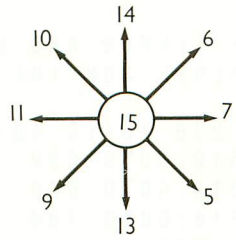
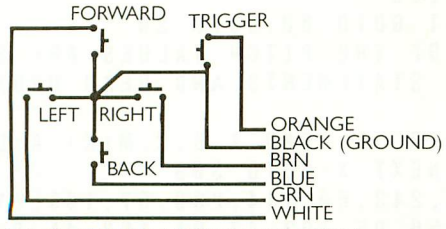


Figure 1



ALL SWITCHES NORMALLY OPEN.

Figure 2

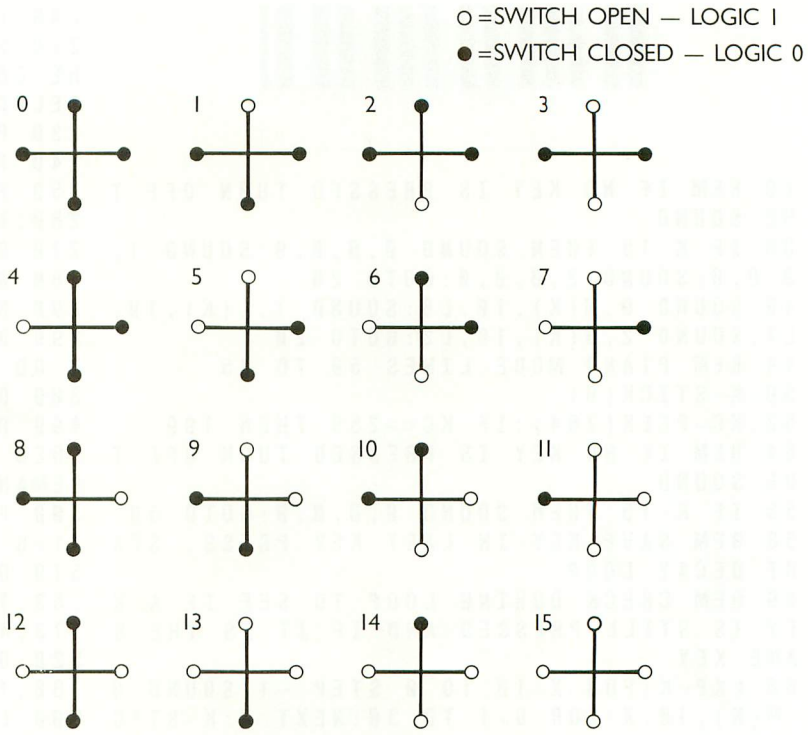
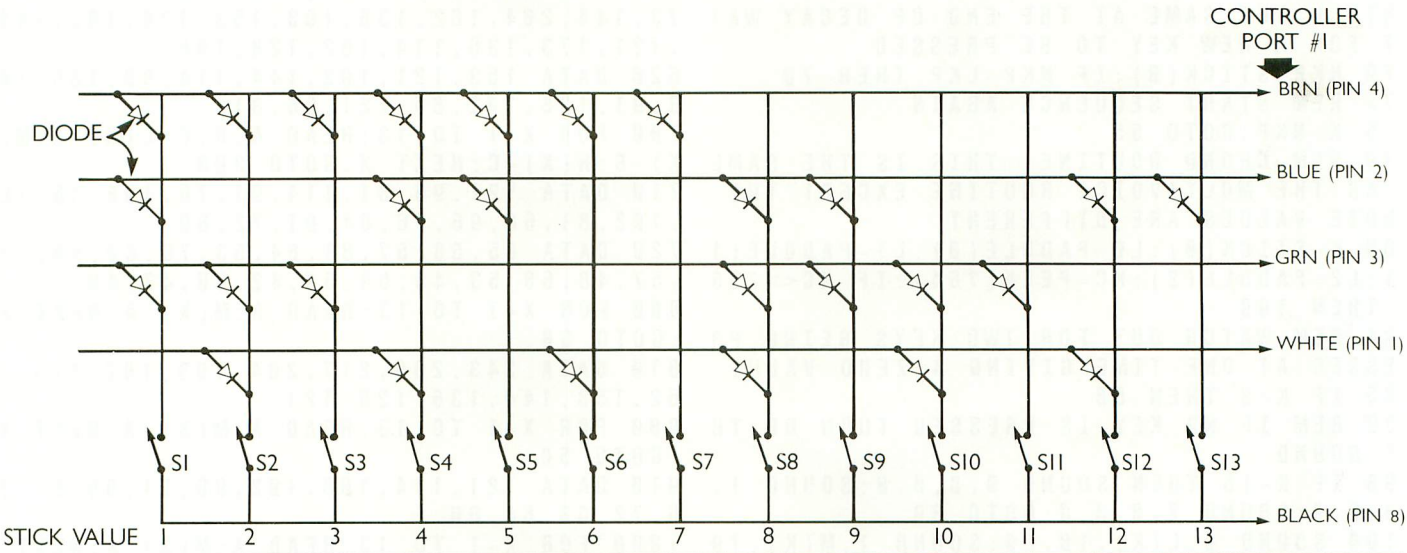
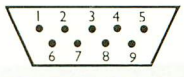


Figure 3



ALL DIODES — IN914 SIGNAL DIODES.
ALL SWITCHES — LEVER ACTION MICROSWITCH.



