Real-time Implementation of Pitch-class Set Operations in HMSL

By

Jonathan Jared Turner

Computer Project Fulfilling Language Requirement
for the Degree of
Doctor of Philosophy
Department of Composition

Eastman School of Music
of the
University of Rochester

June 20, 1991
Allan Schindler, Computer Project Advisor

ENDORSED BY:
I. Hardware and Software--MIDI, Macintosh, and Extended Forth.

In the last decade, the Musical Instrument Digital Interface (MIDI) protocol has fostered the availability of a variety of software and hardware dedicated to the automated production of musical sound in real-time, by allowing a standardized method for computers and music synthesizers to communicate with each other. For example, the most common computer software, known generically as a sequencer, allows the computer to become a tapeless tape-recorder: typically, the keystrokes of a human performer on a synthesizer keyboard are recorded in the computer's memory, along with the time that each keystroke occurs; afterwards, the computer can send the same sequence of keystrokes back to the synthesizer, thus reproducing the original performance. Sequencer software also permits a variety of editing and reorganizing methods not available with audio tape-recorders, but analogous to the editing capabilities of word-processing programs. Even the sounds of many synthesizers can be recorded and edited with special software applications known generically as librarians. Most sequencer programs provide simple commands or tools to correct mistakes in timing, note, loudness, and duration, as well as change tempo or transpose. Additionally, all major software for editing and printing music notation allows MIDI input and/or output to some degree.

In order to create application programs like sequencers, librarians, or MIDI-oriented notation-processors, the major programming languages (C, Pascal, etc.) now support MIDI input and output. One of the most powerful languages for real-time computer applications is Forth, and a special version, Hierarchical Music Specification Language (HMSL), has been created by Phil Burk,
Larry Polansky, and David Rosenboom to provide a variety of MIDI capabilities on Macintosh and Amiga platforms. These authors describe the language and its capabilities in the introduction of the reference manual:

HMSL is a programming language and software environment for experimental music composition. HMSL provides...a language for creating and executing musical data, processes, and functions in complex hierarchical forms...a polyphonic real-time musical environment which allows for dynamic alteration of any of its functions, self-modification, and a high degree of user-machine interaction.

As a composer interested in exploring algorithmic compositional methods in real-time, and as a programmer already familiar with the Forth language, HMSL provides an excellent starting point. In several of my classes at the Eastman School of Music, I have written a number of Forth programs supporting either analysis or composition, and some of this work is included in the present project. In the first part of this paper, I will describe how I have implemented a method of unordered pitch-class set analysis into the Forth of HMSL. In the second part, I will discuss an algorithmic composition based on a series of unordered sets, written in "vector" fashion in Forth but using the MIDI tools provided by HMSL, such as MIDI input and output routines and MIDI-file tools provided with HMSL. In the third part, I will describe the use of some object-oriented features of HMSL, such as instruments, shapes, players, collections, and jobs. The fourth and final part includes a discussion of objects used in a polyphonic algorithmic composition, based on a series of unordered sets derived from Fibonacci numbers.

II. Data Structure of Unordered Pitch-class Set.

To model the unordered pitch-class set (pc-set), the lowest
twelve binary digits (bits) of an integer are used. The presence or absence of the nth bit (0 thru 11) indicates the presence or absence of pitch-class (pc) n as a member of a pc-set. This data structure has several advantages. The data structure is as small as possible, and the minimum memory size (within 2 bytes) is maintained regardless of the cardinality (number of members present) of a particular pc-set. The small size also means that, for many types of operations on pc-sets (transposition, inclusion), a few machine instructions may effect the operation. For example, to determine if two sets have any pcs in common, a single AND instruction yields the resultant pc-set. Transposition is accomplished by a combination of instructions which shift bits and OR. However, multiplicative operations or mappings like inversion must be done bit-by-bit.

III. Tables of Prime Forms and Cardinalities.

Another aspect of a bit-mapped data-structure for pc-sets is the fact that the entire universe of possible pc-sets can be represented by the sequence of ascending integers from 0 (the null set) to 4095 (all twelve pcs). For the speediest performance, it is desirable to pre-calculate certain data about every pc-set and to store these data in tables for quick look-up. These tables can be organized in such a way as to allow the pc-set itself to act as its own numerical index into the table. Currently, two such tables are precalculated: one, PC#TAB, is a table of the cardinality of each pc-set; the second table, PRIMETAB, is a table that gives the prime form of any pc-set, that is, to which set-class (sc) of the basic 224 scs any pc-set belongs, along with transposition and inversion. Use of the prime-form table is
mainly in analysis, but the cardinality table is key to speeding up all set operations, since this data is being constantly referenced throughout the code. Much speed is saved by the one-time precalculation of the cardinality of each pc-set, since the calculation of cardinality is a more time-consuming bit-by-bit process.

