HMSL Intonation Environment

by Larry Polansky

I. INTRODUCTION
This article describes the experimental intonation facilities available in the HMSL (Hierarchical Music Specification Language) (Version 2.0) programming environment developed by Phil Burk, David Rosenboom and this writer at the Mills College Center for Contemporary Music. HMSL runs on both the Commodore Amiga and the Apple Macintosh (as well as on an S-100 68000 based prototype at the CCM). Since the Macintosh version supports only MIDI-based sound synthesis, this article deals mainly with the use of HMSL to drive the tunable "local sound" capabilities of the Amiga.

Affordable MIDI devices are becoming available that allow the specification of polyphonic tuning environments (e.g. the Yamaha FB-01). The software-based tuning procedures described in this article are directly applicable to the Macintosh version of HMSL as well. There is currently support of the FB-01 by HMSL drivers, and we will try and support other standard devices as they appear. All of these drivers will use the concepts, described below, of translators, tunings, and tuning-ratios.

II. THE HMSL PROGRAMMING ENVIRONMENT
HMSL is a real-time performance and compositional environment written in an object-oriented version of FORTH (developed by Phil Burk). Thus, in HMSL there are actually three co-existent and interacting software environments available to the composer and programmer: 1) FORTH, 2) Object Oriented Programming Support (OOPS), and 3) HMSL. The user or programmer may use any part of FORTH in an HMSL program, or extend HMSL itself via the OOPS. This article focuses on the use of the Amiga's local sound for tuning purposes. The main ideas of HMSL are documented elsewhere (see references), as are the main concepts of object-oriented programming, to which HMSL more or less fully conforms.

First a small apologia. HMSL was not designed primarily for experimental intonation, but for formal and perceptual experimentation in the context of a real-time generalized stimulus-response environment. Because of the programming power of HMSL's object-oriented tools, and because of the useful and rather powerful local sound capabilities of the machine, the tuning routines in the Amiga version are, quite candidly, a result of the ease with which they can be implemented. Tunable MIDI equipment has only recently become available, and HMSL has been mainly targeted for MIDI users. Due to the secondary importance of tuning in this environment, and the need to focus development time on the implementation of larger scale compositional and performance tools, intonational tools are limited to a scale or "gamut" based environment, rather than the "dynamic" or "paratactical" tuning systems I have advocated elsewhere. (Polansky, 1987; Polansky, 1985(b)). Future versions of the language will further integrate ideas of experimental intonation with the HMSL-based ideas of perceptual distance functions, dynamic formal hierarchies, and generalized stimulus-response definitions.

III. AMIGA SOUND SYNTHESIS
The Amiga supports 4 voices of arbitrarily-specifiable 8-bit waveforms under a hardware multiprocessing environment. This means that the programmer only needs to specify certain parameters of the sound, and the system takes care of updating the digital-to-analog converters without tying up the CPU. The parameters that may be specified are: period, loudness, starting address of waveform table, length of waveform table, and whether or not the selected voice amplitude or frequency modulates another voice.

What follows in this section is not meant as a complete description of the Amiga sound capability (see the Amiga ROM Kernel and Hardware Manuals for that), but rather as one composer's notes on various uses of that capability.

The current version of HMSL implements the Amiga sound facilities in the simplest possible manner, by writing values directly to the sound co-processor chip, or, more specifically, the memory addresses, referred to as registers, from which the co-processor obtains values for the sound. Although this way of doing things is fast, simple, and very easy to modify, it does not take full advantage of the Amiga Rom Kernel Operating system, which provides device drivers that allow programmers to use the hardware fully without risking multitasking conflicts. So far, we have not experienced any difficulties in doing things this way (in fact, it seems to be the most common way for programmers to use the sound hardware!). However, a later revision of the system will most likely use the Amiga system calls more fully.

For a given voice (or channel), loudness is specified by a number between 0 and 64. Period is more complex, since it is of course more usual to specify frequency. The value for period tells the sound channel hardware how fast to read through the currently assigned waveform table. Thus, the specified value for period is the inverse of the sampling rate. The length of the waveform (number of memory bytes) and the value for period combine to determine the frequency heard. For example, doubling either the length of the waveform table or the period halves the perceived frequency.

