Chapter 1
Introduction to HMSL

Getting Started

Information on how to install and run HMSL can be found in the Macintosh- or Amiga-specific *Manual Supplement*. Once you have HMSL up and running, you may want to take the "quick tour" described in this chapter. We suggest that you then follow the guidelines in the next chapter on learning HMSL. Another good thing to do right now would be to quickly skim through this manual, so that you have some idea of the variety of concepts that lie ahead, and what kinds of things HMSL allows you to do.

About HMSL

HMSL stands for *Hierarchical Music Specification Language*. HMSL is a programming language and software environment for experimental music composition, performance and research. Our goal is to provide tools needed by experimental composers and performers without restricting musical style. We have tried to make HMSL as flexible as possible. The source code for HMSL is provided for the composer to even change HMSL itself, although that is seldom necessary. Some of the tools HMSL provides are:

- a language for creating and executing musical data, processes, and functions in *complex hierarchical forms*
- software tools for *algorithmic composition* including intelligent data structures, process control, real-time user-definable stimulus response support, and many other utilities, including distribution functions (randomness), math routines, and other forms of data manipulation
- support for controlling MIDI synthesizers, parsing and mapping MIDI input, sending system exclusive messages, reading and writing standard MIDI files, and synchronizing with other MIDI systems
- support for *Amiga local sound*, samples and waveforms
- *graphical editor* for manipulating abstract numerical data such as melodies, samples or system exclusive MIDI parameters
- a text based *Score Entry System*
- a simple *Sequencer* that can be customized by the user
- a *User Interface toolbox* for designing custom mouse driven compositions or editors
- a flexible *scheduling* system for accurately timed complex musical events
- a variable rate real time clock, 20-1000 ticks/second
- many example pieces

These features are integrated into 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.

The Organization of HMSL

The HMSL software environment is organized into layers. The composer may use tools from any layer at any time. The foundation of HMSL is the *Forth* language. Forth is an interactive general purpose programming language that lends itself well to experimental systems. On top of Forth are a number of toolboxes. One of these is the MIDI toolbox, with support for sending MIDI commands and responding to MIDI input. Other toolboxes include Timing Support, Graphics, distribution functions, file I/O support, a source level debugger, arithmetic tools, etc.

The next layer is *ODE*, the *Object-Oriented Development Environment.* ODE simplifies programming by letting the programmer create "software things" called "objects" that can be manipulated like objects in the real world. The next layer of HMSL defines special classes of objects called *morphs* that support music composition and performance. Examples of *morphs* are *Virtual MIDI Instruments*, data containers called *shapes*, *players* that play shapes, and others. Above all this is the executive program that coordinates all of these tools, handles the graphics and mouse input, and schedules music events.

At the very top is the composer's own program written using the various HMSL tools. An HMSL program/composition typically consists of code that sets up objects, organizes them into a piece, then passes them to HMSL to be played. The composer can also load HMSL with special functions that specify how to respond to MIDI input, customize the scheduler, or interpret compositional data. *Interactive control grids* can be used to define an interactive *screen* with buttons, faders, and other controls. This allows a performer to interact with a piece as it runs. Since HMSL comes with complete source code, HMSL itself can be modified, if needed, for a particular piece.

An Outline of the HMSL Environment

User Composition

HMSL (Containing):

- Loadable Tools
    - Score Entry System
    - Sequencer, etc.
- HMSL Executive
- Morphs - shapes, collections, etc.
- User Interface Support
- Time and Scheduling Tools
- (Amiga Local Sound Toolbox)
- MIDI Toolbox
- Miscellaneous Tools

ODE - Object Oriented Development Environment

Forth

About Forth

Forth was chosen as the foundation language of HMSL for a number of reasons. Forth is interactive. The programmer can type in commands to be executed immediately. Data structures can also be examined and changed interactively. This is in contrast to "compiler only" languages like 'C' where everything, including simple tests, must be compiled. This can take time and distance the composer from her work. Some other languages, notably LISP and BASIC, are also interactive but often execute too slowly for complex real time performance.