The data structure of the prime-forms table allows the prime-form, inversion, and transposition of each of the 4096 pc-sets to be stored in two-bytes. The highest 4 bits are used to store the transposition (0-11). The lower 12 bits contain the bit-mapped pc-set which is the prime-form. If the indexing pc-set is an inversion of the prime form, however, the lower twelve bits of the table entry will also contain the inversion, but on pc 11, or T11I. The operation T11I must then be applied to obtain the uninverted prime form based, to which the transposition stored in bits 12-15 applies. Although this may seem cumbersome, it allows inversion information to be preserved in the table in a way that is easily testable: excepting the null set (whose cardinality is 0), in any pc-set which is an inversion of its prime form, bit 0 of its prime-form entry will be 0. Another reason for preserving the inversion of a prime-form is that the original set may be reobtained efficiently through transposition alone. And, in terms of analysis, sc-equivalence by inversion may not be a valid assumption (as in diatonic scs).

IV. Definitions of Twelve-Tone Operators: T, I, M, Inclusion.

In accordance with Forth convention, the integers which represent the pc-sets and cardinalities, as well as most other numeric parameters, are passed on a common stack. That is, words
that process pc-sets and cardinalities will, like other Forth words, expect parameters on the stack, and also leave any results on the stack. Known as the "stack effect" of a Forth word, it is, by convention, included symbolically in a comment on the first line of a definition.

One of the powerful features of Forth is its extensibility. This means that any definitions which are created by the user become part of the Forth language itself--that is, new definitions can be used in exactly the same manner as the pre-existing definitions which already make up most of the Forth language. To create a new Forth word, one begins with the word "colon" ([:), followed by a space (Forth words must be separated by at least one space). Following the space after "colon", the name of the new word is given. The execution effect of "colon" is this: it searches the input stream for the next string, which it takes to be the name of a new definition; it creates a dictionary entry for the new word; it switches forth from its usual "interpret" state to its "compile" state. In the "interpret" state, forth words are executed as they are encountered, either from a file or terminal; in the "compile" state, when a forth word is encountered in the input stream, its executable address is appended to the current definition. Thus a list of addresses of other forth words can be placed in the definition of a new word. Not all Forth words are so compiled in "compile" mode; a certain class of "state-smart" words allow for do-loops, if-else-then, begin-until, and so forth. One of these "immediate" words is "semi-colon" (;), whose effect, when ultimately encountered in a colon-definition, is to terminate the definition and return to "interpret" mode.

Let us look at two short definitions. They each start with a
colon and end with a semi-colon. Following the colon is the name of the new word, and following the name is a comment within parentheses (note the required space after the left parenthesis; it is a Forth word). The comment symbolically shows the stack effect, followed by a verbal description. The first, PC#, accepts a pc-set from the parameter stack, looks up the cardinality and returns it on the stack. The second, TN.PCSET, accepts a transposition and a pc-set, transposes the pc-set, and pushes the pc-set onto the stack.

Example 1. Definition of PC#.

```
: PC# ( w -- n , returns the cardinality n of pcset w)  
   setmask and pc#tab + c@  
;
```

The code of PC# will execute as follows. SETMASK has been previously defined as a constant, with a value of decimal 4095, or binary 111111111111. The effect of a constant is to push its value onto the stack. The Forth word AND subjects the top two stack items to be logically ANDed, with the result placed back on the stack. This is done to turn off any bits outside the valid range of a pc-set. The variable PC#TAB will push its address onto the stack. The Forth word "plus" (+) adds two stack items and leaves one. In this instance, it is adding the ANDed pc-set with the address of the cardinality table, PC#TAB. This gives the actual memory address where the cardinality is stored. The word "c-fetch" (c@) takes an address from the stack and returns the 1-byte character at that location.

Example 2. Definition of TN.PCSET.

```
: TN.PCSET ( w t -- w , rotate 12 low bits by t bits, ie transpose)
```
Let us follow the execution of TN.PCSET. The first word is "question-dup" (?DUP), whose stack effect is to leave a copy of the original stack item on the stack if the item is non-zero. Thus, any non-zero transposition will cause two identical items to be left on the stack; transposition by zero, however, will cause ?DUP to leave only one item, a zero. The IF-THEN and IF-ELSE-THEN constructs are supported in Forth, and exhibit the same reverse syntax as Forth in general. The Forth word IF takes an item from the parameter stack; if it is "true" (non-zero), then execution proceeds either to words between IF and THEN, or between IF and ELSE. If the value presented to IF is false (zero), then, if an ELSE clause is present, the words between ELSE and THEN will be executed, and the words between IF and ELSE skipped; if there is no ELSE clause, as there is not in our example, then any words from IF to THEN are skipped. The purpose of this code is to evaluate whether a transposition needs to be done—if transposition is by zero, then the transposition will be dropped and the original set returned.

ASHIFT is a word which accepts two numbers, the top number being the number of binary places to arithmetically shift the bottom number. This has the numerical effect of multiplying or dividing by powers of 2, and is effected on the Macintosh by a single machine instruction. In terms of the pc-set data structure, such shifting of the bit-pattern is equivalent to transposition. However, to effect a rotation T places within the lower twelve bits, opposite shifts are performed on two copies of the original pc-set: one shift is for T places, and one shift for 12-T places. The Forth word 2DUP is used to make a copy on the
stack of the first pair of stack items. These two shifted pc-sets are then ORed into a single pc-set including both sets of shifted bits. Finally, the constant SETMASK is ANDed with the pc-set to trim extraneous high-bits.