There are at least two main ways of specifying steady-state timbre, or waveform. The first, which is sampling (more or less), is to specify or capture a sound event in terms of all iterations of the waveform. A large amount of memory is used for this technique, and the perceived pitch of the sound is largely determined by the actual pattern repetition of the stored sample. This is the technique used by most commercial music packages for the Amiga.

The second technique is to specify only one iteration of the waveform, and use the period register to alter the frequency. A very nice and sonically powerful feature is the ability to specify any memory location and any memory length for the waveform table. For example, one could specify the top of the FORTH stack as the beginning of the table, and since the DAC's are refreshed by the parallel processor, listen to the dynamically changing waveform for debugging purposes. Interesting relationships of phase, tuning, timbre, and modulation can be created between voices by dynamically altering these starting addresses and table lengths.

There are limits to the numerical value that can be assigned to the period. The values themselves are not important here, since their perceptual result is directly related to the size of the waveform table. Note that the upper pitch limit depends directly upon the waveform table length, and for rather long tables (longer than 64 bytes for one iteration of a waveform), that upper frequency limit can be restrictive for most uses. This is, of course, not the case with sampling techniques.

One important consideration in using the Amiga is that frequency resolution decreases exponentially, and inversely to period or directly to frequency. For example, given a constant size waveform table (usable values, regrettably, tend to be more or less between 8 and 64 bytes), there are 2000 discrete frequency values between periods of 4000 and 2000 (with 4000 producing a lower frequency), but only 250 discrete values between 500 and 250, 3 octaves higher. This can be confusing, since it rather directly contradicts the usual perceptual relationships of resolution, pitch discrimination, and frequency. However, for most reasonably-sized waveform tables, and especially for the mid- to low- frequencies, the resolution tends to be within a cent.

HMSL contains simple FORTH drivers for all sound functions. These routines are easy to use inside of FORTH or HMSL routines. For instance, the routine called CRECENDI.RANDOM.PITCHES (EXAMPLE 1, page 10), will produce a sequence of crescendi on random pitches in channel 0. Note that this is not really an HMSL program, but a FORTH routine that uses the HMSL internal sound driver primitives.

These simple sound drivers are "below" the object-oriented environment, but serve as basic routines available to the composer for experiment.

Since the sound hardware in the Amiga is "write-only," HMSL keeps a memory image of current parameters for all four channels. This allows the programmer to "read" the period, loudness, (DA.PERIOD@@, DA.LOUDNESS@@) or other parameters at any given time.

Other important aspects of the sound hardware relate mainly to timbre. There is a 7k low-pass filter on each channel of the stereo output, with a rather steep slope. This value is conservative, and it is preferable to disable it and replace it with a programmable or analog filter. I have done this, with the help of Dave Lucas and Greg Keller of Amiga, and will soon publish the method. A higher cutoff frequency gives another octave or so of partials for most frequencies, and greatly improves the quality of the 8-bit sound. There are several other techniques for increasing the timbral resolution, for example, "ganging" together the amplitude modulation channel and the carrier to effectively produce a waveform with higher resolution. Since the loudness resolution is only 6 bits, and since this "ganging" is multiplicative of one channel's waveform value by the other's loudness, the resultant resolution would be around 12 bits. This, like the use of an AM channel for envelopes, effectively reduces the sound capabilities from four simple voices to two complex ones. This technique, unlike sampling, is memory efficient and offers the programmer powerful modulation possibilities.

IV. HMSL TUNING PROCEDURES

HMSL's support of the Amiga's local sound, written by programmer Phil Burk, is based on an instrument concept. It fully supports the use of samples, user defined envelopes, experimental tunings, and arbitrary waveform definition. These routines are high-level object-oriented code, so the following section assumes the reader's familiarity with the basics of that type of programming. The following description, however, should be comprehensible even if one has not had experience with object-oriented programming.

Instruments are a class in the system with instance variables for tuning (that is, the address of a pre-defined scale), envelope, channel#, period, and waveform. These instance-variables are actually references to other objects, and the instrument itself could be thought of as simply a collection of parameters. Each user-defined instance of the class of objects called Instruments has built-in intelligent procedures, called methods, for handling these parameters. For example, one can inspect or change envelopes, samples, waveforms, tunings, etc., very simply (and of course in real time) in the object-oriented code. Although one can play an instrument one "note" at a time, these instruments are designed to be used in the hierarchical environment of HMSL as a virtual device, similar to process for sending MIDI information. As such, the instruments can also be used to execute the complex hierarchical and morphological data results of HMSL's compositional engines.