FRONT VIEW OF CONSOLE
(PORT) CONNECTOR.

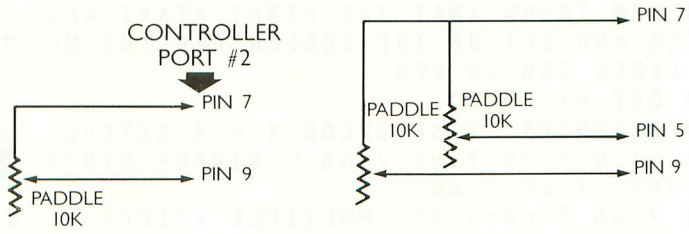
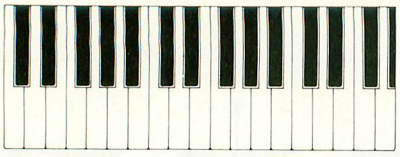


Figure 4

Requires 16K RAM and BASIC

```
1 REM PROGRAM 4
2 REM THE SIMPLE SYNTHESIZER BY VERN L
. MASTEL
4 REM SET UP MATRIXES, OPEN KEYBOARD F
OR INPUT
5 DIM H(13),M(13),L(13):OPEN #1,4,0,"K
":TRAP 300:GOTO 200
19 REM MULTIVOICE ROUTINE LINES 20-40
20 K=STICK(0):L0=PADDLE(0):L1=PADDLE(1
):L2=PADDLE(2):KC=PEEK(764):IF KC<>255
 THEN 199
```