Once these definitions have been compiled into the Forth dictionary, they can be used within the definitions of other words. The highest-level analytical words supply prime forms (PRIMES?) or inclusions (INCL?) for a list of pc-sets, interactively, from the terminal. The pc-sets are classified according to Forte's set-class numbers.
Example 3. Interactive Pc-set Analysis. User commands are boldface.

**primes? 0235689 024 791 3652**

0235689 =T9I{0134679} (7-31) = <0235689>
024 =T0{024} (3-6) = <024>
179 =T7{026} (3-8) = <791>
2356 =T2{0134} (4-3) = <3652> ok0

**incl? 0235689 024 791 3652**

----------------PCSET INCLUSION---------------

024 within 0235689. Primes: T0{024} (3-6), T9I{0134679} (7-31).

---NO INCLUSION---

0235689 within 1356789AB. Primes: T9I{0134679} (7-31), T5{01234568A} (9-6).

---NO INCLUSION--- Hit key when ready!

----------------PCSET INCLUSION---------------

024 within 179. Primes: T0{024} (3-6), T7{026} (3-8).

---NO INCLUSION---

024 within 0234568AB. Primes: T0{024} (3-6), T6I{01234678A} (9-8).

T0 T2 T4 T6 T8 TA
T0I T2I T4I T6I T8I TAI Hit key when ready!

----------------PCSET INCLUSION---------------

179 within 2356. Primes: T7{026} (3-8), T2{0134} (4-3).

---NO INCLUSION---

2356 within 0234568AB. Primes: T2{0134} (4-3), T6I{01234678A} (9-8).

T0 T9
T5I T8I Hit key when ready!

ok0

V. Example of Real-time Composition Using Forth Without Objects.

Since every possible pc-set can be mapped into the bits of a decimal number between 0 and 4095 inclusive, any process that generates a series of number within that range can be called upon to generate a succession of pc-sets for an algorithmic music composition. The simplest such generation is to count by 1 from 0 to 4095 in a DO-LOOP structure, and then to use the DO-index (in Forth, returned by the word "I") as the source pc-set for some music process. This would cause every possible pc-set to be used once. Since HMSL has a powerful set of MIDI tools, we can start by building some words for playing any pc-set on a MIDI synthesizer.

The simplest way to play a pc-set is by simultaneously
sounding the corresponding pitches in the middle-c octave. To accomplish this, we need one word to send a MIDI note-on message for each pc, and another word, to invoke after a duration of time, to send a MIDI note-off message. HMSL provides the words MIDI.NOTEON for this purpose. MIDI.NOTEON removes a MIDI note-number and velocity from the stack, and immediately sends the MIDI note-on message out of the Macintosh serial port on the current MIDI channel (as set by the word MIDI.CHANNEL!). To turn a note off, the standard method is to send a note-on message with zero for velocity; (In spite of the fact that the MIDI 1.0 specifications provide for a distinct note-off message, the valid alternative of a zero-velocity note-on event is almost universally implemented both in hardware and software throughout the electronic music industry.)

Thus, in Figure 4., we have words that play any given set as a chord. The difference between PCSET.ON and PCSET.OFF is the velocity passed to MIDI.NOTEON. To provide some flexibility, two variables are used to hold velocity (VEL) and the offset MIDI note-number (PSPACE0), respectively. Although the loop tests each bit of the pc-set and plays the notes in quick succession, the audible effect is simultaneity, due to the speed of execution. The HMSL word MS takes a number and then waits that many milliseconds. Since the variable WAIT is initialized to 100, the word PLAY.ALLSETS1 will cause each and every pc-set to be sounded chordally at a rate of ten pc-sets per second; this study lasts 6 minutes and 50 seconds.
Figure 4. Words that play pc-sets via MIDI.

: PCSET.OFF ( w -- , turns off the low 12-bits as chord in middle octave)
  12 0 do
    dup i bit.on? if i pspace0 @ + 0 midi.noteon then
    loop drop
  ;

: PCSET.ON ( w -- , play the low 12-bits as chord in middle octave)
  12 0 do
    dup i bit.on? if i pspace0 @ + vel @ midi.noteon
    then
    loop drop
  ;

: PLAY.ALLSETS1 ( -- , a kind of example piece that uses every set )
  4096 0 do \ go thru each possible set
    i pcset.on \ play it as a chord
    wait @ ms \ wait
    i pcset.off \ turn it off
    ?terminal if key drop leave then \ kill the noise with a key
    loop
  ;

Other basic ways of playing pc-sets in pitch-space can be
constructed, such as playing the notes up and down (PCSET.UP,
PCSET.DN) or playing them according to order mappings (PCSET.BY.7,
PCSET.BY.5). In the latter pair of words, the order in which the
bits are searched for pcs to play sequentially is according to the
M and MI operations. For example, the search order of PCSET.BY.7
is <0,7,2,9,4,11,6,1,8,3,10,5>. These processes may yield
interesting, i.e., irregular and complex, melodic contours.
Figure 5. More words that play pc-sets.