The relationship between instruments and tunings is particularly relevant. Each instance of the class of instruments (that is, each user-defined instrument) "knows" what its current tuning is. The user can change the tuning quickly with code like the following: PENTATONIC-SLENDRO PUT.TUNING: MY-INSTRUMENT This places the predefined PENTATONIC-SLENDRO "in" the instrument called MY-INSTRUMENT. This is done in software, so algorithms can easily be written which might, for example, change the tuning of an instrument in response to a given stimulus (like a keystroke, or the input from an analog-to-digital converter).

The tuning itself can easily be changed in real-time. For instance, if one wanted two different ratios from 1/1 for the sixth degree of a scale depending on its context, one could sense a stimulus (e.g. which other keys were activated), and update the actual tuning array quickly. In a sense, this would be a software implementation of Harold Waage's innovative logic circuit for context-dependent intonation; but it is a more general method because the contexts, stimuli, responses, and possible intonations are all user defined and easily altered.

Envelopes are 2-dimensional arrays with instance variables for sustain point and time values. These shapes can be changed by the user either graphically, or point-by-point in software via pre-defined methods. The details of their use is not particularly germane to this discussion of the tuning environment.

Waveforms are a class of objects which incorporate the notion of samples or computed waveforms. Their use is highly flexible, and they can be updated quickly. They are "file" oriented, in a special HMSL format, and can be read to or written from, or edited easily, via the HMSL shape editor (not described in this article).

Translators are a class of objects which must be described before discussing tunings. Translators are general operators which convert from one numeric system to another. In our case, their main use might be to convert a set of values from one gamut to another. They are based on the idea of a translation table, which has associated with it an offset and a modulus. The offset might be

(text continued on page 10)

(Tritriadic...continued from page 1)
identical to each other, and thus, there are only 18 distinct tritriadic scales derivable from a given triad. These are shown in Table 4 (page 8).

The three primary types of tritriadic scale have been plotted on the d x m plane to produce the tonal lattices of Figures 1 through 6 (centerfold). As may be seen, each consists of three triads of the PRIME type and two of the CONJUGATE. If m=5/4 and d=3/2, then Figures 1 and 2 are the major and natural minor modes, respectively. Because the RETROGRADE and INVERSE scale forms are related to the PRIME and CONJUGATE forms through circular (modal) permutation, and this appears as a simple translation (transposition) of the lattice,the PRIME (P) and CONJUGATE (C) forms have been plotted. Hence, there would appear to be only six fundamental scales from which the rest may be derived by transposition and permutation.

Moreover, the M->T and D->M types are related in an interesting fashion through the CONJUGATE (d->d/m) transformation. (Figures 3, 4, 5, and 6) These types have a novel, fresh sound, harmonically and melodically, even when constructed of familiar 4:5:6 triads. Furthermore, the two Order 2, D->M scales are tetrachordal when d=3/2 and 9/8<m<4/3, and this affords a means of harmonizing the Dorian Mode of such tetrachordal scales where x/y x 9/8 x z/w = 4/3, m or d/m = 9*x/8*y or d/m = 9*z/8*w.

The entire set of 18 tritriadic scales with 1/1 as the common tonic requires only 19 tones (Figure 7), although the six scalar lattice-forms can be aggregated through shared tones so that 12 pitches suffice to demonstrate the existence of each type. This 19-tone array would seem to be the minimally complete tonal field for harmony derived from the generalized triad T:M:D.

Table 1.

DERIVED FORMS OF THE TRIAD T:M:D

ORDER	PRIME	CONJUGATE	INVERSE	RETROGRADE
1.	T:M:D	M*T:D*T:D*M	D*M:D*T:M*T	D:M:T
2.	T:D:M	D*T:M*T:D*M	D*M:M*T:D*T	M:D:T
3.	D:T:M	D*T:D*M:M*T	M*T:D*M:D*T	M:T:D

Table 2.

DERIVED FORMS OF THE TRIAD 4:5:6

ORDER	PRIME	CONJUGATE	INVERSE	RETROGRADE
1.	4:5:6	20:24:30	30:24:20	6:5:4
2.	4:6:5	24:20:30	30:20:24	5:6:4
3.	6:4:5	24:30:20	20:30:24	5:4:6

Table 3.

