1987

Creation and performance of music structures

Charles J. Zacky
The University of Montana

THE CREATION AND PERFORMANCE OF MUSIC STRUCTURES

Charles J. Zacky

B.A., University of California, 1974
M.M., University of Montana, 1983

Presented in partial fulfillment of the requirements

for the degree of

Master of Science

University of Montana

1987

Approved by

[illegible]
Chairman, Board of Examiners

[illegible]
Dean, Graduate School

Dec. 7, 1987
Date

12-23-87

Zacky, Charles J., M.S., Dec. 1987   Computer Science

The Creation and Performance of
Music Structures (85 + iv pp.)

Director: Spencer L. Manlove SLM

Recent years have witnessed the development of microprocessor-based musical instruments. More recently, a communications specification has been developed allowing these instruments to be networked together. The Musical Instrument Digital Interface (MIDI) specification has resulted in the development of hardware and software systems designed to play and record data generated by these digital instruments. Systems that record and play MIDI data are called sequencers.

A musical composition is a hierarchy of structures. Current sequencer technology focuses on the objects at the bottom of the hierarchy, notes and phrases. The design of a composition requires the creation and manipulation of structures at all levels of the hierarchy. Therefore, an effective environment for the development of musical compositions must allow the creation and manipulation of high-level musical constructs.

An approach to the definition and performance of music structures is presented. Abstract data structures and operations on them are defined which provide for the creation of music structures. For performance, these data structures are transformed into a tree which more directly represents the hierarchical nature of the music. The tree is traversed during the performance, each node representing a part of the music structure. The contents of a tree node and its attendant operations are defined. An algorithm for traversing the tree during performance is presented.

Acknowledgments

Special thanks to Professor Patrick C. Williams of the Music Department, Jerry Durrin, owner of Bitterroot Music, Philip E. Rosine, and Douglas F. Marsh.

This effort is dedicated to my mother and to the memory of my father.

Table of Contents

Chapter One

Background

1.1. Thesis Overview

This paper discusses a means to organize and manipulate abstract musical structures and to realize these structures in real-time.

Electro-acoustic hardware has evolved from its formative stages. The variety and sophistication of electronic instruments make possible the development of software systems which can concentrate on high-level functions rather than low-level control. The current technological milieu, of both hardware and software, from which this project developed is discussed.

A model of musical structure is presented which reflects the hierarchical nature of music and which reflects a constructive approach as used by those who compose music.

A notation to represent musical structures is introduced and illustrative examples are developed.

The sequencer is a tool for recording, manipulating, and performing music. Current trends in sequencer design are examined. A model is presented which is constructed by associating sequencer functions with the various levels of musical structure. A case study examines an existing sequencer in terms of the model presented.

The requirements for a compositional tool which provides functions for the creation and manipulation of abstract musical structures is presented. It is examined in terms of its representation of the musical structures previously presented and compared with the sequencer model for functionality.

1.2. Chapter Overview

In recent years, microprocessor technology has permeated the electronic music industry, giving rise to a plethora of microprocessor-based devices. Some of these devices are sound producing instruments, others act as controllers or signal processors. The result is that the music technologist of today has music-producing tools of great power available. In response to the development of intelligent instruments, a digital communications specification has been developed which allows dissimilar devices to be interfaced into a single system. This Musical Instrument Digital Interface (MIDI) specification not only provides for the connecting of MIDI-based equipment, but also allows a computer to be part of a MIDI system.

1.3. MIDI Instruments

MIDI instruments can be classified into three broad categories; sound generators, controllers, and signal processors. Sound generators are the most familiar MIDI instrument. The sound may be created by one of two methods, synthesis or sampling. A **synthesizer** contains oscillators which generate periodic waveforms such as sine, square, or triangle waves. These waves are manipulated and combined by other circuitry to produce complex audio signals. A **digital sampling instrument** is like a digital tape recorder in that it digitizes audio signals directly. In general, a sampler has less sound modification functions than does a synthesizer. Samplers are useful primarily for reproducing existing sounds, whereas synthesizers are more flexible in producing new sounds.