: PCSET.UP ( w -- , plays each pc sequentially from low to high )
12 0 do
dup i bit.on? if i pspace0 @ +
dup vel @ midi.noteon
wait @ ms 0 midi.noteon
then loop drop
;

: PCSET.DN ( w -- , plays each pc sequentially from low to high )
12 0 do
dup 11 i - bit.on? if 11 i - pspace0 @ +
dup vel @ midi.noteon
wait @ ms 0 midi.noteon
then
loop drop
;

: PCSET.BY.7 ( w -- , plays each pc sequentially checking every 7th pc )
12 0 do
dup i pc>m7 bit.on? if i pc>m7 pspace0 @ +
dup vel @ midi.noteon
wait @ ms 0 midi.noteoff
then loop drop
;

: PCSET.BY.5 ( w -- , plays each pc sequentially checking every 5th pc )
12 0 do
dup i pc>m5 bit.on? if i pc>m5 pspace0 @ +
dup vel @ midi.noteon
wait @ ms 0 midi.noteoff
then
loop drop
;

Even a small collection of such simple algorithmic tools can
be used in combination to achieve a wide variety of more complex
performance gestures, in spite of the limitations of the "vector"
approach to algorithmic computation. In PLAY.ALLSETS3 (the title
of the music is "Ondine's Curse"), pc-sets are generated in the
same manner, in ascending integer order by means of a DO-LOOP
index, as in PLAY.ALLSETS1. A CASE statement is used to choose
which of thirteen performance options will be applied to any given
pc-set, and the cardinality of the pc-set is the datum by which
the CASE statement chooses. Each of the possible gestures plays
the pc-set using one or a combination of several of the above-
described words, along with manipulation of certain variables that
the words use in common, namely VEL, PSPACE0, and WAIT. Before
each loop, these variables are set to default values. Within a
gesture, they may be altered or scaled, which in turn modifies the
response of words like PCSET.ON, changing the dynamics,
transposition, or durations.

Example 6. Forth Definition of PLAY.ALLSET3.

: PLAY.ALLSET3 ( fromset toset -- , another kind of piece. max 4096 , 0)
  rst.vars  use.piano.sound \ setup vars and synths for playing
  0 4096 clipto swap \ enter DO parameters to sectionalize performance
  0 4096 clipto -2sort \ use CLIPTO and -2SORT to edit valid DO parms
  do
    ?terminal if key drop cr quit then \ EXIT with ANY KEY!!!
    i cls .set \ PRINT SET STUFF, FYI
    i pc# ( -- # )
    case ( do a different thing for each set cardinality )
    1 of i pcset.on
      wait @ 4 ashift ms i pcset.off
      endof
    2 of wait @ dup -1 ashift + wait !
      i pcset.dn
      endof
    3 of i pcset.by.7
      endof
    4 of -12 pspace0 +! i pcset.on wait @ ms
      24 pspace0 +! i pcset.on wait @ ms
      i pcset.off -24 pspace0 +! i pcset.off
      endof
    5 of wait @ dup dup -1 ashift + + -2 ashift wait !
      -12 pspace0 +! i pcset.up
      -8 vel +! 12 pspace0 +! i pcset.up
      -8 vel +! 12 pspace0 +! i pcset.by.5 i pcset.by.7
      endof
    6 of 16 vel +!
      i pcset.on wait @ 2 ashift ms i pcset.off
      endof
    7 of 32 vel +! i pcset.on wait @ 15 + ms i pcset.off
      -12 vel +! i pcset.on wait @ 5 + ms i pcset.off
      -12 vel +! i pcset.on wait @ 5 - ms i pcset.off
      -12 vel +! i pcset.on wait @ dup 15 - + ms i pcset.off
      endof
    8 of -12 pspace0 +!
      wait @ -1 ashift wait !
      i pcset.by.5
      endof
    9 of -12 pspace0 +!
      wait @ dup -1 ashift + -1 ashift wait !
      i pcset.by.7
Another feature of the MIDI protocol is the standard file structure for MIDI data, making possible communication and data transfer among MIDI applications such as sequencers, notation programs, even between Macintosh, IBM-PC, or other hardware platforms. Example 7 shows a terminal command causing PLAY.ALLSET3 to be executed in such a way as to also produce a MIDI file of the first eighth of "Ondine's Curse."

Figure 7. Execution of PLAY.ALLSET3 captured in MIDI file.

```
midifile{ ha:part1 512 0 play.allsets3 }midifile
```

VI. Object-oriented programming in HMSL.

Many limitations exist in this kind of algorithmic composition, with perhaps the most important being the lack of polyphony. PLAY.ALLSET3 is generated by one "vector" of code, and is primarily monophonic, or homophonic. Even though, for example, the case of cardinality 11 causes a low chord to be held while a higher melodic line is stated, the program is still generating
both events in a single path. What if one instead wanted to divide an 11-member pc-set into ps-sets of cardinality (say) 7 and 4, and then cause the respective gestures of cases 7 and 4 to be started simultaneously, and on different MIDI channels? For this polyphonic purpose, the fundamental vector design is not easily adapted.

