PROGRAMMING SMALL COMPUTERS TO PRODUCE EXPERIMENTS IN MUSIC COGNITION

W. Jay Dowling
University of Texas at Dallas

This article offers suggestions for programming small computers (particularly the Commodore 64 and Amiga) to produce stimuli for experiments in music cognition. The main suggestions concern the overall structure of an experiment-producing program, the organization of pitch data in memory, the handling of time and timbre data, the organization of melody data, and randomizing aspects of the experiment.

The following article is based on several years experience programming small computers to produce stimuli and record responses for psychological experiments in music cognition (for example, the experiments reported by Dowling, Lung, & Herrbold, 1986). The most useful things I have to say amount to a cluster of helpful hints for the organization of such programs and the data upon which they operate. I have phrased them in a general way so that they can be adapted to a variety of computers. My most recent experience is with the Commodore 64 and Amiga 2000. Most of the suggestions I shall make apply to both machines, and hence span a wide range of size and speed of processing. I shall also include a few specific suggestions for those two machines.

Compared to the average desktop personal computer, the Commodore 64 is small and slow. However, it has sufficient capability to do interesting music cognition experiments. Its video-game quality sound generating chip is too noisy and imprecise to be suitable for precisely controlled psychoacoustic studies. However, the complex sounds it produces are sufficiently precise in pitch and time to be useful for studies in cognition, and the monophonic audio output can easily be directed to external amplifiers and tape recorders. The Commodore 64 can be programmed in its resident BASIC to produce experiments of complicated design. When BASIC is supplemented with an assembly language subroutine to control sound production, better than millisecond precision in temporal control and response time measurement is easily achieved, provided the video screen is blanked during stimulus production. (For developing the assembly language routine I recommend a monitor-assembler cartridge such as Human Engineered Software's HesMon64 plugged into the game port.) The Commodore 64 is an easy machine to use for experiments in which the computer interacts with the subject online in real time. Its durability and cost (in the under-$500 range) make it a good choice where precisely controlled pure tones are not required.

The Amiga, with its stereo, 8-bit digital-to-analog converters operating at 28 kHz sampling rates, is capable of producing much more precisely controlled sounds than the Commodore 64. In the under-$2000 range, it is among the least expensive machines taking advantage of the flexibility and speed of 68000-series processors. While it can be programmed in BASIC, sophisticated programming requires the use

of the C program language. Though the Amiga is capable of far more elaborate sounds and visual displays than the Commodore 64, it requires far more expertise and effort to program. In particular, it is very difficult to make it collect response time data online in conjunction with the production of sounds. The following suggestions assume that programmers have available one of the public domain software packages for the production of single tones defined by frequency, duration, and waveform, such as Audiotools by Rob Peck (updated version of the programs published in *Amiga World*, July-August, 1987).

The main suggestions I shall make concern the overall structure of an experiment-producing program, the organization of pitch data in memory, the handling of time and timbre data, the organization of melody data, and the randomization of aspects of the experiment.

Program Structure

There are several tasks the program must do, and it is best (especially in the long run) to organize those tasks into separate modules in the program. Some of those tasks, such as defining pitch parameters and initializing the sound generating device, will be done just once in the program. Some tasks will be nested within other tasks; for example, the program segment that passes parameters to the sound generator for each stimulus sequence will be nested within the program segment that plays a trial of the experiment, which in turn will be nested within the segment that plays all the trials. It is strongly recommended that each task be contained within a subroutine or "procedure" and called by the outer program segment as it is needed. This "structured" style of programming is very natural in the languages Pascal and C (the latter being the standard language for the Amiga), but even in BASIC it is very useful to proceed that way. The utility of the structured approach will become apparent as soon as you begin to modify the program for a *second* experiment. The outline of such a program for playing a series of trials in an experiment is shown in Table 1.

In this discussion it is assumed that the programmer has available an external subroutine that plays a note (or sequence of notes) on the sound generating device when the appropriate parameters are passed to it. The calls to that subroutine occur in the innermost loop of the program, in order to produce the notes of a particular stimulus. The outermost layer of the program initializes the sound device and sets up default values of the sound parameters (such as loudness and timbre, if those are not to be changed). Next it initializes the data sets to be used, reading them in from the disk or DATA statements if necessary. Then it is ready to play sample trials. In playing a trial the program retrieves the data describing the stimulus sequence to be played and translates it note by note from humanly accessible mnemonics to a parameter list to be passed to the sound generator. When the sample trials are done, the program proceeds to randomize the order of trials in the experiment proper, and then plays them using subroutines that translate the note-defining mnemonics into parameters to control the sound, and sends the parameters to the sound device. When the experiment is done the program turns off the sound generator.

Table 1
Outline of Program to Produce Stimuli for an Experiment in Music Cognition. (Note that loops are executed several times before going on.)