continued on next page



```
29 REM IF NO KEY IS PRESSED TURN OFF T
HE SOUND
30 IF K=15 THEN SOUND 0,0,0,0:SOUND 1,
0,0,0:SOUND 2,0,0,0:GOTO 20
40 SOUND 0,M(K),10,L0:SOUND 1,L(K),10,
L1:SOUND 2,H(K),10,L2:GOTO 20
49 REM PIANO MODE LINES 50 TO 75
50 K=STICK(0)
52 KC=PEEK(764):IF KC<>255 THEN 199
54 REM IF NO KEY IS PRESSED TURN OFF T
HE SOUND
55 IF K=15 THEN SOUND 0,0,0,0:GOTO 50
58 REM SAVE KEY IN LAST KEY PRESS, STA
RT DECAY LOOP
59 REM CHECK DURING LOOP TO SEE IF A K
EY IS STILL PRESSED AND IF IT IS THE S
AME KEY
60 LKP=K:FOR X=10 TO 0 STEP -1:SOUND 0
,M(K),10,X:FOR D=1 TO 30:NEXT D:K=STIC
K(0):IF K=LKP THEN NEXT X
69 REM READ KEYBOARD AGAIN. IF KEY IS
STILL THE SAME AT THE END OF DECAY WAI
T FOR A NEW KEY TO BE PRESSED
70 NKP=STICK(0):IF NKP=LKP THEN 70
74 REM START SEQUENCE AGAIN
75 K=NKP:GOTO 55
79 REM CHORD ROUTINE. THIS IS THE SAME
 AS THE MULTIVOICE ROUTINE EXCEPT THE
NOTE VALUES ARE DIFFERENT
80 K=STICK(0):L0=PADDLE(0):L1=PADDLE(1
):L2=PADDLE(2):KC=PEEK(764):IF KC<>255
 THEN 199
84 REM WATCH OUT FOR TWO KEYS BEING PR
ESSED AT ONE TIME GIVING A ZERO VALUE
85 IF K=0 THEN 80
89 REM IF NO KEY IS PRESSED TURN OF TH
E SOUND
90 IF K=15 THEN SOUND 0,0,0,0:SOUND 1,
0,0,0:SOUND 2,0,0,0:GOTO 80
100 SOUND 0,L(K),10,L0:SOUND 1,M(K),10
,L1:SOUND 2,H(K),10,L2:GOTO 80
198 REM THROW AWAY THE FIRST ATARI KEY
PRESS AND SET UP THE SCREEN FOR THE ME
NU LINES 200 TO 220
199 GET #1,NULL
200 GRAPHICS 17:SETCOLOR 4,5,4:SETCOLO
R 2,11,6:? #6:? #6:? #6;" SIMPLE SYNTH
ESIZER":? #6:? #6
210 ? #6:? #6:? #6;"MULTIPLE VOICES---
1":? #6:? #6;"1ST OCTAVE CHORDS-2":? #
6:? #6;"2ND OCTAVE CHORDS-3"
215 ? #6:? #6;"PIANO OCTAVE 1----4":?
#6;"PIANO OCTAVE 2----5":? #6;"PIANO O
CTAVE 3----6"
219 REM MAKE SURE THE INPUT IS VALID A
ND PROCESS THE CHOICE
220 ? #6:? #6:? #6;" SELECT MODE (1-6)
";:GET #1,C:IF C<49 OR C>54 THEN 200:F
LAG=0
229 REM TURN OFF THE SCREEN AND LOAD T
HE CORRECT MATRIX VALUES FOR THE MODE
SELECTED
230 POKE 559,0:GOTO 230+10*(C-48)
240 FLAG=0:RESTORE 510:GOTO 500
250 FLAG=1:RESTORE 610:GOTO 600
260 FLAG=1:RESTORE 710:GOTO 700
270 RESTORE 810:GOTO 800
280 RESTORE 910:GOTO 900
290 RESTORE 1010:GOTO 1000
299 REM DO CHORDS IF SELECTED OTHERWIS
E DO MULTIVOICE
300 ON FLAG=1 GOTO 80:GOTO 20
499 REM ALL OF THE PITCH VALUES ARE ST
ORED AS DATA STATEMENTS AND READ UPON
DEMAND
500 FOR X=1 TO 13:READ A,B,C:M(X)=A:L(
X)=B:H(X)=C:NEXT X:GOTO 300
510 DATA 121,243,60,114,230,57,108,217
,53,102,204,50,96,193,47,91,182,45,85,
173,42,81,162,40,76,153,37
520 DATA 72,144,35,68,136,33,64,128,31
,60,121,29
600 FOR X=1 TO 13:READ A,B,C:L(X)=A:M(
X)=B:H(X)=C:NEXT X:GOTO 300
610 DATA 243,193,162,230,182,153,217,1
73,144,204,162,136,193,153,128,182,144
,121,173,136,114,162,128,108
620 DATA 153,121,102,144,114,96,136,10
8,91,128,102,85,121,96,81
700 FOR X=1 TO 13:READ A,B,C:L(X)=A:M(
X)=B:H(X)=C:NEXT X:GOTO 300
710 DATA 121,96,81,114,91,76,108,85,72
,102,81,68,96,76,64,91,72,60
720 DATA 85,68,57,81,64,53,76,60,50,72
,57,48,68,53,45,64,50,42,60,47,40
800 FOR X=1 TO 13:READ A:M(X)=A:NEXT X
:GOTO 50
810 DATA 243,230,217,204,193,182,173,1
62,153,144,136,128,121
900 FOR X=1 TO 13:READ A:M(X)=A:NEXT X
:GOTO 50
910 DATA 121,114,108,102,96,91,85,81,7
6,72,68,64,60
1000 FOR X=1 TO 13:READ A:M(X)=A:NEXT
X:GOTO 50
1010 DATA 60, 57,53,50,47,45,42,40,37,
35,33,31,29
```

TYPO TABLE

```
Variable checksum = 243534
   Line num range     Code   Length
      1      - 30       SL      502
     40      - 60       CW      523
     69      - 89       OK      520
     90      - 210      AI      612
    215      - 260      AP      521
    270      - 600      WT      521
    610      - 900      QW      521
    910      - 1010     GW      143
```

PICK A CHORD

Your ATARI can play most of them

by KARL WIEGERS

Impressive sound-generating capabilities have been built into all ATARI computers. Four separate "voices" can be used to generate a wide variety of music and sound effects through the speaker in the television set. A special ATARI BASIC statement, the SOUND statement, is used to control the pitch, distortion, and loudness of each voice.

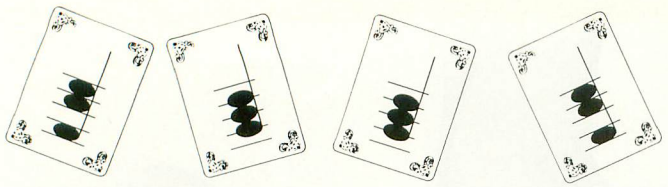
Since most chords commonly used in music involve only three to five separate notes, the four voices in the ATARI can be used in unison to play many different chords. The program which accompanies this article allows you to hear twelve different kinds of chords, in any musical key. This program can teach you a bit about musical chord structure and also how to use the sound-producing features of ATARI computers.

CHORD THEORY IN A NUTSHELL

There are really only twelve different musical notes: C, C# (or Db), D, D# (Eb), E, F, F# (Gb), G, G# (Ab), A, A# (Bb), and B. All other notes are formed by altering the pitch of one of these twelve by one or more octaves. Playing twelve notes in sequence constitutes a "chromatic scale." However, since you can start the scale at any one of the twelve notes, it follows that there are twelve different chromatic scales, each of which defines one of the twelve possible "keys" in which music can be played.