A great power of HMSL lay in its ability to maintain the concurrent execution of multiple processes. This power is reserved in the object-oriented data structures which HMSL provides. Briefly, at the highest level of HMSL, a timer- and scheduler- object keeps track of any and all task-objects "posted" to it, maintaining each task's timing and execution. This HMSL "executive" can even be run in the background, allowing Forth commands to be entered in the foreground while MIDI-generating processes are running, allowing real-time modification, interrogation, and debugging of "posted" tasks. But before we explore the higher levels of HMSL, and in order to understand programming "objects," let us examine, first, some general aspects of objects, and then, second, an example of a polyphonic composition generated by a hierarchy of HMSL objects.

The HMSL manual contains succinct discussions and definitions of its object-oriented capabilities, and some of these are worth quoting here:

Object-oriented programming allows you to design your program around the manipulation of objects. You can use existing classes, or types, of objects, like arrays, or define new classes of objects for your application. A class defines what an object is made of and what it can do. Once you have defined a class, you can create as many objects of that class as you want. You can think of an object as an intelligent data structure. Each object knows how to manipulate its own internal data. All objects of a given class will have the same internal structure. They will also use the same methods for manipulating that data.

Class: a description of a data structure that describes what data is contained within it, and what methods are used to manipulate it.
Object: an instance of a class of data structures.

Method: a procedure or function that is associated with a particular class. By convention, method names generally end in a colon, for example, PRINT:.

Superclass: the class from which a new class is derived. All classes are derived ultimately from a root class named OBJECT.

Instance variable: a variable that is contained within each object of a given class. To access an instance variable from outside an object, you must use only that object’s defined methods. ... Instance variables can themselves be objects.

Inheritance: Each new class automatically has available all the properties of its superclass, including all instance variables and methods. The new class then adds new methods and/or instance variables.

HMSL contains many predefined classes of objects, which range from simple integer objects and arrays of one, two, or more dimensions, to shapes, instruments, MIDI instruments, translators, players, collections, jobs, and other more specialized objects. For example, shapes, of the class OB.SHAPE, are objects that may contain ordered lists of MIDI note data, and, as such, are the primary data object in HMSL. Objects of class OB.PLAYER are executable, by means of the START: method. The START: method assumes that the HMSL timer-scheduler has been activated, via the invocation of the words HMSL, HMSL.START, or HMSL.PLAY. When an executable object is started, it "posts" itself to the timer-scheduler at the heart of HMSL which then oversees the object's execution.

To generate musical MIDI output, an object of the OB.PLAYER class must be started, and it must contain an object of the OB.INSTRUMENT class along with at least one object of the OB.SHAPE class containing MIDI data (timing, note numbers, velocities, on-times). The player then receives timing information from HMSL, compares the system time to the time specified by the shape(s), and passes the appropriate shape data to the instrument, which controls the actual sending of the MIDI messages.

This ubiquitous configuration of shape-, instrument-, and
player-class objects is still a monophonic, or homophonic structure. But since HMSL supports the concurrent execution of many player objects at once, polyphony is thus obtained.

VII. Example of Real-time Composition Using HMSL Objects.

Perhaps the best way to illuminate the mystery of these software objects and their methods is to examine the code I have written to play 3-part polyphony. The pitch material is provided from the generation of a series of integers whose bits are then mapped to pcs, as described above. Instead of the index of a DO-LOOP, however, the integer series is the Fibonacci series (the lowest 12 bits). In this, like other Lucas series numbers, the next number of the series is the sum of the previous two. As these numbers grow larger and larger, the ratio between any pair tends to converge upon the fabled "golden mean." It has been my informal hypothesis that this series would yield a certain proportionality that would lend itself to "classical" balance, even in the context of a computer algorithm.

My research has found that, when viewed as binary numbers, the bits of successive numbers form repeating patterns whose lengths are equal to 3 times the value of the bit. Figure 8 shows the first two dozen Fibonacci numbers, in decimal and binary, along with the corresponding pc-set, prime form, and Forte set-class number. Note that in the column of the lowest (rightmost) binary digit, a repeating pattern of length 3 is apparent (011 011 011...). The next column to the left has a pattern of length 6 (the spaces are zeros, 000110 000110 000110 ...). In other words, the lowest bit, with a value of 1, engenders a pattern of length 3; the next higher bit, with a value of 2, has a pattern of
length 6; the next bit, valued at 4, creates a length 12 pattern, and so forth. By the 12th binary digit, the length of the column pattern is 6144. This means that after 6144 Fibonacci numbers, by which time the full Fibonacci integers are growing to astronomical values, the pattern of the 12th bits will begin to exactly repeat. Thus this series of pc-sets, in which each pair of pc-sets is unique, is derived from the lowest 12 bits of the first 6144 Fibonacci integers.