There are a great number of synthesizers available. Product uniqueness is partially derived from functionality but more fundamental is the uniqueness of sounds which the synthesizer is able to generate. There are several methods of synthesis currently in use: frequency modulation (FM), additive synthesis, linear arithmetic synthesis, phase distortion, wavetable synthesis, pulse code modulation, subtractive synthesis, and structured adaptive synthesis.

Sampling instruments are characterized by the large amounts of memory which must hold digitized audio signals. Two or more megabytes of RAM is not uncommon on low-end instruments with 8 to 50 megabytes[1] available for more expensive products.

A class of instruments which do not include sound generation capabilities but specialize in sending MIDI data to sound sources are called **controllers**. As the name implies, they control or drive the devices connected to their outputs. Controllers are played in real-time and usually take the form of traditional instruments. The most common controller is the keyboard controller. Other controllers include guitar controllers, wind controllers which resemble clarinets, and percussion controllers.

Related to controllers, in that they do not generate sound directly, are **sequencers**. Whereas controllers resemble

---

[1] The Fairlight Series III has 50 megabytes of RAM, a 600 megabyte hard disk, a 400 megabyte optical storage and a price tag of $175,000.

standard instruments, sequencers can be compared to tape recorders. They store MIDI data for later playback. In addition to recording and playback functions, sequencers offer editing functions. Editing is usually accomplished on two levels, event editing and track editing. Event editing allows modification of individual parameters such as the pitch of a single note. The scope of track-level editing is an entire track. Transposing the pitch of an entire track is an example of a track-level operation.

Signal processors are not new to the electronic music field. In the past they were exclusively analog devices but today are primarily digital. Signal processors are effects devices which produce effects such as reverberation, delay, echo, chorus, flange, distortion, tremolo, stereo panning, compression, and equalization. The most common type of MIDI implementation in signal processors is the ability to call up a predefined effects configuration. Some devices allow the modification of variable parameters through MIDI messages.

MIDI processors are a class of device which includes switchers, mergers, channelizers, converters, and filters. These devices function as modifiers of the MIDI data stream. For example, it is often desirable to reduce bandwidth by filtering out certain types of controller messages.

Synchronizers are used to lock machines together which use differing timing formats. A sequencer and drum machine need to be synchronized to a tape recorder in order to record multiple tracks. For film production, MIDI instruments must be locked with video tape recorders. Performers combining sequenced and nonsequenced material during live performance have devices which translate audio signals into MIDI clock messages enabling the machines to follow the human performers rather than vice versa.

The **computer/MIDI interface** primarily converts the computer output to the MIDI data rate of 31.25 kilobaud. Timing information is often handled by the interface. Interfaces are of two types, dumb or intelligent. The dumb interface handles the conversion to the MIDI data rate. Intelligent interfaces manage the data stream in various ways such as filtering unwanted information.

1.4. MIDI

The Musical Instrument Digital Interface is a hardware and software specification defined by the major synthesizer manufacturers Kawai, Korg, Roland, Sequential, and Yamaha in a meeting held in Japan in August 1983 [Cooper 1986]. A year of discussion by these and other manufacturers preceded this meeting. The MIDI specification allows electronic instruments of different manufacturers to be connected together by the MIDI hardware specification and to communicate with each other by the MIDI software specification. This communication is necessary as even a modest MIDI system will have several components which must interact in real-time. The software specification presents a protocol for the distribution of musical data. This takes the form of a command byte optionally followed by data bytes. It is important to note that digital representations of analog signals is not the information which is sent over the MIDI bus[2], but rather it is control information directing a device to perform some action.

The MIDI command structure is outlined in figure 1-1. It is broadly divided into two categories, channel messages and system messages. System messages are received by all devices in the message's path. Channel messages have an encoding which specifies one of sixteen channels. The individual devices in the data chain may be configured to respond to a single channel. Thus, a single device or group of devices may be addressed.

[2] Waveform information such as that used by a digital sampling instrument can be sent over the MIDI bus, but this is different from sending a digitized audio signal for real-time processing.