Chords are composed of certain notes selected from a particular chromatic scale. It is the "intervals" between the notes which give each chord its characteristic sound. The specific notes involved affect only the key the chord is played in, not the nature of the chord itself. For example, a major chord always consists of the 1st, 5th, and 8th notes in the chromatic scale for some key. For the key of C these notes are C, E, and G. In the key of F, a major chord (F major) contains the notes, F, A, and C. The intervals remain the same as the key changes, although the actual notes played will be altered. Changing

continued on next page



just one note in the chord can dramatically alter the sound of the chord and hence its auditory and emotional impact on the listener.

Let's return to the C major chord. There is nothing that says the lowest-pitched note in this chord must be a C. The notes can be played in any "inversion," or sequence of pitch. Some possibilities are (in order of increasing pitch): C, E, G; E, G, C; G, C, E; G, E, C; and any other sequence. If more than three notes are to be played, then one of these three must be repeated an octave (or several octaves) higher or lower. That is, the 17th note in the chromatic scale is the same as the 5th note, but is one octave higher in pitch. Adding any different notes will change the chord from C major to something else. Each inversion has its own particular sound, but the chord often sounds most characteristic when the lowest-pitched note is the first note in the scale.

ABOUT THE CHORD PROGRAM

The chord program shows you two menus, one for the key and one for the type of chord you wish to hear. There are only twelve possible keys, but there are many chords besides the dozen common ones included in this program. To select a chord, just enter the number for a key and then the number for the chord when the program prompts you. Any invalid entry will result in an obnoxious noise and a reminder to enter only numbers 1 through 12. The name of the chord you selected will be displayed, and arrows will appear next to the notes which constitute that chord. (If only three arrows appear, then the first note in the chromatic scale is being duplicated one octave higher by the fourth voice.) Then press [RETURN] to play another chord, or press [ESC] to quit using the chord program.

Be very careful to type in the program correctly, and SAVE it to cassette or diskette before running it for the first time. Count the spaces in lines 70 and 80 carefully, or the menus will look funny and you may get a BASIC error number 5 at line 230 when you run the program. It might help to remember that each key name occupies five characters in the MENU1$ array, while each chord name occupies six characters in MENU2$.

The DATA statements in lines 100–130 contain the numeric pitch values for the musical notes used by the program. These were taken from Table 10–1 of the *Atari BASIC Reference Manual*. Errors here will result in incorrect notes being played. Notice that the computer will create sounds if pitch values between the numbers in lines 100–130 are used, but the sounds will not correspond exactly to any note in the chromatic scale.

The DATA statements in lines 150–170 contain the notes of the chromatic scale which are to be sounded for each kind of chord. For example, the first four numbers in line 150 correspond to the notes in a C major chord: the 1st, 8th, 13th (same as 1st but one octave higher), and 17th (same as 5th but one octave higher) notes in the chromatic scale starting with C. You will hear a C, G, high C, and high E for a C major chord. You can add or substitute other chords for these twelve by modifying the appropriate numbers in lines 150–170 and changing the corresponding chord name in line 80. The table below gives the intervals for some other chords you might like to try.

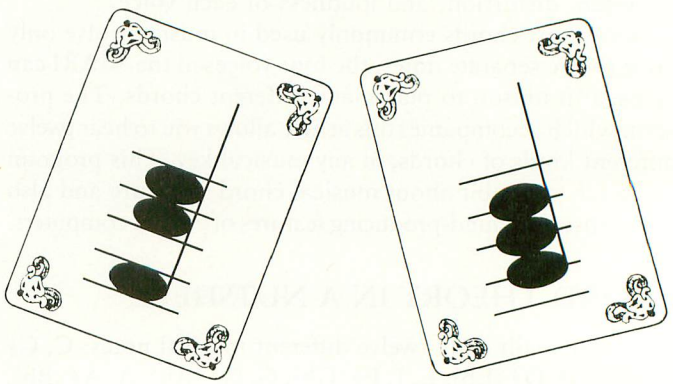
The DATA statements in lines 190–200 are a short machine language program called a "display list interrupt," or DLI (see ANTIC, June 1983, page 24). The DLI changes the color of the last five lines of the screen to orange with white characters, instead of green with black characters. If you don't like orange, change the 248 in line 190 to some other number. If you don't like green, change the SETCOLOR 2,12,8 statement in line 50. Typing errors in lines 190–200 may cause the computer to "lock up," so SAVE the program before running it the first time.

The rest of the program sets up the screen display, allows you to enter your chord selection, and plays the desired chord. Line 370 loads the correct pitch values into an array, and line 400 uses these values to activate the four voices. All four voices are sounded as pure tones (distortion = 10). When using several voices at once like this, the sum of all the loudness settings should not exceed 32, to avoid distortion of the sound. You may have to turn the volume on your TV set up a bit higher than normal, since the voices are all played at a loudness of only three for best sound quality. Line 410 plays the chord for about two second using an empty FOR/NEXT loop, then line 420 turns off all four voices.