Example 8. Repeating Patterns in Binary Fibonacci Series.

<table>
<thead>
<tr>
<th>Fibonacci Number</th>
<th>Binary Representation</th>
<th>pc-set 1</th>
<th>pc-set 2</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>{}</td>
<td>{}</td>
<td>0-1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>{0}</td>
<td>{}</td>
<td>1-1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>{}</td>
<td>{0}</td>
<td>1-1</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>{}</td>
<td>{01}</td>
<td>1-1</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>{}</td>
<td>{02}</td>
<td>2-2</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>{}</td>
<td>{3}</td>
<td>3-6</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>{}</td>
<td>{023}</td>
<td>3-2</td>
</tr>
<tr>
<td>21</td>
<td>10101</td>
<td>{}</td>
<td>{024}</td>
<td>3-6</td>
</tr>
<tr>
<td>34</td>
<td>100010</td>
<td>{}</td>
<td>{15}</td>
<td>4-12</td>
</tr>
<tr>
<td>55</td>
<td>110111</td>
<td>{}</td>
<td>{01245}</td>
<td>5-3</td>
</tr>
<tr>
<td>89</td>
<td>1011001</td>
<td>{}</td>
<td>{0346}</td>
<td>4-12</td>
</tr>
<tr>
<td>144</td>
<td>10100000</td>
<td>{}</td>
<td>{47}</td>
<td>2-3</td>
</tr>
<tr>
<td>233</td>
<td>11101001</td>
<td>{}</td>
<td>{03567}</td>
<td>5-36</td>
</tr>
<tr>
<td>377</td>
<td>101111001</td>
<td>{}</td>
<td>{034568}</td>
<td>6-39</td>
</tr>
<tr>
<td>610</td>
<td>1001100001</td>
<td>{}</td>
<td>{1569}</td>
<td>4-19</td>
</tr>
<tr>
<td>987</td>
<td>1111101111</td>
<td>{}</td>
<td>{01346789}</td>
<td>8-18</td>
</tr>
<tr>
<td>1597</td>
<td>11000111101</td>
<td>{}</td>
<td>{023459A}</td>
<td>7-14</td>
</tr>
<tr>
<td>2584</td>
<td>101000011000</td>
<td>{}</td>
<td>{349B}</td>
<td>4-16</td>
</tr>
<tr>
<td>4181</td>
<td>1000000101010</td>
<td>{}</td>
<td>{0246}</td>
<td>4-21</td>
</tr>
<tr>
<td>6765</td>
<td>1101001101101</td>
<td>{}</td>
<td>{023569B}</td>
<td>7-31</td>
</tr>
<tr>
<td>10946</td>
<td>10101011000010</td>
<td>{}</td>
<td>{1679B}</td>
<td>5-24</td>
</tr>
<tr>
<td>17711</td>
<td>100010100101111</td>
<td>{}</td>
<td>{012358A}</td>
<td>7-23</td>
</tr>
<tr>
<td>28657</td>
<td>1101111111110010</td>
<td>{}</td>
<td>{0456789AB}</td>
<td>9-1</td>
</tr>
<tr>
<td>46368</td>
<td>1011010100100000</td>
<td>{}</td>
<td>{58A}</td>
<td>3-7</td>
</tr>
<tr>
<td>75025</td>
<td>100100101000100001</td>
<td>{}</td>
<td>{048A}</td>
<td>4-24</td>
</tr>
<tr>
<td>121393</td>
<td>111011010001100001</td>
<td>{}</td>
<td>{0459B}</td>
<td>5-20</td>
</tr>
</tbody>
</table>

Since there is a relationship between each pair of numbers, I decided to derive the pc-sets for three part polyphony in the following way. Every pair of sequential Fibonacci numbers yield two pc-sets. To obtain a third pc-set, the original pair are ANDed, yielding the intersection, if any. Then the intersecting
pc-set is XORed (exclusive or) with each original pc-set to removed the common pcs from the original pair. This produces three pc-sets without pc duplication. These three sets are ultimately placed in different octaves of pitch-space and assigned different timbral voices.

Thus having created a large supply of pc-sets, the next problem to address is rhythmic. It was decided that one trio of pc-sets would be realized per some arbitrary measure. First, several words were defined to transform an unordered bit-mapped pc-set into a series of elements (one per pc) in a shape object. The basic word, PCS>SHAPE, places the pcs of a pc-set in ascending order, with zero for each starting time, default velocities of 64, and each on-time equal to the measure length. This is roughly equivalent to the purpose of PCSET.ON, except that no corresponding PCSET.OFF is necessary, since, because the shape object has on-time for each element, HMSL will turn off the notes at the time specified by the shape.

In order to test the rhythmic independence of HMSL, the word PCSH.MEAS.PROL was defined. PCSH.MEAS.PROL takes a shape, looks at how many elements (pcs) are in it, and then evenly prolates the elements' starting times over the measure length, and adjusts the on-time of each element appropriately as well. If a shape has 7 elements, PCSH.MEAS.PROL will create an even, monophonic septuplet over the measure length.