Fig. 1-1: MIDI Command Structure

I. Channel Messages

  A. Voice

    1. Note-Off
    2. Note-On
    3. Polyphonic After-Touch
    4. Control Change
    5. Program Change
    6. Channel After-Touch
    7. Pitch-Bend

  B. Mode

    1. Local Control
    2. All Notes Off
    3. Omni Off
    4. Omni On
    5. Mono Mode
    6. Poly Mode

II. System Messages

  A. Common

    1. Song Position Pointer
    2. Song Select
    3. Tune Request
    4. End of System Exclusive

  B. Real-Time

    1. Timing Clock
    2. Start
    3. Continue
    4. Active Sensing
    5. System Reset

  C. System Exclusive

The channel voice messages make up the bulk of the MIDI data that is transmitted as they specify the primitive elements of musical material, such as pitch, loudness and sound modifiers. A basic element of music is pitch. The channel voice messages provide two pitch-related commands, **Note-On** and **Note-Off.** A component of each of these commands is a note number in the range 0 to 127. Note number 60 corresponds to middle-C on the piano keyboard. The note number does not actually represent a specific pitch, but rather refers to a key on the keyboard. The actual pitch depends on the setup of the sounding device. The final data byte following the Note-On/Off commands indicates the velocity with which the note is struck or released. This produces gradations in volume such as accent, diminuendo, or crescendo.

Two sound modifier commands effect notes which have already been struck and are currently being held. **Polyphonic Key After-Touch** requires two data bytes to specify a key number and a pressure value. **Channel After-Touch** requires only a pressure value as it establishes one overall level for the entire keyboard. The pressure value may represent various parameters such as volume level, modulation level, low frequency oscillator (LFO) speed, or timbre depending upon the receiver's configuration.

Control Change implements a wide range of functions. The data bytes which follow this message specify a controller number and a controller position. This command allows physical knobs, dials and buttons on an instrument to be set under software control. Of the 128 possible controller numbers, a few are defined by the MIDI specification, others may be defined by individual manufacturers for particular devices. Some of the predefined controllers include the modulation wheel, breath controller, main volume, portamento time, sustain pedal, data increment, and data decrement.

As MIDI usage has evolved, the limitations of the original specification have become apparent. One such limitation is the limit of 128 addressable controllers. A very recent extension to the MIDI specification has greatly expanded the number of addressable controllers [Cooper 1987]. This new area of controllers is divided into two main groups, registered and non-registered. Registered controller numbers must be approved by the MIDI Manufacturers Association (MMA) and the Japanese MIDI Standards Committee (JMSC). The non-registered controller numbers are available to manufacturers to be used as they wish. Over 16,000 controller numbers are now available in each of the newly defined controller areas. This was accomplished by setting aside four of the previously undefined Control Change numbers. Two numbers specify the least significant byte (LSB) and most significant byte (MSB) of a non-registered controller number and two specify the LSB and MSB of a registered controller number. This gives 14 bits of resolution and thus 16,384 possible controllers.

As MIDI usage has become widespread, MIDI has been implemented on devices that had not been considered before. A mixing console used in a recording studio or for live performance has many parameters that must be varied during the course of the performance. A lighting controller may have several hundred parameters which control a multitude of lights. MIDI provides a way to automate the operation of both of these devices and others such as home control systems. Examples such as these prompted the expansion of the MIDI specification.

The single data byte following a **Program Change** message selects a particular patch or voice number in an instrument. A patch contains the timbre or tone color that a sound producing instrument will use. The patches are stored in memory and are referenced by a patch number. This approach is very general, as the voice defining parameters need not be transmitted, but rather a predefined voice is selected.

Pitch-Bend is actually a controller but was given its own command. The two data bytes which follow this command indicate the position of the pitch wheel. As data bytes always have the eighth bit clear, this gives 14 bits of resolution. This increased resolution may be the reason it was given its own controller status. Pitch-bend is the process of gradually altering the pitch of a note or group of notes. Examples of pitch-bend are a trombonist sliding up to a note or a guitarist bending a string.

Channel Mode messages are configuration commands which determine which channel the receiver will respond to. **Omni On** mode allows the receiver to respond to voice messages on all channels. **Omni Off** mode requires the receiver to respond to only one transmitted channel.