If you want the chord to play continuously until you request a different one, then delete program lines 420–480 and change line 410 to GOTO 280. The resulting program turns your ATARI into a kind of chord organ. Try playing different sequences of chords, or "progressions," to see how they sound. You can even play songs now, but the rhythm is rather monotonous! Stop the sound and exit from the program by pressing [SYSTEM RESET].

Table 1. Additional Chords

CHORD	INTERVALS	CHORD	INTERVALS
7sus4	1,8,11,18	7#5	1,11,17,21
7#9	1,5,11,16	7b9	1,5,11,14
13th	1,11,17,22	min13	1,11,16,22
maj13	1,12,17,22	maj9	1,5,12,15
6/9	1,5,10,15	min6	1,8,10,16



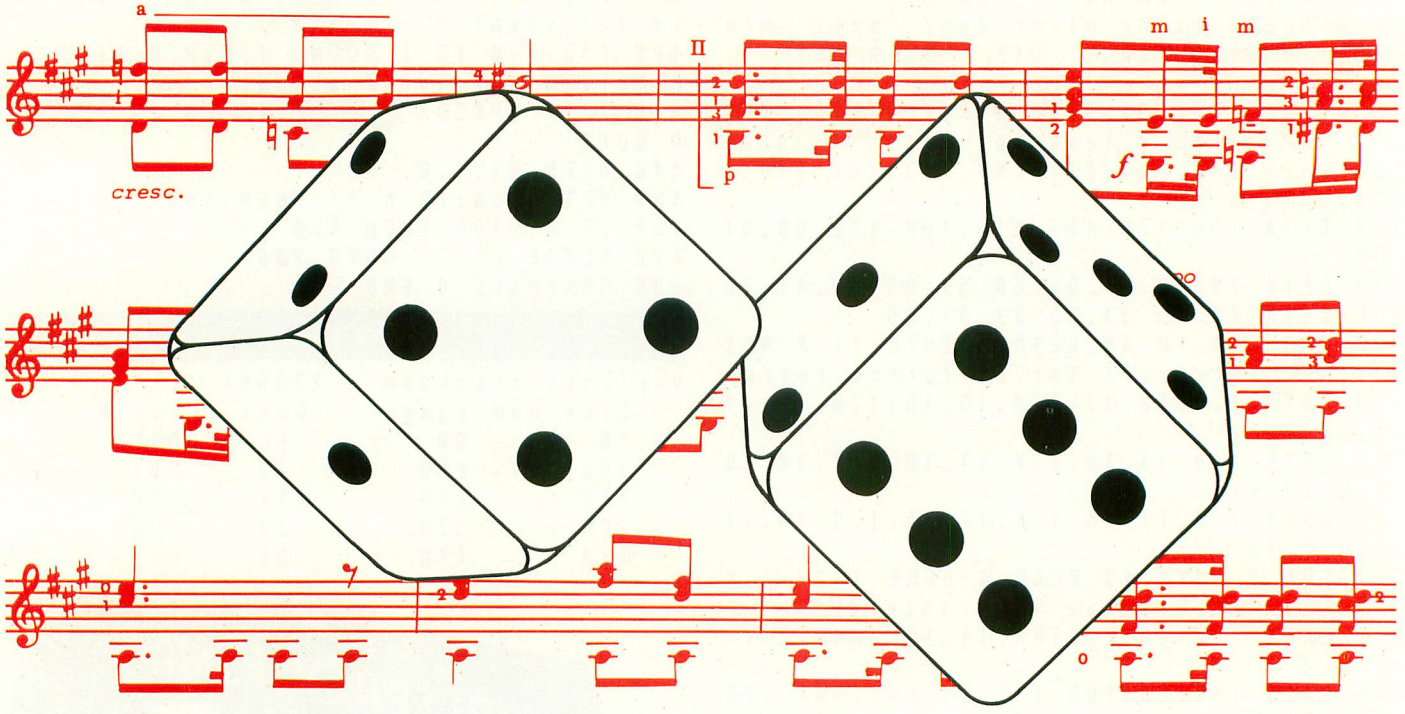
Requires 16K RAM

```
10 REM ......CHORD PROGRAM......
20 REM ......by Karl E. Wiegers.....
30 REM .....ANTIC MAGAZINE 1983.....
40 DIM MENU1$(60),MENU2$(72),PITCH(37)
,INTS(48),N(4)
50 GRAPHICS 0:SETCOLOR 1,0,0:SETCOLOR
2,12,8:POKE 752,1
60 POSITION 5,2:? "ONE MOMENT, PLEASE.
..."
70 MENU1$="C     C#/DbD     D#/EbE     F
   F#/GbG     G#/AbA     A#/BbB     "
80 MENU2$="major minor dom7  maj7  min
7 min7b5dim7  aug   9th   min9  6th
sus4  "
90 FOR I=1 TO 37:READ A:PITCH(I)=A:NEX
T I:REM Pitch values for musical notes
100 DATA 243,230,217,204,193,182,173,1
62,153,144
110 DATA 136,128,121,114,108,102,96,91
,85,81
120 DATA 76,72,68,64,60,57,53,50,47,45
130 DATA 42,40,37,35,33,31,29
140 FOR I=1 TO 48:READ A:INTS(I)=A:NEX
T I:REM Intervals for different chords
150 DATA 1,8,13,17,1,8,13,16,1,8,11,17
,1,8,12,17
160 DATA 1,8,11,16,1,7,11,16,1,7,10,16
,1,9,13,17
170 DATA 1,5,11,15,1,4,11,15,1,8,10,17
,1,6,8,13
180 FOR I=0 TO 19:READ A:POKE 1536+I,A
:NEXT I:REM Display list interrupt
190 DATA 72,138,72,169,14,162,248,141,
10,212
200 DATA 141,23,208,142,24,208,104,170
,104,64
210 ? CHR$(125):REM Clear the screen
220 POSITION 10,2:? "KEY":POSITION 25,
2:? "CHORD"
230 FOR I=1 TO 12:POSITION 8-(I>9),3+I
:? I;". ";MENU1$(5*I-4,5*I)
240 POSITION 23-(I>9),3+I:? I;". ";MEN
U2$(6*I-5,6*I):NEXT I
250 POKE 703,4:REM Set up a text windo
w in this Graphics Mode 0 display
260 POKE (PEEK(560)+256*PEEK(561)+23),
130:REM Set DLI on line 18 of display
270 POKE 512,0:POKE 513,6:POKE 54286,1
92:REM Enable display list interrupt
280 TRAP 300:? "Enter a Key: ";:INPUT
KEY
290 IF KEY>0 AND KEY<13 THEN TRAP 4000
0:GOTO 310
300 ? "ENTER A NUMBER FROM 1-12":? :G
OTO 280:REM ESC CTRL-2 FOR
310 POSITION 14,18:? #6;MENU1$(5*KEY-4
,5*KEY)
320 TRAP 340:? "Enter a Chord: ";:INPU
T CHORD
330 IF CHORD>0 AND CHORD<13 THEN TRAP
40000:GOTO 350
340 ? "ENTER A NUMBER FROM 1-12":? :G
OTO 320:REM ESC CTRL-2 FOR
350 POSITION 20,18:? #6;MENU2$(6*CHORD
-5,6*CHORD)
360 FOR I=4 TO 15:POSITION 6,I:? #6;"
":NEXT I
370 FOR I=1 TO 4:A=INTS((CHORD-1)*4+I)
+KEY-1:N(I)=PITCH(A):REM Choose notes
380 POSITION 6,3+A-12*(A>12)-12*(A>24)
:? #6;CHR$(159):REM Draw arrows
390 NEXT I
400 FOR I=0 TO 3:SOUND I,N(I+1),10,3:N
EXT I:REM Here is the chord
410 FOR I=1 TO 400:NEXT I:REM Play cho
rd for about 2 seconds
420 FOR I=0 TO 3:SOUND I,0,0,0:NEXT I:
REM Turn off sound channels
430 ? :? "PRESS RETURN TO GO ON, ESC T
O QUIT"
440 OPEN #1,4,0,"K:"
450 GET #1,A:IF A=27 THEN 480
460 IF A<>155 THEN 450
470 CLOSE #1:? :GOTO 280
480 GRAPHICS 0:END
```