TYPES OF TRIADIC MATRICES[1,2,3]

FUNCTION	D->T	M->T	D->M
SUBDOMINANT	2/d m/d 2/1	2/m 2/1 d/m	m/d m^2/d m
TONIC	1/1 m d	1/1 m d	1/1 m d
DOMINANT	d d*m d^2	m m^2 m*d	d/m d d^2/m

1. The T->D, T->M, and M->D matrices exchange the "dominant" and "subdominant" functions, but comprise the same tones and lattice.

2. All twelve of the M->T matrices are identical to forms of the D->T.

3. Only six of the D->T matrices are distinct.

(HMSL...continued from page 6)
used, for example, to specify the lowest note on a given MIDI keyboard, so that any index passed to the translator would be converted to a meaningful value for the output device. The modulus specifies the value above which the translator will repeat its pattern. For example, a modulus of 12 and an offset of 36 would allow the user to define standard scales for a MIDI device. A simple routine, called TR.SET.KEY (EXAMPLE 2), accepts as parameters the offset, scale values, and number of values in a scale and puts them in a translator called TR-CURRENT-KEY.

The default modulus for translators is 12, so TR.SET.KEY does not ask for a modulus. This can be changed and inspected with the PUT.MODULUS: and GET.MODULUS: methods.

Since the modulus, offset, and translation method (which could be quite complex) are all user defined, complicated data transformations are possible using these simple methods. This becomes especially relevant for changing scales or gamuts, where, for example, the modulus might be the number of pitches per octave, and the translation might be a non-simple function of indices (that is, the scale degree) into frequency values. Like all of these object-oriented sets of routines, "translators" are easy to learn and use, but are surprisingly powerful in their applications.

The class of objects called tunings are actually a subclass of translators. In object-oriented programming, this means that the subclass inherits from its parent class (called the superclass) all methods and instance variables (which might be thought of as "hidden data" specific to each instance of that class). For example, since the class translators has a method called GET.MODULUS: which returns the current modulus for any

```
EXAMPLE 1
( A simple example using HMSL Amiga sound drivers)

: CRESCENDI.RANDOM.PITCHES
0 DA.CHANNEL! (select current channel as 0 )
     ( The prefix DA stands for Digital Audio )
DA.START ( start the channel )
BEGIN
     4000 128 WCHOOSE  ( pick a random number between 4000 and 128)
     DA.PERIOD! ( set period of channel 0 to  the random number )
     DA.LOUDNESS@     ( get current loudness of channel )
     1+ 64 MOD ( increase the value until the peak is reached, and then
         back to 0 )
     DA.LOUDNESS! ( change the loudness of the channel )
     500 200 WCHOOSE ( pick a random number between 500 and 200 )
     MSEC ( delay by that number of milliseconds )
     ?TERMINAL ( test for keystroke to indicate end of BEGIN ...UNTIL)
UNTIL
DA.STOP ( stop the channel )
;
```

```
EXAMPLE 2
( Put the mixolydian mode into the variable TR-CURRENT-KEY )
: TR.MIXOLYDIAN.MODE
     36 ( offset, bottom note for CZ-101 ...)
     10 9 7 5 4 2 0 ( Mixolydian scale values in descending order )
                   ( FORTH is RPN....)
     7    ( number of values in scale )
     TR.SET.KEY ( sets TR-CURRENT-KEY )
```

instance of that class, so does every instance of the class tunings. (Once again, the reader is referred to standard references on object-oriented programming for more information on these ideas.

The TRANSLATE: method for tunings is slightly simpler than for translators. For tunings, no modulus or offset is used, so that an absolute frequency or period is generated from an index. Whereas translators "wrap" around their moduli, tunings "clip" at their own limit (or the number of values in the table). One of the design motivations for this was to allow the possibility of non-octave-replicating tunings. Since "tunings" contain actual frequency or period values, it is easy to fill a tuning for several octaves by simply multiplying by two.