The Forth compiler can be extended to support new language constructs. One can actually write a *word* (Forth subroutine) which executes at *compile time* instead of at *run time*. This feature facillitates ODE, which needs to compile some complex data structures and has an unusual syntax.

A somewhat whimsical example of extending the compiler might be to play a MIDI note every time a word is compiled. The pitch could correspond to the size of the word's code. As the program was compiled, you would hear whether there were mostly large or small words.

Forth executes quite fast. This is important for a real time application like music. Some Forths are faster then others. Most Forths compile into a token language that is interpreted at run time. This is also true of most BASICs and some versions of PASCAL. It is possible, however, on the 680x0 processor to write a Forth that compiles directly to machine code. Forths that do this are called *subroutine threaded* Forths. These Forths execute about 2-3 times faster than traditional Forth and are about as fast as 'C' code. The two Forths that HMSL uses (JForth for the Amiga, HForth for the Macintosh) are subroutine threaded.

Historically, Forth has also been a kind of "lingua franca" for live computer music since the early 1970's. Many composers using computers for real time performance, algorithmic composition, and experimental music adopted this language because of its economical memory requirements (no longer perhaps the issue it was back in the old days) and its tremendous flexibility. It remains to this day one of the most widely used languages for real-time computer music, especially among experimental composers.

The Amiga version of HMSL uses *JForth*, a third party Forth from Delta Research. The Macintosh version uses *HForth*, written specifically for HMSL. On disks you will see the abbreviation "H4th" for this. For more information on JForth see the Delta Research JForth manual. For more information on HForth, see the *Macintosh Supplement* of the HMSL Manual. You may also want to look in the bibliography of this manual for suggested text books on Forth.

Our intention is that pieces written in HMSL on the Macintosh will run on the Amiga, and vice versa. Pieces written in HMSL that use only the documented features in this manual should run on both the Amiga and the Macintosh.

What is Object Oriented Programming?

Object Oriented Programming is a style of programming which became popular in the 1980's, and was designed to simplify program development, promote reuseable code and simplify code maintainance. Most object oriented programming languages are derived from *Smalltalk*, created at Xerox PARC. The Macintosh and Amiga interfaces, for example, are based on the object oriented interfaces developed there.

What are *objects*? Essentially, objects are intelligent data structures. They not only contain data like a 'C' structure or a Pascal record, they also know what to do with it. There are different kinds, or *Classes* of objects. Once a class of object is defined, you can make, or *Instantiate* as many of them as you want.

To use an object, you send it a *message* telling it what you want it to do. The message can be generic like PRINT or DRAW. Each object will respond to the message based on the methods defined for that class of object, and the data values contained within that object. This means a generic message like START can be sent to a bunch of different kinds of objects and they will all "do their own thing". This makes it easier to write editors, for example, because you don't have to know exactly what it is you are editing. You just send some "thing" commands like INSERT and DRAW, and it does what it needs to. In HMSL the same editor is used for editing melodies and Amiga Audio samples. They are drawn differently and behave differently, but because HMSL is object oriented, the editor doesn't need to know that.

HMSL uses an *Object Oriented Development Environment* called *ODE* written by Phil Burk. For more information, see the manual chapter on ODE.

Morphologies — HMSL Classes for Composition

HMSL has a number of object classes designed for music composition and performance. These classes are referred to generically as *Morphs*, which is short for *Morphologies.* Morphs can be organized into hierarchies and executed as a unit. The following is a brief description of some of the major morphs in HMSL. Each of these classes will be explained in more detail in later chapters.