TYPO TABLE

```
Variable checksum = 235644
   Line num range     Code    Length
   10      - 90        FL       518
   100     - 200       ZJ       507
   210     - 270       TY       516
   280     - 370       ZJ       563
   380     - 480       QI       509
```

RANDOM MUSIC



Can an ATARI make a melody?

by CHARLES JACKSON

Can a computer generate melodies? It can make a series of notes, just as a bird can sing without producing a melody, but can even a bird generate a "good" melody? While considering such abstruse points, another question to ask is why anyone in their right mind would want to make a computer write melodies. Music is a form of human expression! One answer is that by simulating melodies, we might gain insight into what makes a melody, and knowing this we might be able to write better melodies.

You will be pleased to know that without buying additional plug-ins or other expensive equipment, without requiring musical ability beyond an interest in learning, and without an understanding of higher mathematics or physics, the built-in sound functions of the ATARI make it possible to answer these questions.

This article investigates ways that your ATARI can generate simple melodies. First, some short examples will show you how to generate random melodies. You can insert these short examples into programs or games that you have written. The "melodies" generated this way sound like ones in popular commercial games, like **Protector** by Synapse Software. However, let's be realistic from the start; the sounds from random music generation will never spice up your programs or games as much as Jerry White's **Music Box** from Program Design, Inc.