MIDI Real-Time messages provide a means to synchronize devices in a MIDI system. Usually, one instrument is designated as the system clock with the other instruments slaved to it. The **Timing Clock** is a pulse which is sent at a rate of 24 clocks per quarter note. Thus, it determines the tempo of the system. Instruments which contain sequencing functions such as sequencers and drum machines can lock onto the system clock. Other Real-Time commands allow a system to **Start** and **Stop** together as well as resume (**Continue**) from the location of the most recent Stop command.

The original MIDI specification has no provision for the timing of events. Any automated electronic device which sends MIDI data as control information must provide its own format for handling absolute time information. MIDI systems have become commonplace in the film scoring and television industries. These industries have a code (SMPTE time code) that has been in use which specifies absolute time intervals between events. This is used to synchronize music and sound effects to film. Recently (April 1987), an extension to MIDI called the **MIDI Time Code** (MTC) was approved. MTC defines a specification for absolute time intervals. This is used in interfacing MIDI instruments with SMPTE devices.

System Exclusive messages provide a flexible and open-ended means to implement a variety of MIDI functions. Each manufacturer defines the system exclusive codes that will be used by their instruments. It consists of a header which includes the manufacturer's identification number, a variable number of data bytes, and an End of Exclusive byte which terminates the message.

Nearly all MIDI instruments store configurations such as voice definitions or data such as drum machine patterns in random access memory. System Exclusive messages provide a means whereby the contents of memory can be downloaded to an external storage device. Many MIDI implementations contain System Exclusive codes which duplicate the functions provided by front panel controls. Voice librarian software uses these MIDI functions to allow voice definitions to be entered at a computer which can have a much friendlier interface.

Chapter Two

Music Structure

2.1. Overview

Music has structure. Structure is evident in the melodies of children's nursery rhymes which repeat for each new verse. Structure is evident in the organization of the movements in a Haydn symphony. Throughout music history, composers have written structured music. Many of the archetypical musical structures have been formalized by theorists and studied by music students. Forms such as the isorhythmic motet, sonata-allegro, rondo, and fugue are a part of the modern composer's knowledge base. These formal structures point out the hierarchical nature of music. The specific formal structures which have been created during the course of music's history are not the particular concern of this chapter. Rather, it is the notion of the hierarchy that these structures imply which is discussed here. The concern is how the structures are put together. What are the pieces and how are they arranged?

2.2. Components of Musical Structure

The primitive element of the musical hierarchy is the **note**. The note is multi-dimensional as it is composed of several elements which exist and may vary in time. A note has duration. Duration implies an onset time and a termination time. The duration is, of course, the time in between these two events. Thus a note exists in time. A note has pitch. This is a fundamental attribute. A note will often be abstracted to its pitch as when doing a harmonic analysis. Pitch remains relatively constant for the duration of the note although minor fluctuations are common, as when a violinist adds vibrato to a held note. For a note to be perceived, it must have some volume, so loudness is another element of a note. A note's loudness need not remain constant for its duration. Finally, the note must be produced by some sound source, so it must have a timbre. Timbre is the characteristic quality of a sound which distinguishes one instrument from another.

A **phrase** is defined as a division of the musical line, comparable to a sentence of prose [Apel 1972]. In formal music study, there are many types of phrases. The technical definition of phrase is not important here. A merely descriptive definition will suffice. A collection of notes which are heard sequentially, expressing a complete thought, and which are usually performed by a single instrument is called a phrase. Phrases are defined in terms of notes, and thus constitute the next level of the musical hierarchy.

Phrase attributes are determined by the attributes of its individual members. If each successive note in a phrase is slightly louder than the previous note, the phrase gradually crescendos.

A collection of phrases is called a **part**, the next level of the hierarchy. What the trumpet plays during a composition is the trumpet part, the violin plays the violin part, etc. This is an obvious and somewhat insubstantial definition, but it nevertheless describes an existing phenomena and is a necessary component of musical structure. The distinction between a phrase and a part is vague, as a part may consist of a single phrase.