A related word, PCSH.FIT.PROL, accepts a shape and a prolation, then forces the elements of the shape to conform to the desired prolation. If a four-element shape is passed to PCSH.FIT.PROL along with a prolation value of 7, then the four elements will be aligned on the nearest septuplet boundaries. PCSH.FIT.PROL will subdivide the prolation again if necessary. If
a seven-element shape is passed to PCSH.FIT.PROL along with a prolation value of 4, then the seven elements will be aligned on the nearest octuplet boundaries. These and other related definitions are listed in Figure 9.

The code listed in Figure 9 also supplies many instances of the syntax of object methods. If an particular object is specified in the code, then its name follows the method. For example, in the first definition, the TRANSLATE: method is used to pass a value to a specific object, ORDERNUMS, whose name follows the method. If the object's name is not specified at compile time, its address can still be passed to a method on the stack, by following the method with the word "bracket-bracket" ([]); the equivalent way of coding the previous example would be ORDERNUMS TRANSLATE: []. In the definitions of Figure 9, any address of a shape object may be passed on the stack (it is then placed in the local variable SH). Thus these routines are generalized and can process any shape object. The most important methods of shape objects are used here: the NEW: method initializes an empty shape of certain 2-dimensional size; the ADD: method adds an element (time, note, velocity, ontime) to the shape; the MANY: method returns the number of elements, equivalent here to cardinality; the ED.AT: method returns the data item at a particular element (row) and dimension (column) of a shape; and the ED.TO: method stores a datum in a particular element and dimension.

Figure 9. Definitions for Converting Pc-sets into Shape Objects.

: pcs>shape { w sh -- , one element per pc, ascending order}  
w pc# 4 sh new: []  ( allocate pc# elements in TYPE 3 SHAPE)  
12 0 do ( place pcs into successive elements of shape )  
   1 i ( this bit will be shifted...i will be order-mapped)  
      translate: ordernums
ashift w and
?dup if
  0  ( all starting times 0, meaning all notes play at once )
  swap 2* primetab + wθ -12 ashift
  ( pitch-class, as tn in primetab)
  dflt_vel @  ( velocity...)
  measure_len @  ( ontime...)
  sh add: [] then
loop
;
: shape>pcs { sh -- w , collapses dimension 1 into a pc-set}
  0
  sh many: [] 0 do
    i 1 sh ed.at: []
    12 mod dup 0< if 12 + then  ( convert pspace to pcspace)
    bit.on
  loop
;
: pcsh.meas.prol { sh prol -- , fits elements into prolation equally as possible}
  sh many: [] 0 do
    i measure_len @ * dup   ( where i=0, the value is always 0 )
    if sh many: [] / then   ( mult by measure_len, then div by pc#)
    i 0 sh ed.to: []   ( set absolute time )
    measure_len @ sh many: [] /   ( divide initial measure sust by card)
    shave.ontime   ( 3/4 max duty cycle for good MIDI performance)
    i 3 sh ed.to: []
  loop
;
: within.prols { tval prol -- lo hi , finds two prolpoints nearest tval }
  0
  prol 1+ 0 do
    i measure_len @ * prol / tval over >
    if swap drop else leave then
  loop
;
22
These words and others are used to fill the three shapes in a variety of ways. The variant seen in Figure 10 imposes different ordering scheme for the three pc-sets, but forces each shapes to conform to the rhythmic prolation of the largest pc-set. Since, in the last line, the PIANO-PART is left essentially as PCS>SHAPE has made it, i.e., with zero for each element's starting time, a chord will result. The word PCSH.SHAVE.ONTIME slightly reduces the sustain (dimension 3) of every element in a shape, improving the MIDI response.
Figure 10. Definition of SET.PARTS4.

: set.parts4 ( -- )
  lucas+ ( generate another pair of integers)
  lucas@ set.trio.pcs ( break the two ints into 3 pc-sets)

  ( PLACE THE PCS OF EACH SET INTO A SHAPE HAVING PC# ELEMENTS)
  m7.order
  get: vn-pcs violin-part pcs>shape
  p.order
  get: pf-pcs piano-part pcs>shape
  measure_num @ 1 and if i.order else p.order then
  get: vc-pcs cello-part pcs>shape

  ( SET DURATIONS AND ON-TIMES FOR EACH SHAPE )
  violin-part many: []
  piano-part many: [] -2sort drop
  cello-part many: [] -2sort drop ( find the highest cardinality )

  violin-part over pcsh.fit.prol ( force all the sets to conform to it)
  cello-part swap pcsh.fit.prol
  piano-part pcsh.shave.ontime

;

Now our attention focuses on the actual creation of a hierarchy of objects—in this case, three players, three instruments, a collection object, and a job object—which can effect the performance of a measure, and, ultimately, the performance of many measures. First, both a shape and an instrument must be placed in each of three player. Then all three players are placed in a single collection object; when the START: method is used with the collection, each of the objects in the collection will also receive a START: message, and all three players will perform concurrently. Collections are important objects in creating execution hierarchies in HMSL, since they can contain other collections, and can pass START: and STOP: messages to any objects "down the hierarchy." They can be set to execute simultaneously, sequentially, or according to an arbitrarily programmed behavior.
Having set up the hierarchy so far, we can now send the collection a START: message and all three players will play their shapes. The fact that we are going to continually remake the three shapes at the very bottom of the hierarchy has no effect on the hierarchy itself. One need not repeat placing the shapes in the players, nor the players in the collection, and so on, in spite of the fact that the contents of the shapes are going to be revised between each execution of the collection.