Listening to the short examples provides ideas for a second, more detailed way to generate melodies. Surprising results can be made by the random music generation routines described here; the melodies will never make the top ten, but some sound as good as "dentist's office music." (This may or may not be a real achievement.) Some of the internal workings of the more

---

Charles Jackson is a recent graduate of UCLA who bought his ATARI 800 to help him with class work (reports, calculations, etc.). He also discovered it could help him play four-part harmonies.

detailed program (Listing 1) are described at the end of the article, but try to use the program and listen to the types of melodies that you can generate before worrying about the technical aspects of random number generation.

The ATARI generates sound by the use of the BASIC command:

```
SOUND A,B,C,D
```

where A is the channel or voice (0–3), B determines the pitch (B = 0 to 255), C sets the tone quality (try 10 and 14 for pure tones and other even numbers less than 16 for special effects — especially 6,8,12), and D sets the volume (10 is typical). The frequency can be changed by poking a pitch value (B = 0 to 255) into locations 53760 for channel 0, 53762 for channel 1, 53764 for channel 2, and 53766 for channel 3.

By using either the built-in random number generator located at location 53770 or the BASIC command RND(X), one simple statement allows the generation of random melodies. Location 53770 always contains some number between 0 and 255, so it can easily be used in the ATARI sound command. The ATARI BASIC command RND(X), where X is a number or a variable, returns a decimal fraction between zero and one. As shown below, for use in a sound command RND(X) must be multiplied by a number less than or equal to 255.

The two random number generators can be demonstrated in short one-line programs:

```
1 SOUND 0,255*RND(0),10,10:WAIT = 2^3:GOTO 1
1 SOUND 0,PEEK(53770),10,10:WAIT = 2^3:GOTO 1
```

Enter the short one-line programs and listen to the melody. WAIT = 2^3 is just a way to make the notes go slower — try removing it! While you are at it, try changing the value of C to the even numbers between 0 and 16. By the way, hit the [BREAK] key to end the program.

The SOUND commands take a lot of computer time to execute, so if you want to insert a bit of random music into your programs without making a funeral march, a faster way of changing notes is needed. All you have to do is to start a note and then POKE a new one every now and then. First, turn on the sound in the program by using a SOUND command, for example:

```
1 SOUND 0,1,10,10.
```

Then enter your program. Somewhere in your program, perhaps in a FOR/NEXT loop or a statement that is used often, insert the following line:

```
XXX POKE(53760),PEEK(53770)
```

The XXX is whatever statement number you decide on, the PEEK value returns a random number, and the POKE location is for the pitch of voice number 0. If the music is too fast try placing the statement elsewhere, and if it moves too slowly, try inserting another statement.

One problem with this random music is that it doesn't even use the notes of the scale; it could even be called noise. Maybe choosing random musical notes (like hitting the keys of a piano at random) would sound better than choosing random pitches. Try this:

```
1 F = (2^(INT(24*RND(0))/12))/512:B = (1–2*F)/(2*F):
  SOUND 0,B,10,10: GOTO 1
```

(the complicated expressions for the pitch B choose notes on a piano keyboard).

This isn't very melodious, but it is an improvement musically. Also, the intervals are just too big. Most melodies are meant to be sung, but not many people can sing a very low note followed by a very high one. Restricting the range of notes improves the sound of the melodies, as Listing 1 shows.

One way to circumvent the problem of large intervals is to choose the intervals at random, rather than choosing the pitches at random. This way of choosing numbers describes the motion of a small particle in a liquid, called Brownian motion. It is also called a random walk: take a step or two, stop, choose a new direction at random, and take a few steps. Keep on repeating this until you are convinced that you are moving at random. Brownian motion describes many things, such as the prices on the New York Stock Exchange, and it can also generate melodies called brown music.

Another problem with these simple melodies is that all of the note values are the same, so there is no rhythm. Varying the rhythm would help make a more musical sound, but it leads to so many interesting possibilities that I have to leave it for you to investigate.

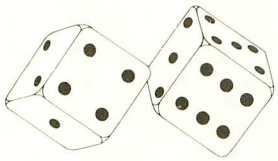
Finally, the melody goes nowhere. This is a fundamental problem with random music generation programs. Two physicists, Mandlebrot and Voss, have found that the way you generate random numbers can get around this fundamental problem. Mandlebrot and Voss call random music "white music" because a random series of colors is white to our eyes, so a random pattern of sounds should seem white to our ears. If you have a color TV set, turn it on to channel 3 with your ATARI turned off—notice the white (or grey) color? The static you hear is another form of white noise or music.

These two physicists from IBM have found that most music, from Ukrainian folk melodies to Indian sitar music, from classical music to rock 'n' roll, can be described as "1/F" (one-over-F) music. In their terms, 1/F music occurs between white music and brown music. 1/F music is easy for a computer to generate, as shown in Listing 1. Voss described a method for producing 1/F music in *Scientific American* in April 1978. While white music chooses pitches at random, and brown music chooses intervals at random, 1/F music chooses both pitches and intervals at random.