SHAPE — raw numeric data organized as points in a multidimensional space, eg. time/pitch/loudness

PLAYER — plays one or more shapes, responsible for timing

INSTRUMENT — intelligent software interface to a physical instrument. Interprets shape data for output

COLLECTION — contains other morphs, hierarchy building block

STRUCTURE — Collection with associated Markov transition table to determine order of execution

JOB — periodically executes user functions, for background processing

PRODUCTION — executes user functions once

ACTION — gives programmable response to programmable stimuli

Macintosh and Amiga Implementations

Our intent is to make the Macintosh and the Amiga implementations equivalent and portable. This is made difficult by the great differences in the operating systems of the two machines. This manual will point out areas where the user should be aware of differences in the two versions, and suggest ways to avoid letting those differences inhibit the portability of code. Please see the accompanying Amiga or Macintosh Supplements for more information on each machine. *If you always work on an Amiga, or always work on a Macintosh, you will never need to be concerned about these issues of portability.*

History of HMSL

HMSL grew out of the works of a number of composers, programmers and theoreticians. Key among these were David Rosenboom, Larry Polansky and James Tenney, who worked together for a period of time in the Electronic Media Studios of the Music Department at York University in Toronto. Tenney, a pioneer in computer based composition, had begun formulating theoretical ideas in the early 60's (see *Meta+Hodos* and its 70's expansion*META Meta+Hodos*) in which he defined new ways of describing modern electronic and acoustic music in terms of perceptible sound events. In the 70's he and Larry Polansky collaborated on a program to model *hierarchical temporal gestalt perception*.

Since the late 60's Rosenboom had been working on models of musical perception and performance involving analysis of electrical activity in the brain employed to direct sound synthesis and algorithmic composing processes. The emphasis was not only on generating sounds to map brain activity, but on the spontaneous evolution of musical *forms*, directed by shifts in selective musical attention as revealed in the brain signals (see *Biofeedback and the Arts*, and the *Extended Musical Interface with the Human Nervous System*). During the 70's David also worked on a number of *intelligent* musical instrument languages, all of which included realizations of hierarchical structuring capabilities. These were implemented on early mini-computers, usually connected to Buchla hardware for sound synthesis output and performance input. Much of this development was supported by grants from the Canada Council's Exploration Program and by York University. David also collaborated during this period with Donald Buchla and Doug "Lynx" Crowe on instrument software designs. Buchla and Associates' PATCH-IV language already included a hierarchical patching system for dynamically structuring and restructuring electronic instrument setups. David wrote an early prototype of HMSL to run on the TOUCHÉ, a digital keyboard instrument developed in 1979 by Buchla Associates and Rosenboom. This prototype included HMSL-type data structures, like *shapes* and *collections*, along with *perform* and *action* environments. Though these features were not as advanced as those that exist in HMSL today, they contained many similar characteristics.

In 1980 David and Larry were able to join forces at the Mills College Center for Contemporary Music to realize what had been a long stanbding goal of both, the creation of HMSL. A prototype was written for an ERG an ERG 68000 based S-100 system running a 16 bit Forth, containing both digital and analog I/O and an interface to a Buchla and Associates 400 series digital waveshaping oscillator system.

At around the same time, Phil Burk was developing music systems based around a single board Z-80 computer system. Phil developed a performace oriented sequencer that controlled various home brew synthesis devices. He also developed a pitch detection system that converted an incoming guitar signal to an arbitrary digital waveform.

In 1984 Larry, David and Phil converged on a single project, a new, portable version of HMSL based on Object Oriented Programming techniques (Version 2.0). Support for MIDI was also added. In 1985, a grant from the Inter-University Consortium on Educational Computing (ICEC) accelerated this development. In order to reach a wider audience, HMSL was then ported over to the Amiga where they used the Multi-Forth compiler from Creative Solutions. Polansky, Burk and Phil Stone (a graduate student at the CCM and now a member of *The Hub*) performed a computer music concert in San Francisco in 1986 using this early version of HMSL. That may have been the first concert where only Amiga local sound was used.