Either a tuning or a translator can be "put" to an instrument, and the instrument "knows" which it has. This is a

```
EXAMPLE 3
( Two simple TUNING.RATIO definitions )
OB.TUNING.RATIOS RATIOS-CHALMERS-NEUTRAL
OB.TUNING.RATIOS RATIOS-OVERTONE
    ( define two instances of the class TUNING.RATIOS )
: RAT.INIT (  --- , Initialize tuning systems )
    7 NEW: RATIOS-CHALMERS-NEUTRAL ( 7 scale values for this one)
    ( scale: 1/1, 24/23, 12/11, 4/3, 3/2, 36/23, 18/11 )
          1 1 ADD: RATIOS-CHALMERS-NEUTRAL ( start adding in ratio
                                             values )
          24 23 ADD: RATIOS-CHALMERS-NEUTRAL
          12 11 ADD: RATIOS-CHALMERS-NEUTRAL
          4 3 ADD: RATIOS-CHALMERS-NEUTRAL
          3 2 ADD: RATIOS-CHALMERS-NEUTRAL
          36 23 ADD: RATIOS-CHALMERS-NEUTRAL
          18 11  ADD: RATIOS-CHALMERS-NEUTRAL

    12 NEW: RATIOS-OVERTONE ( 12 values for this one )
          12 0 DO ( this one will use a DO...LOOP )
              I 12 + 12 ADD: RATIOS-OVERTONE
                  ( this adds 12/12, 13/12, 14/12, ...
                  23/12 to the array )
          LOOP
```

```
EXAMPLE 4
( use of RATIO* to fill TUNING.RATIOS with sequential intervals, or
intervals specified by ratio to intervals other than  1/1)

2 GET: MY-RATIO ( returns contents of second place in tuning, 8 7 )

11 9 RATIO* ( multiply the two ratios together )

3 PUT: MY-RATIO ( put the resultant ratio in the third place )
```

good example of the power of object-oriented programming. The name of the method used to obtain a value for tunings and translators is TRANSLATE: In both cases, the same parameter protocol exists for each: the user specifies a note index, and TRANSLATE: returns a value of a table. Thus, the instrument, to get the appropriate value, in its PLAY: method, only has to get the address of the object stored in its tuning instance variable. This might be either a tuning or a translator, and the TRANSLATE: method acts differently depending on which class it is acting upon. This means that even though the definition of the TRANSLATE: method is crucial to these classes, the user rarely accesses it directly. Rather, the instrument's PLAY: method uses TRANSLATE:, which automatically "adjusts" itself to the class it TRANSLATE:'s.

Finally, a subclass of tunings is defined, called tuning.ratios. Tuning.ratios are the basic HMSL/Amiga Just intonation engine. This class has one added instance variable, called OB-RAT-1/1 (pronounced "object ratio 1/1"), which specifies the fundamental to which a ratio is tuned. There are PUT.1/1: and GET.1/1: methods for this class, as well as a new TRANSLATE: method. Tuning.ratios, unlike translators and tunings are defined as two-dimensional arrays. The translate method, as in both of the superclasses, takes an index and returns a value, but in the tuning.ratios class it computes the value for period or frequency using the value for the 1/1 and the two elements of the indexed value of the 2-dimensional array.

Unlike tunings, which simply "clip" indices greater than the length of the tuning, the TRANSLATE: method for tuning.ratios is, by default, octave replicating (that is, the modulus is assumed to be 2*(1/1)). However, that modulus is not used by the TRANSLATE: method for this class unless one specifies an index greater than the number of ratios specified. Thus, one can fill a tuning with ratios at any interval (3/1 for example), and they will be interpreted correctly. The modulus function for tuning.ratios is thus octave replicating if the user does not want to specify distinct tunings for different octaves of a gamut; it will be "overridden" by a more detailed multi-octave scale.

For anyone who has ever written a simple ratio-to-frequency conversion utility in FORTH (which is trivial, using "*/"), the definition of the TRANSLATE: method for tuning.ratios should be easy to understand. However, the flexibility gained by the separation of "intelligence" between tuning.ratios and instruments is significant. One can "put" a tuning.ratio into an instrument just as with the superclasses, and the appropriate version of TRANSLATE: will be used automatically, but now one can also change the value for the fundamental and individual ratio values independently of other instrument parameters, and even exchange tuning.ratios in an instrument rapidly.

EXAMPLE 3 (page 11) is a FORTH/HMSL definition of two simple tuning.ratios. One of these, RATIOS-OVERTONE, is currently defined in HMSL (as an example). The second is a scale, called the neutral diatonic, suggested by John Chalmers, that was generated on the Amiga (using a simple "noodling" program developed by John Chalmers and this writer), and used for a KPFA show on experimental tunings in August, 1986. I include these to illustrate the ease with which and the manner in which these are constructed. (Note for FORTH programmers: The ADD: method takes two numbers off the stack, and places them into the next two-dimensional cell in the tuning.ratio array.)