The note and the phrase may be thought of as the primitive structures, out of which the higher-level structures are composed. What makes a structure high-level is its ability to be defined in terms of other structures of the same type. A phrase is always concatenated notes but a part is a concatenation of phrases and other parts. As an illustration, assume four phrases are concatenated. This could be described as one part composed of four phrases, or one part composed of two subparts which are each composed of two phrases, etc. The distinction is merely one of intent and descriptive convenience.

It is difficult to give attributes to parts as they may vary greatly over the course of the composition. A general observation is that a part is performed by a single instrument.

So far, only sequential structures have been considered; a phrase is a series of one or more notes, a part consists of the concatenation of phrases. Music is more than the succession of individual notes. A string quartet has four performers who often play simultaneously. An orchestra may sound many notes at the same time. A collection of parts sounding concurrently constitutes the next level of the music hierarchy, the **section.** A piece of music may contain several sections, as in a song which alternates between verse and chorus, or a symphony whose movements could each be considered a section.

2.3. Notation

A means of visually representing music structures is necessary for further discussion. Computer programs have structure. A well written program can be read from the top down, each level filling in more detail. Just as a part is made up of phrases, so a module is defined by references to lower level modules. A program written for a computer in a structured language is a hierarchical structure. The similarities between program structure and music structure indicate that a programming language may be an expressive medium for musical structures[3]. Parts and sections are differentiated primarily by their temporal characteristics; parts are sequential structures, sections are parallel structures. A programming language must have constructs which express concurrency as well as sequential structures. The musical examples presented in this paper are given in a psuedo-code based on the Occam language [Pountain 1984].

The basic unit of the Occam language is a "process" which performs a sequence of actions. The **PROC** keyword begins the declaration of a process. For the structural description language contained herein, a process contains the information necessary to perform some part of the musical hierarchy. What appears as a procedure call in a traditional language, is actually an invocation of a

[3] This is verified by the existence of several music-description languages. See [Byrd 1974, Gourlay 1986, Maxwell 1984, Smith 1973].

substructure to begin its performance.

Notes are the foreground of music. They are responsible for nuance and detail. However, the domain of structure is background, the large-scale movement of music. So, notes are not included in the representations to follow. Phrases, parts and sections are all assumed to be made up of notes at the lowest level.

We assume the existence of a primitive type, **PHRASE**, which is the same phrase structure described above. The contents of each PHRASE type is described in a comment when it is declared and given in traditional music notation in the appendix.

A sequential structure represents a part and a parallel structure represents a section as previously defined. For structural clarity and continuity with developments in later chapters, a substructure is either a parallel event or a sequential event. The two types of processes are not mixed together. More concretely, a process declaration contains either the keyword SEQ or the keyword PAR but not both.

Listing 2-1 shows how parts are represented. To indicate their sequential nature, the **SEQ** construct is used. The invocations following the SEQ keyword are executed in sequential order. In this example, the parent node, Part, is realized by first performing its first child, A. When the first child has completed, the second child, B, is begun. The completion of the last child signals the termination of the parent. The colon indicates the end of the declaration.

Listing 2-1: part representation

```
PROC Part =
  SEQ
    A()
    B():
```

Listing 2-2 illustrates a section node. The processes following a **PAR** keyword are executed concurrently. To realize the parent, Section, all its children are begun at the same moment. The parent is completed when all its children have completed.

Listing 2-2: section representation

```
PROC Section =
  PAR
    A()
    B():
```

Attributes may be attached to each invocation of a structure. Attributes will be more thoroughly discussed in later chapters. When an attribute, other than the default, is associated with a structure invocation, it is indicated by naming the attribute as an argument and assigning it a value. Listing 2-3 shows that A is played twice and that B is transposed up three semitones.

Listing 2-3: attribute assignment

```
PROC Attributes =
  SEQ
    A( repeat    = 2 )
    B( transpose = 3 ):
```

Listing 2-4 presents a complete example[4] which includes each type of structure. The example represents the structure of a major scale, played one octave ascending and descending, harmonized at the minor sixth below. The primitive elements, phrases, each consist of a one octave scale, ascending or descending.