Shortly after that concert, Burk tried out a new Forth compiler called JForth which turned out to be about 2-3 times faster then Multi-Forth. Burk worked with Mike Haas to port HMSL to JForth. In the spring of '87, HMSL was released to a small group of composers and artists in New York City at *PASS*, the Public Access

Synthesizer Studio. It has been available commercially since that time and is being used all over the world by many individuals and at many colleges, universities, studios, and electronic music and arts facilities.

In late 1987 HMSL (Version 3.xx) was ported to the Macintosh using the Mach 2 Forth compiler from Palo Alto Shipping Company. This latest version of HMSL (Version 4.0) on the Macintosh uses a new Forth compiler called HForth written specifically for HMSL by Phil.

Development of HMSL continues steadily. Many users have made execllent suggestions that have been incorporated in the official version. Features added to Version 4.x include the *Score Entry System*, *MIDI File support*, *Event Buffering*, the *Sequencer* and *Graphics Controls* and support for the new Apple MIDI Manager. At present, Phil Burk manages the maintainance and distribution of HMSL, along with the addition of new features.

A Quick Tour of HMSL

Let's take a brief look at the major features of HMSL. This section is not intended to teach you how to use HMSL. We just want you to get an idea of what HMSL can do. Once you have the "big picture" you can proceed with the next chapter which has suggestions on how to learn HMSL. You do not need to know Forth to do this tour although it could help. Some of the examples will use Forth concepts without explaining them. If you have problems with this tour, please go back and do some of the Forth tutorials before proceeding. Please remember that all of the features shown in this tour are explained in detail in other parts of the manual.

Before starting this section, please look in the Macintosh or Amiga Supplemental Manuals for instructions on how to run and test HMSL. We will assume that you now have HMSL running and that the MIDI output and audio system has been tested. HMSL can be used for compositions that do not involve MIDI. Since most HMSL users have MIDI equipment, however, we will use MIDI for this tour.

Simple MIDI Commands

Let's look first at the HMSL command environment. You can give commands to HMSL by typing them in the HMSL window and hitting RETURN. This environment is called the *Forth Interpreter.* Commands in Forth are called *words*. Words and numbers in Forth are separated by *one or more spaces*. You can build new words by combining standard Forth words with HMSL words and words you have previously defined.

As an example let's use the HMSL word for *turning on* a MIDI note, MIDI.NOTEON (in this tour we'll put *actual Forth words* in bold when we put them in the text like this). The word MIDI.NOTEON has a *period* in the middle of it and is entered as *one word with no spaces*. It is pronounced "midi dot note on." We'll indent the text that you should enter. Always *end each line* by hitting the <RETURN> key. Enter:

```
    60  80  MIDI.NOTEON
```

You should hear a note begin to play. All MIDI notes are assigned numbers. The 60 indicated "middle-C." MIDI note numbers range from 0 to 127. You will probably want to stick within a range of 36 to 96 (some synthesizers will not play lower than 36, or higher than 96). The second number, 80, referred to the *MIDI velocity*, or loudness. A velocity of 80 will generally produce a fairly loud note. The MIDI loudness range is 1-127.

Notice that the numbers came *before* the command. This is called ***Reverse Polish Notation***. This is a standard feature of Forth. It may seem a little odd at first, but it has some real advantages over traditional notation. Since the parameters for a command have to be determined before the command is given, everything flows in a logical *left to right order.* In addition, mathematicians and logicians like it because it's completely unambiguous — you never need parentheses to group arithmetic operations. However, it may take a bit of getting used to if you haven't seen it before!