It is also possible to specify ratios in a "sequential" fashion, using the utility RATIO*, which multiplies two ratios together to produce an "absolute" ratio. For example, given the ratio 8/7 as the second in the TUNING.RATIOS object, MY-RATIOS, to specify that the third ratio in the array is an 11/9 above the second, the code in EXAMPLE 4 (page 11) will suffice.

This may be used not only for adjacent ratios, but for any interval in a gamut, since any index can be specified in the TUNING.RATIO for the PUT: or GET: method. It should be simple to see, for example, how RATIO* could be used in conjunction with the ADD: method (see example 2 above) to fill a scale with successive ratios rather than with ratios to the 1/1. Perhaps more interestingly, one could continually modify indirect intervallic relationships in a gamut, using the RATIO* and the PUT: and GET: methods, producing a dynamic, real-time intonational environment.

V. CONCLUSIONS

HMSL is a programming environment which includes facilities for experimental intonation. These facilities are designed to be user extensible. The advantage is that HMSL is quite flexible and powerful. The disadvantage is that some programming is required to make use of it fully; it's not a "turnkey" system.

The planned release date for HMSL (Version 3.0 for the Macintosh and Amiga) is around February 1, 1987 (although Version 2.0, a pre-release, is already in use by several composers, including this author). Price and terms will be announced soon. For information, write to: Larry Polansky, Center for Contemporary Music, Oakland, CA 95613 with questions or to be put on a mailing list for release information. To use HMSL on the Amiga, the user must first purchase a copy of JFORTH (Delta Research, 4054 Wilkie Way, Palo Alto, CA 94306). On the Macintosh, the user must first purchase MACH-2 FORTH, available from Palo Alto Shipping Co., P.O. Box 7430, Menlo Park, CA 94026, 800-44F-ORTH.

REFERENCES

BYTE Magazine, August 1986, Vol.11, No. 8, special issue on object-oriented languages. Two articles in this issue are of special relevance to this article: "Object-Oriented FORTH" by Dick Pountain, and "Elements of Object-Oriented Programming" by Geoffrey Pascoe

Burk, Phil, and Polansky, Larry, HMSL Source Code, Version 2.0, CCM in-house document, update to be distributed with HMSL Version 3.0 release.

Burk, Phil; Polansky, Larry, and Hays, Dorothy; HMSL Programmer's Manual, currently CCM in-house document; to be distributed with Version 3.0

Peck, Robert; Deyl, Susan; and Miner, Jay; Amiga Hardware Manual, Commodore Business Machines, West Chester, PA., 1985

Peck, Robert; Sassenrath, Carl; and Deyl, Susan; Amiga ROM Kernel Manual, Volume 1 and Volume 2, Commodore Business Machines, West Chester PA, 1985

Polansky, Larry, and Rosenboom, David; "HMSL (Hierarchal Music Specification Language): A Real-Time Environment for Formal, Perceptual and Compositional Experimentation"; Proceedings of the International Computer Music Conference, Vancouver, Canada; edited by Barry Truax, 1985, available from the Computer Music Association

Polansky, Larry; "Confessions of a Lousy Carpenter: Some Thoughts on Composing for Standard Instruments in Just Intonation", in 1/1, Vol. 1, Number 1, Winter 1985

(Text continued on page 15)

Index To 1/1 Volume Two

(Tutorial...continued from page 14)
play the said triad in tune must likewise be part of said scale. However, so long as you play the 10/9 where harmony requires it, you may call it what you will.

It should be noted that none of the ideas presented in this tutorial originate with this writer. All of the above material is present, implicitly or explicitly, in Alexander Ellis' excellent appendices to Helmholtz's On the Sensations of Tone. Unfortunately, most composers and theorists to date have failed to grasp the significance of this material (presuming that they studied it), thereby unduly retarding our harmonic evolution.

(HMSL...continued from page 12)

Polansky, Larry; "Paratactical Tuning: A Suggested Agenda for the Use of Computers in Experimental Intonation," Computer Music Journal, forthcoming, 1987

Waage, Harold M.; "The Intelligent Keyboard," in 1/1, Vol. 1, Number 4, Autumn, 1985