Working from the highest level down, the root process, _Scale_, is composed of two processes, _Up_ and _Down_, played in sequence. These structures are the harmonized scale, ascending and descending. The process _Up_, which is the harmonized ascending scale, consists of the two PHRASES _TonicUp_ and _SixthUp_ performed concurrently. Likewise, the structure _Down_ is composed of _TonicDown_ and _SixthDown_ played concurrently.

Note that the representation for a structure need not be unique. Alternative descriptions exist and may be chosen

[4] The appendix provides traditional music notation for several examples presented in this chapter.

Listing 2-4: Scale

```
PHRASE
  TonicUp,   --a major scale ascending one octave.
  TonicDown, --the same major scale descending one octave.
  SixthUp,   --TonicUp harmonized a minor sixth below.
  SixthDown: --TonicDown harmonized a minor sixth below.

PROC Up =
  PAR
    TonicUp()
    SixthUp():

PROC Down =
  PAR
    TonicDown()
    SixthDown():

PROC Scale =
  SEQ
    Up()
    Down():
```

for their clarity and expressiveness. Listing 2-5 shows the previous example, but emphasizes the harmony (Tonic and Sixth are high-level nodes), whereas the previous figure emphasized the up and down motion.

2.4. Two Examples

The next two examples will further illustrate the use of a structural description language to represent music structures. Later chapters will refer to and expand upon the examples presented in this section. The examples have been chosen for their brevity and simplicity. They each address different problems in organizing music, as will be

Listing 2-5: ScaleHarmony

```
PHRASE
  TonicUp,    --a major scale ascending one octave.
  TonicDown, --the same major scale descending one octave.
  SixthUp,    --TonicUp harmonized a minor sixth below.
  SixthDown: --TonicDown harmonized a minor sixth below.

PROC Tonic =
  SEQ
    TonicUp()
    TonicDown():

PROC Sixth =
  SEQ
    SixthUp()
    SixthDown():

PROC ScaleHarmony =
  PAR
    Tonic()
    Sixth():
```

demonstrated in later chapters.

The first example is a piano prelude. It is a monophonic composition and so, contains only sequential structures. Taking a top-down approach, the high-level structures are presented in listing 2-6. The prelude is a tripartite form, ABA. For brevity, only the B part is detailed here. The B part forms a triangular-shaped symmetrical structure, abcba. Each of its component parts is composed of four iterations of an individual substructure.

Listing 2-6: Prelude, C.J. Zacky, high-level structures

```
PROC a =
  SEQ
    a'()
    a'()
    a'()
    a'():

PROC b =
  SEQ
    b'()
    b'()
    b'()
    b'():

PROC c =
  SEQ
    c'()
    c'()
    c'()
    c'():

PROC B =
  SEQ
    a()
    b()
    c()
    b()
    a():

PROC A =
  SEQ
    .
    .
    .

PROC Prelude =
  SEQ
    A()
    B()
    A():
```

Listing 2-7 shows the contents of the low-level structures. Each of the lowest level routines from the previous listing, <u>a′</u>, <u>b′</u>, and <u>c′</u>, are similar in structure; the concatenation of some form of <u>x</u> with the PHRASE <u>y</u>. The phrases, <u>x</u> and <u>y</u>, contain only two notes each, so the prelude is an example of creating a relatively large structure from very few, simple pieces.

Listing 2-7: Prelude, low-level structures

```
PHRASE
   x,    --left hand two note motive
   y,    --right hand two note motive
  x2,    --x transposed up 2 semi-tones
  x3:    --x transposed up 3 semi-tones

PROC a′ =
  SEQ
    x()
    y():

PROC b′ =
  SEQ
    x2()
     y():

PROC c′ =
  SEQ
    x3()
     y():
```

Listing 2-8 is the structure of a canon from <u>The Musical Offering</u> by Johann Sebastian Bach. A canon is a polyphonic composition in which a leading voice is strictly imitated by a following voice or voices. A common type of canon is the round, of which <u>Three Blind Mice</u> is an example. The following canon has two imitative voices, played by violins, set over a third, non-imitative voice, which plays the theme given to Bach by King Frederick the Great. For this example, we assume a process called rest, which outputs silence information for the number of beats contained in its argument.