If the note you played was on a percussive instrument, the sound died away. If it was a sustaining instrument, the sound is still going. To *turn off* the note, enter:

```
    MIDI.LASTOFF
```

This will turn off the last note turned on. You can *reenter commands* using the cursor keys. Try hitting the "up-arrow" cursor key twice. You should now see the line with **MIDI.NOTEON** before your cursor. Hit RETURN to hear the note. This can save you a lot of typing. Do this again to enter **MIDI.LASTOFF** so the note stops. This feature is called *Command Line History***.** You can edit previous command lines using the left and right cursor keys and the backspace key. Use <shift-leftarrow> to move to the beginning of the line, and <shift-rightarrow> to move to the end. Use the up arrow key twice to get back to the line with **MIDI.NOTEON**. Use the other cursor keys to change the 60 to a 67. When you hit return, you should hear the G above middle C. Use the cursor keys and command line editing to hear some different combinations of MIDI note and MIDI velocity, always using **MIDI.LASTOFF** to turn off the note.

We can *define new words* using the Forth compiler. Instead of having to enter **80 60 MIDI.NOTEON** and **MIDI.LASTOFF** each time, we'll define a word that does that for us. Let's also put a *delay* of 20 *ticks* (about one third of a second, standard HMSL "ticks" are about 1/60th of a second, although you can get much faster if you need to!) between the ON and OFF ofthe note, so that it will have time to sound. New words in Forth are defined using a *colon* ' **:** '. The *name* of the new word *immediately follows* the colon, after a *space*. A definition is ended with a *semicolon*, ' **;** '. For example, enter:

```
:  BANG  (  note -- )
   80  MIDI.NOTEON 20 DELAY MIDI.LASTOFF
;
```

(Note: If for some reason you still hear a note hanging, try typing MIDI.LASTOFF again, or if that doesn't work, MIDI.KILL). This new *program* finished compiling as soon as you entered the line with the semicolon. Note that it contains three simple ideas: turning a note on, waiting 1/3 of a second, and turning the note off. It does not know what note to turn on; that is a number passed to it on the Forth stack when you execute the word. The text between *parentheses* is a *comment* called a *stack diagram*. It shows what is on the *Forth stack* before and after the word executes. To test the new word, execute it with two different MIDI note numbers by entering:

```
60  BANG   63 BANG
```

You should hear two notes (C and Eb), at the same loudness, separated by 1/3 of a second. Experiment with this by using different numbers, or try typing in a variation on **BANG** which has, for example, a 1/2 second delay (30 ticks).

A Simple Example: Playing Random Notes

HMSL has a number of *distribution functions*, or different types of random number generators. The simplest one is **CHOOSE** which gives random numbers *uniformly distributed over a given range*. You give CHOOSE the top of its range as a stack parameter. (Remember that numbers are passed between Forth words on a *stack*, which is explained more thoroughly in the Forth tutorials). We can print the result of **CHOOSE** by using ' **.** ' , a Forth word which is pronounced "dot" and spelled as a *single period*. Try entering the following line several times. (Don't forget to use the Command Line History.)

```
10  CHOOSE  .
```

Notice that the numbers range from 0-9, or10 possible results. Let's use this random number generator in a new word that *plays a random note*. Enter:

```
: RNOTE  12  CHOOSE   60  +  BANG ;
```

This picks a random number from 0-11, adds it to 60, and plays that new note, between 60 and 71, by passing the number on the stack to our old friend **BANG**. You can test this by entering **RNOTE** several times. When you are satisfied that it works, let's put it in a *loop*. Enter:

```
: MANY.RNOTES    (  -- , play many random notes )
   BEGIN   RNOTE
      20 CHOOSE  MIDI.PRESET
```

```
        ?TERMINAL
    UNTIL
;
MANY.RNOTES
```

You should hear a bunch of random notes. Hit the <RETURN> key when you want it to stop. (You may get a message from **DELAY**. This is OK.) Between the colon and the semicolon was the definition of the word (including the comment, showing that it took nothing from the stack, and left nothing on the stack). After the semicolon, the word was executed by simply typing it followed by a return. Notice that we randomly changed the *MIDI Preset* between every note. The HMSL word **MIDI.PRESET** takes one number off the stack. IN HMSL we sometime use the term "MIDI Preset" for "MIDI Program" to avoid confusion with the use of the word "program" as an application. HMSL has a complete *toolbox* for outputting MIDI commands including controllers, pitch bend, pressure and system exclusive messages.