Listing 1 generates these types of music (white music, brown music, and 1/F music). The intervals can be modified and different scales can be heard by changing the DATA statements. You may wish to try using a different tempo (but it is hard to get each type of music to have the same tempo because the different subroutines take different times), or to modify the note durations or rhythms. The program shows a rough outline of the melody on the screen while the music is being played.

Even the 1/F music program is really quite short. To produce more complex computer-generated music would require

continued on page 53



RANDOM MUSIC *continued from page 51*

longer programs. One interesting example is the program for generating four-part harmony in the **Atari Music Composer** documentation. This program uses rules to assure good harmony, and it chooses notes at random until a good note turns up. (In other words, it generates white music and uses rules to make it more "musical".) It would be a large project, but using a 1/F note generator might make for a better sounding harmony.

After typing the program into your ATARI, try to decide which type of music is more musical. Can you tell one type of music from the other? Does the graph on the TV screen resemble the melody? Try the program on friends and see what they say. I'll leave these questions for you to answer — is the ATARI making music? Is it good music?

PROGRAM NOTES

The main features of the program are that the melody contour is displayed on the screen, and the tempo is made clearer with a click (or attack) generator. Finally, three types of scales can be used: the usual diatonic scale, in this case C major; the pentatonic scale, often used in folk melodies; and the harmonic scale, often called bugle notes. Pitches from the DATA statement with the lowest BASIC statement number will be played, so change the line numbers around if you want to hear different scales. Try your own scales, or even a short melody!

The 1/F generator was described in Martin Gardner's feature column in *Scientific American* in April of 1978. Imagine three colored dice — our program (Listing 1) uses seven. Initialize their values by rolling them all. Put them in order, blue, green, red. Give each color a binary digit. Let red be the units, green be the two's, and blue be the fours.

Keep count of the notes in binary form. When a binary digit changes from 0 to 1, or from 1 to 0, then roll the die associated with that digit. After each roll for a given note, add up the total of the dice to find which pitch is to be played.

Note #	Binary #	
0	000	Roll all three dice
1	001	Roll only red die
2	010	Roll green and red dice
3	011	Roll only red die
4	100	Roll all three dice
5	101	Roll only red die

The above example shows how 1/F music is in between white music and brown music. Sometimes (notes number 0 and 4) all the dice are rolled and the new *note* is random in the sense of white music, and other times (notes number 1 and 5) only one die is rolled and the *interval* is random in the sense of brown music.

In Listing 1, there are seven dice. DICE(I) holds the binary bit for each die calculated in statement 325. OLDICE(I) is the binary bit from the last note. When a bit changes between OLDICE and DICE, then that die is rolled (statement 330), and VALDICE(I) is changed. The total of all the dice (statement 340) determines the pitch from array P(I). After each type of music is played, a GOTO 4000 statement leads to the final statement, which returns to Graphics 0 and lists the program. This last statement should help in your debugging stages, and it provides a long note to end the melody.

Requires 16K RAM

```
0 REM PROGRAM LISTING ONE
1 REM ANTIC MAGAZINE
5 GRAPHICS 3
10 DIM P(16),DICE(7),OLDICE(7),VALDICE
(7)
15 COLOR 1
20 FOR I=1 TO 16:READ P:P(I)=P:NEXT I
25 ? "TYPE OF MUSIC":? "1=WHITE; 2= BR
OWN; 3=1/F":INPUT TYPE:ON TYPE GOTO 10
0,200,300
45 REM SOUND AND PLOT
50 SOUND 2,12,10,14
55 SOUND 3,P(PIT),10,10
60 SOUND 2,0,0,0
65 PLOT I-40*INT(I/40),18-PIT
70 FOR WT=1 TO WAIT:NEXT WT
75 IF (I-40*INT(I/40))=39 THEN GRAPHIC
S 3
80 RETURN
100 REM WHITE MUSIC
105 FOR I=1 TO 128
110 PIT=1+INT(16*RND(1))
120 ? "WHITE MUSIC",PIT
130 WAIT=100
140 GOSUB 50
150 NEXT I
160 GOTO 4000
200 REM BROWN MUSIC
205 FOR I=1 TO 128
210 PIT=PIT-2+INT(5*RND(1))
215 IF PIT<1 OR PIT>16 THEN PIT=10
220 ? "BROWN MUSIC",PIT
230 WAIT=105
240 GOSUB 50
250 NEXT I
260 GOTO 4000
300 REM 1/F MUSIC
305 FOR J=1 TO 7:VALDICE(J)=INT(3*RND(
1)):DICE(J)=0:OLDICE(J)=0:NEXT J
310 FOR I=1 TO 128
315 PIT=0:NUM=I
320 FOR J=1 TO 7
325 DICE(J)=NUM-2*(INT(NUM/2)):NUM=INT
(NUM/2)
```

continued on next page