One more piece of code is needed, and at a higher level than the collection. Its tasks are to (1.) check the time to see if it is the beginning of a measure, (2.) if it is, generate three new pc-sets and stuff them into the shapes, and also (3.) start the collection. The object we need is of the class OB.JOB, and objects of this class, like players and collections, use the
The function of a job is to do something periodically--the something and the period are supplied by the programmer. That is, a job object contains one or more addresses of Forth words. Each time the job is executed by HMSL, the Forth words it contains are executed. By placing a large repeat-count in the job (using the PUT.REPEAT: method) and a zero duration (using the PUT.DURATION: method), then, once started, the job will be executed over and over again as fast as HMSL finds possible (HMSL may be concurrently executing other objects). Under these conditions the behavior of the object of class OB.JOB resembles an operating system interrupt, and so the Forth words that a job executes must be clean and well-designed; otherwise, the entire system may crash. (It is a well-defended right of Forth programmers to be free to crash their systems.) Thus, to play the entire composition (which, in its present form, lasts a few hours) a START: message is sent to the object TIMER, which then posts itself to the HMSL executive and its repeatedly executed; TIMER then executes the Forth word TIMER.JOB each time; TIMER.JOB launches the collection TRIO, which in turn launches the three players it contains.

Figure 12 shows the definition of TIMER.JOB, the Forth word whose executable address is placed in TIMER, an object of class OB.JOB. Following the definition is the line of code which will assign the Forth word TIMER.JOB to the job object TIMER. TIMER.JOB keeps a measure count, as well as re-formulating and re-starting the collection. Drawing on my experience as a computer game programmer, TIMER.JOB has been designed as a flexible, interrupt-style, word, which can be modified in real-time via
Forth commands from the terminal.

Figure 12. Definition of TIMER.JOB, Initialization of TIMER.

```
: timer.job { job -- , like an INTERRUPT ROUTINE...}     
timer-status @ case
  0 of
    test-cfa @ ?dup if execute then
      -1 timer-status ! 0 time!
      1 timer-reps +! 1 measure_num +!
    prep-cfa @ ?dup if execute then
      start-adr @ ?dup if start: [] then
      print-cfa @ ?dup if execute then
      endif
    endif
  -1 of
    measure_len @ doitnow?
    if 0 timer-status !
      timer.job-cfa @ ?dup
      if 0 swap execute { simple recursion...}
      else 1 timer-reps +! then
      endif
    endif
  endcase

; stuff{ 'c timer.job }stuff: timer
```

To maximize the flexibility and versatility of the highest hierarchical level, practically all tasks initiated by the execution of TIMER.JOB are vectored—that is, the addresses of the subtasks that TIMER.JOB will execute are found in variables.

The most important of these variables are PREP-CFA and START-ADR. PREP-CFA contains the address of a Forth word like SET.PARTS4, whose purpose is to generate a new Fibonacci pair, derive the new pc-sets, and refill the three shapes. The CFA, or Code Field Address, of a Forth word is its executable address. In HMSL, the CFA is obtained by using the word "tick-c" ('c) followed by the name of a word; "tick-c" returns the executable address on the stack.

START-ADR contains the address of an object, an address which is not equivalent to the executable address of a forth word. As
an object, it must be manipulated by a method of its class, namely the START: method. START-ADR will have been initialized to the address of our collection, TRIO.

One other variable, TEST-CFA, was included at the beginning of TIMER.JOB to execute code for testing and further experiments. The Forth word currently executed, TIMER.TESTER, chooses from among several shape-filling words like SET.PARTS4, and then dynamically stores the address in PREP-CFA. This demonstrates HMSL's ability to self-modify in real-time.

VIII. Conclusions.

Any unordered pitch-class set can be represented by the bits of some single integer in the range 0 to 4095. Twelve-tone operations on this data structure are simple and run fast. Any integral function which generates a series of integers can thus be used to provide pitch-class material for algorithmic compositions. This paper demonstrates the use of two such integer series, (1.) the series of ascending integers from 0 to 4095, and (2.) the Fibonacci series.

HMSL provides a flexible real-time programming environment for such experiments, including powerful MIDI and object-oriented tools. However, knowledge of MIDI, and some computer programming knowledge (the Forth language, in particular) are necessary to take full advantage of HMSL's features. This paper demonstrates and describes some of the capabilities of HMSL by implementing two algorithmic compositions: one, using ordinary linear, or "vectored," programming and the other using object-oriented techniques to effect real-time polyphony.
IX. References.