This example was a very simple one that showed how HMSL programs can be written that use different tools to generate "musical events" These simple tests are of course not pieces, just simple examples. We have tried to demonstrate the way that HMSL programs build upon each other.

MIDI Input

HMSL can also respond to *MIDI Input*. To make sure we have the MIDI Input hooked up correctly, enter:

```
MIDI.SCOPE
```

As you play on the MIDI keyboard, you should see a listing of the note numbers as you play them. Also try changing presets and using the pitch bend wheel. Hit <RETURN> when you are through.

HMSL also has a system for *parsing* MIDI commands and mapping them to *user functions*. This means that you can interpret MIDI input to mean whatever you want it to mean. Let's compile a demonstration program that uses the *MIDI Parser* to respond differently to each note in the octave. Enter:

```
INCLUDE  HP:DEMO_PARSER
```

The should take a few seconds, you are compiling a program which already exists. When it is finished compiling, enter:

```
DEMO.PARSER
```

then play notes on the keyboard. (You may want to put the keyboard in LOCAL-OFF mode.) Play each note in the octave to see what they do. If you want to see what the program looks like, enter:

```
TYPEFILE  HP:DEMO_PARSER
```

After you learn HMSL you can edit this and other example files to meet your own needs.

Shapes and the Shape Editor

These previous examples used the *low level* parts of HMSL. Let's now look at a higher level of HMSL, the *morphs*. The fundamental type of morph is the *shape*. A shape contains raw, multidimensional abstract data. This data can be interpreted in many ways. One common way is to treat 3 dimensions of the shape as time, pitch and loudness. This allows us to specify a melody using a shape. HMSL has a graphical editor that allows us to edit these shapes while they are being played. HMSL includes a simple test word called SHEP that is short for **SHAPE-1 HMSL.EDIT.PLAY**. Enter:

```
SHEP
```

A window should open and you should hear two or more notes repeating. This window contains the *Shape Editor*. The grids are called *controls*. The controls are collected together into what is called a *screen*. Find the Control labelled "**Dim**" near the middle of the window. Click on the top of this control where the "up-arrow" is. The number in the box should change to a 1. Now click on "**Draw**" right above this Control. It should highlight. Now click down with the mouse near the left side of the big rectangle and drag the mouse to the right while holding down the mouse button. This should add notes to the melody. The line that is drawn represents one dimension of a shape. The x axis is the index into the shape. The y axis is the value in the shape. Dimension 1, in this case, represents pitch.

Now click on the "**Track**" button near the bottom right of the screen. You should see a vertical line marching from left to right. This indicates which *element* of the shape is currently being played. If the line and the sounds don't seem to always line up, it is because HMSL *precalculates* its output to provide more accurate timing. The user can adjust how far in advance to precalculate. (This is explained in detail in the chapter on *Time and Scheduling.*)

Now click on the "**Select**" button right above "Draw." This puts you in a mode where you can select a range of elements. Click on the line and drag the mouse left or right and release. You should see a highlighted region. Now click on the "**Reverse**" button in the "**Operations**" control. Notice the values in the selected region are reversed. (A full explanation of this editor is in the chapter on Shapes.)

This Shape Editor was built using the User Interface Toolbox. You can build your own custom screens using this toolbox.

Feel free to play around with this a bit. When you are done, click on the close box in the upper left of the window. The word **SHEP**, by the way, is a simple and more or less “traditional” way for HMSL users to see if their MIDI and audio gear are all hooked up correctly!