Initialize Sound Generator and Randomizer
Initialize Data Sets for Stimuli
    (scale, melodies, trial types, etc.)
Play Sample Trials (loop)
    Select Trial Type
    Call PLAYTRIAL
Randomize Experiment Trials
    Call PERMUTE
    Store Answer Key
Play Experiment Trials (loop)
    Select Trial Type from Randomized List
    Call PLAYTRIAL
Turn Off Sound Generator
Subroutines
PERMUTE (see Table 5)
PLAYTRIAL
    Select Stimulus
    Translate Mnemonics to Parameters
    Call PLAYNOTES
PLAYNOTES
    Pass Parameters to Sound Generator
    Start Sound

Organization of Pitch Data

A program like that diagrammed in Table 1 needs a way of translating pitches from a human-oriented mnemonic scheme to the pitch parameters the sound generator expects. The method I have been using wastes some memory, but is fast and convenient. I represent each pitch as a two-digit number which is used as an index of the array in which the pitch parameters are stored, as shown in Table 2. The first digit indicates the octave level, and the second digit indicates pitch level within the octave, beginning with C. Thus, if we have three octaves below middle-C, the white notes of the keyboard would be labeled 41 (C), 42 (D), 43 (E), etc., up to 47 (B). The pitch parameters for those white notes are stored in column 1 of the array, in the row indicated by the note label. The mnemonic for the black notes involves labeling flats by preceding the note labels with a minus sign: -42 (Db), -43 (Eb), -45 (Gb), -46 (Ab), -47 (Bb). The black-note parameters are located in column 2 of the array. When the program encounters a negative number as a note label it sets the column to 2 to retrieve the parameter. (For the Commodore 64, which uses two-byte pitch parameters, a three-dimensional array is employed, in which the third index indicates the high and low bytes of the parameter. Incidentally, use the *formula* in the Commodore 64 manuals, and don't trust the values they provide in tables.)

Table 2

Organization of Array of Pitch Parameters (C, D, E, etc.) Retrieved by Mnemonic Row Labels. Black Notes are Labeled with Negative Numbers, for which Parameters are Retrieved from Column 2 (Flats). Unused Row Labels are in Parentheses.

	Column	
Row Label	1 White Notes	2 Black Notes
11	C (32.75 Hz)	
12	D	Db
13	E	Eb
14	F	
15	G	Gb
16	A	Ab
17	B	Bb
(18)		
(19)		
(20)		
21	C (65.5 Hz)	
22	D	Db
...		
41	C (262 Hz)	
42	D	Db
...		

I find these mnemonics easy to remember, and especially easy to type into the machine, since numbers are easier to type than letters. Consider "Frere Jacques" in the key of Bb: -37, 41, 42, -37, etc., is much easier than Bb, C, D, Bb, etc. And typing "Mary Had a Little Lamb" in D (-45, 43, 42, 43, -45, -45, -45, etc.) is much, much easier than F#, E, D, E, F#, F#, F#, etc., once you get used to using -45 (Gb) for F#.

In some experiments I have used quarter steps, dividing the octave into 24 logarithmic steps using the twenty-fourth root of two. In that case the pitch parameters can be conveniently arranged in an array with four columns as shown in Table 3, in which I have used the plus sign to indicate the quarter-step above a pitch—F+ for the quarter-step between F and F#, for example. Now the mnemonic labels require three digits, with the third digit indicating the column. Middle C would be indicated by 410, C+ by 411, C# by 412, C#+ by 413, D by 420, etc. The program decodes the third digit to retrieve the column number. It is usually convenient to preserve the foregoing mnemonic for Db and C#, so the program

should retain the capability to decode negative numbers as indicating flats; for example, it should treat -42 the same as 412 in order to retrieve that note's parameter (C# or Db). In so doing, it needs to aim at the third column and go up one row (from 42) to find the correct parameter.

Especially with the Amiga—because of its speed and because its pitch parameters are simply waveform frequencies or periods—it is convenient to have the program calculate the contents of the parameter array as part of its initialization procedures. With the Commodore 64 it is usually more convenient to read the array in from the floppy disk during initialization. In either case it is easy to change the tuning standard on which the experiment is based with only minor alterations in the program, and that is often useful for purposes of experimental control. For example, when dealing with perceptual differences between quarter-steps and semitones or diatonic pitches, it is often a good idea to run a condition in which the stimuli are tuned not to A = 440 Hz, but a quarter-step away, so that what was a semitone is now a quarter-step and vice-versa. Thus if some equipment malfunction were slighting pitches that were quarter-steps, in the new tuning those pitches would fall on standard semitones, and the former semitones would now be quarter-steps.

Table 3

Organization of Array of Pitch Parameters (C, D, E, etc.) Retrieved by Mnemonic Row Labels. Black-Note Parameters are in Column 2; Parameters for Quarter-Steps are in Columns 1 and 3. The Third Digit of the Mnemonic Label (here shown as x) Indicates Column Number. Quarter-Step Parameters are Indicated with +.

	Column			
Row Label	0 White Notes	1 Quarter Steps	2 Black Notes	3 Quarter Steps
11x	C	C+	C#	C#+
12x	D	D+	D#	D#+
13x	E	E+		
14x	F	F+	F#	F#+
15x	G	G+	G#	G#+
16x	A	A+	A#	A#+
17x	B	B+		
(18x)				
(19x)				
(20x)				
21x	C	C+	C#	C#+
etc.				