Looking at Morphs

We can look at the shape we were just editing by entering:

```
PRINT:   SHAPE-1
```

SHAPE-1 is an *object* of the *class* **OB.SHAPE**. The **PRINT:** sends a message to the object that follows telling it to print its contents. **PRINT:** is a *method*. Notice that **SHAPE-1** has 3 *dimensions* or columns. Shapes print statistics about their contents when printed.

Shape are played by objects called *players*. There is a predefined player that **SHEP** uses called **SE-PLAYER**. We can print **SE-PLAYER** by entering:

```
PRINT: SE-PLAYER
```

Notice that it prints different information then **SHAPE-1** yet we used the same **PRINT:** method. The fact that objects "know" how to *respond differently to generic messages* is an important aspect of Object Oriented Programming. Notice that **SE-PLAYER** has something called a *repeat count*. Players can be told to repeat a certain number of times when played. Notice also that **SE-PLAYER** has **INS-MIDI-1** listed as its *instrument*. Players use instruments to *interpret* shape data and to interface with music hardware like synthesisers. Enter:

```
PRINT: INS-MIDI-1
```

Notice that the Instrument has an *Interpreter*. The interpreter is a special function that reads the data in a shape and outputs it as notes, or MIDI system exclusive commands, or graphical information, or whatever *you program* it to do. HMSL has an example of an interpreter that interprets one dimension of a shape as a MIDI preset. Enter:

```
INCLUDE   HP:DEMO_PRESET

DEMO.PRESET
```

When the Shape Editor comes up, use the "Dim" control to get to dimension 3. Select the "Replace" mode and edit the Preset values. Click on the Close Box when done.

Run Some Example Pieces

To get an idea how some of these objects are used together, let's run some example pieces. We have included a number of examples in a directory called "Pieces" or "HP:". many of these examples require several channels of MIDI. If you have a multi-timbral synthesizer, set up several channels starting with channel 1. If you have only a mono-timbral synthesizer, set it to OMNI mode.

XFORMS

The first example plays a simple theme repeatedly on one channel, and on another channel it plays a copy of that theme that is slowly transformed. The transformations include inserting notes between existing notes, deleting notes, and transposing notes and sections. Occasionally the original theme is recopied and development resumes. On a third channel, the developed material is echoed after a lengthy delay. Sometimes the third voice is slowed down. Each channel has a shape containing note information. The theme is developed by manipulating the data in the shape. The best way to understand this is to hear it. Enter:

```
INCLUDE  HP:XFORMS
XFORMS
```

The Shape Editor comes up so you can view and edit the shapes as they are played. Notice that there is a grid called "Select Shape." Click with the mouse on the Up and Down arrows to scroll through the available shapes. When you see the name of one you want, click on the name to display it. If you click on SHAPE-2, you can watch dimension 1 being transformed. Try inserting notes into dimension 1 of SHAPE-1 to change the basic theme. XFORMS will stop after a while. To finish sooner, click on the close box at the top left of the HMSL window. (Note: to Macintosh Users. If the timing seems off, check to make sure *Application Sync* is on in the *PatchBay Desk Accessory.*)

Music for Bookstores

This piece, by Phil Burk, was first performed in a bookstore but is probably not conducive to relaxed reading. It uses an abstract shape that has two dimensions representing *complexity* and *intensity.* As the shape is played, the values of these two dimensions are placed in variables that can be read by other parts of HMSL. Executing in *parallel* with this shape are several *jobs* that are spewing out notes. They are roughly tracking a master melody that is generated using *1/F noise*. The jobs play notes faster and louder when the intensity is high. They also form more harmonically complex chords when the complexity value is high. To hear this piece, enter:

```
INCLUDE HP:BOOKS
BOOKS
```

SUBDIV

This example was suggested by composer Philip Corner. It involves subdividing a time interval into several different equal length pulses. An 8 second "measure" is divided into anywhere from 1 to 20 notes. For each measure a different set of subdivisions is selected. Thus one measure might have 1/4 notes and 1/7 notes, and the next could have 1/3, 1/6, 1/8, 1/11, and 1/14 notes. The example *dynamically creates* 20 shapes, players, and instruments when it starts up (see the chapter on ODE for more about this technique). At the beginning of each measure it decides which of these 20 players to play. This piece works best with percussive timbres like bell and drum sounds. It uses up to 8 MIDI channels. Enter:

```
INCLUDE HP:SUBDIV
SUBDIV
```

Score Entry and Sequencing

Now let's take a look at some of the other *toolboxes* in HMSL. Before we continue, we should make more room in the *Forth dictionary.* You can easily enlarge the dictionary to hold more code, but HMSL is shipped with a smaller dictionary so that it will run on everyone's machine. To *UNCompile* the pieces that we've already compiled, enter:

```
ANEW  SESSION
```

If you get a message saying that **SESSION** is protected, hit 'Y' to continue. This will *forget* all the routines used in the examples we've used up till now. This technique is fully explained in the Forth manuals under the words **FORGET** and **ANEW**.

Now we can compile two HMSL toolboxes: the *Sequencer* and *Score Entry System*. These will be only briefly demonstrated here since they are explained in detail later in this manual. You do not have to type in the things between parentheses, from this point on, we're using comments to help you understand what is going on. Enter:

```
INCLUDE   HSC:SEQUENCER
SEQ.INIT ( initializes sequencer)
SCORE{  ( start score data entry )
```

Don't forget the "curly-bracket" attached to the end of the word "SCORE" — curly brackets are common HMSL symbols for "start-entry" and "end-entry" types of words. First let's try out the Score Entry System. Enter:

```
1   MIDI.CHANNEL!     ( sets current MIDI channel to 1 )
10  TIME-ADVANCE  !  ( for faster response )
PLAYNOW  1/4  A D  1/8  C5 D G A ( play some notes )
PLAYNOW  1/5 G G E E E  CHORD{ 1/4  C  E G A }CHORD
```

You should have heard short melodic fragments. To hear the chord, your synthesizer should be in polyphonic mode. (Note to Macintosh Users: If the notes were too fast, make sure Application Sync is on in the PatchBay Desk Accessory.)

The Text Based Score Entry System allows you to enter complex scores that include arpeggiation, accelerando, crescendo, and parallel play. The scores can be heard immediately or saved in shapes or MIDIFiles. (The SES is described in detail in a later chapter.)

Now let's take a brief look at the HMSL *sequencer*. The HMSL sequencer is not intended to compete with commercial sequencer applications, but we did find that a simple sequencer can be often handy within HMSL. Put your synthesizer back in multi-timbral, multi-channel mode and enter:

```
HMSL
```

Now pull down the "Screens" menu and select "Multi-Tracker." The HMSL sequencer is a 16 track sequencer that can be *customized* by the user. You can record by selecting a "Record" track number then clicking on "Start." Play a few notes then click on "Stop." By repeating this process, you can layer tracks like on a conventional sequencer. The material recorded on these tracks can be pulled out and used in other HMSL programs, saved in a MIDI file, or manipulated by programs running in parallel. The source code for the sequencer is on disk so you can even change the way the sequencer work to suit your own needs. (The sequencer is described in detail in a later chapter.)

Conclusion

Hopefully, this chapter has given you some idea of what HMSL is all about. These features represent just a small fraction of what HMSL is capable of. We realize that we have shown you a lot without fully explaining everything, and this can certainly cause you some initial confusion. Don't worry. This is to be expected at this point, and things will clear up quickly. HMSL is a *large system* and can take some time to learn, but you can do a lot even at the beginning. We hope that we have been able to demonstrate, however, that HMSL offers great potential to those who are willing to learn it. We are continually impressed by the creativity and diversity shown by HMSL users, and are convinced that the more fun something is, the easier it is to learn. Since experimental music is more fun then almost anything else, we think you'll learn quickly.