Organization of Time and Timbre Data

For me, the most convenient way to organize time data in melodies is to pick some modulus (such as 166 ms) and designate note values in multiples of it. For example, if stimuli are presented at a rate of six eighth-note values per second, then the modulus would be 166 ms and an eighth note would be represented by 1, a quarter note by 2, etc. The program would translate those numbers into the time parameters sent to the sound generator, usually millisecond timings.

The program usually has the task of inserting silences for purposes of articulation between the notes. Thus if the experiment calls for gaps of 20 ms between notes, then with a 166 ms modulus a 1 is translated into a note 146 ms long followed by 20 ms of silence, a 2 into a 312 ms note plus 20 ms silence, etc. This process is illustrated in Table 4 for the start of "Mary Had a Little Lamb" in the key of D. The first row contains the melody expressed as a series of pairs of mnemonic pitch and time parameters, one pair per note. The second row shows the translation into pitch parameters from the array in Table 2 (represented as note names) and time parameters in milliseconds.

Similarly, a timbre parameter can be added to the pairs in Table 4, usually in the form of a number designating the waveform to be used in output.

Organization of Melody Data

As suggested by Table 4, data for a melody can be organized as pairs, triples, etc., of parameters, depending on how many aspects of each note need to be specified. For example, one could specify the pitch, duration, timbre, loudness, and (for the Amiga) the stereo channel for each note. The mnemonics for the *m* notes of *l* melodies can be stored in an l x m x n array, where *n* is the number of parameters to be specified for each note. As indicated in Table 1, the program selects a melody (by specifying the row number in that array) and then translates the mnemonics into parameters to be passed to the sound generator. The melody set can either be read in initially from a disk file, or (more usually) initialized when the array is declared (Amiga) or listed in DATA statements (Commodore 64).

Table 4

Illustration of the Translation of Pitch and Time Mnemonics (Row 1) for "Mary Had a Little Lamb" in D Major into Pitch and Time Parameters (Row 2) to be Sent to the Sound Generator. Pitch Parameters are as in Table 2; Time Parameters are in Milliseconds. "0" Indicates Silence.

-45,2, 43,2, 42,2, 43,2, -45,2, -45,2, -45,4
Gb,312,0,20, E,312,0,20, D,312,0,20, E,312,0,20, Gb,312,0,20 Gb,312,0,20, Gb,644,0,20

Table 5
Outline of General Purpose Routine for Producing Random Permutations of N Numbers. (Array subscripts are indicated within square brackets[].)

PERMUTE

Enter with LIMIT = size of permuted array.
Returns array PERM containing a pseudorandom
    permutation of LIMIT numbers.

Fill beginning of array NUMBERS with integers 1 through LIMIT.
For i = 1 to LIMIT - 1; get a random number x between 1 and LIMIT; check to see
    if it's been used (i.e., is NUMBERS[x] negative?); if x hasn't been used then
    let PERM[i] = x, and NUMBERS[x] = NUMBERS[x] * -1.
Now we have PERM filled from 1 to LIMIT - 1, so we need to find the last number
    in the list. All the ones that have been used in NUMBERS are negative.
For i = 1 to LIMIT, test NUMBERS[i] to find the one that's still positive; put that
    number in PERM[LIMIT].
Return.

Randomization in Experiments

Good experimental control requires that variables at several levels in the design need to be scrambled so as to be unpredictable to the subjects. Thus strict randomization is not necessary, and the pseudorandom number routines that come with BASIC and C are entirely satisfactory. Generally variables are randomized within some constraints; for example, if there are eight trial types in a particular experiment, and we want to present each trial type ten times, then we might construct a trial list consisting of ten successive random permutations of the eight trial types. For this purpose a permutation subroutine such as that outlined in Table 5 is useful. Note that that subroutine picks all but the last number in the list randomly, and then hunts to see which number it hasn't used yet. For permuting long lists, especially on the rather slow Commodore 64, we may want it to switch its procedure before the very last item—say for selecting the last three or four.

The permutation subroutine is useful for randomizing other aspects of the experiment as well, and that is why it is given in Table 5 in a general form that adapts itself to permuted lists of varying length. For example, in my experiments I often intersperse randomly arranged distractor notes among the notes of the target melody. The distractor notes are selected in random order from a list of pitches. Suppose that I want to use the notes C, D, and E, and that each is to appear twice in every set of six consecutive distractors. In that case the list to be permuted would consist of two of each note: C, C, D, D, E, E; and the PERMUTE subroutine would be called to arrange the order of each six distractors.

Reference

Dowling, W. J., Lung, K. M.T., & Herrbold, S. (1987). Aiming attention in pitch and time in the perception of interleaved melodies. *Perception & Psychophysics*, *41*, 642-656.

Author Note

Requests for reprints should be sent to Dr. W. Jay Dowling, Program in Human Development and Communication Sciences, Unversity of Texas at Dallas, Richardson, TX 75083-0688.