Listing 2-8: Canon, J.S. Bach

```
PHRASE
  Theme,     --written by King Frederick the Great
  Violin,    --canonical voice
  FirstNote: --first note of the Theme, needed for ending

PROC KingsTheme =
  SEQ
    Theme()
    Theme()
    Theme()
    FirstNote():

PROC Violin1 =
  SEQ
    Violin()
    Violin()
    Violin():

PROC Violin2 =
  SEQ
    rest(4)
    Violin1():

PROC Canon =
  PAR
    KingsTheme()
    Violin1()
    Violin2():
```

It is evident from the top-level module, Canon, that the canon consists of three parallel events, the king's theme, and the two violin parts. The module KingsTheme, contains three repetitions of Theme. In order to provide an ending, the KingsTheme must wrap around to the beginning again and play the first note of the theme.

The first violin part (Violin1) is straightforward. It consists of three repetitions of the canonic voice, Violin. The second violin (Violin2) is the element of interest in this example. It is the first violin part delayed by four beats. This creates a temporally overlapping structure in which Violin1 terminates before Violin2. Suppose another three-voiced structure is concatenated to the end of Canon. Should its first voice begin execution as soon as the canon's first voice terminated, thus creating another, possibly unintentional, overlapping structure? This type of behavior is addressed in chapter four.

Chapter Three

Sequencer Structure

3.1. Overview

As sequencer technology has advanced, musicians have adopted it into their collection of tools. It is now an integral part of many musicians' professional lives. It has been incorporated into live acts and film scoring. It excels as a compositional sketch pad, performing many of the same functions as a multi-track tape recorder but with superior editing capabilities. It has been compared to both a player piano and a word processor. It is like a player piano in that it automatically plays the music which has been programmed into it. Its music editing capabilities make it comparable to the use of a word processor in arranging and perfecting text.

A sequencer records MIDI data, not audio signals. The data can then be manipulated in ways not possible with audio signals. Wrong notes can be corrected. Rhythmic inaccuracies can be repaired. Music production is no longer tied so closely to the physical process of performance.

A brief description of what a sequencer is was presented in chapter one. The present chapter examines current sequencer technology by examining functions provided by sequencers. A case study of a commercial sequencer is presented, followed by a look at its suitability for programming the examples given in chapter two.

Two new terms are needed in this chapter. They are related to terminology introduced in chapter two.

An **event** is a complete MIDI message as described in chapter one. Just as a note was the primitive element described in chapter two, a MIDI event is the primitive element in music generation using MIDI. Notes were described as single multi-dimensional elements having several attributes. MIDI has no single event to correspond to a note. Indeed, it requires two midi messages, Note-On and Note-Off, to produce a single note. Attributes associated with notes are shared by several MIDI event messages. For example, to add vibrato to a held note requires controller data indicating the changing position of the modulation wheel. Thus, a single note has required a Note-On message, some number of controller messages, and a Note-Off message. The idea of a MIDI event is more general than the concept of a note. MIDI events generally refer to MIDI voice messages, but all MIDI messages may be considered events.

A **track** is a sequence of MIDI events. It is a term inherited from analog tape recorder technology where it designates one of the parallel recording surfaces put on magnetic tape during the process of recording. In MIDI terminology, a track consists primarily of Note-On/Off events but other note modifying commands may be included. A track corresponds to the phrase of chapter two where it is a sequence of MIDI events usually generated by a single source.

3.2. Sequencer Functions

3.2.1. Input/Output Functions

As the primary function of the sequencer is the recording and playback of MIDI data, input and output functions must be provided. Input functions take two forms:

real-time and **step-time** entry.

Real-time entry is similar to the traditional method of input to a tape recorder. The MIDI instrument is connected to the sequencer's input and the record standby mode is activated. After an optional metronome lead-in, the record mode is activated and the performer begins to play. Recording may be terminated after a preselected number of beats or measures, or when a "stop record" command is given. The material has then been recorded onto a single track in the sequencer where it is now ready for playback or editing.

Options for real-time input may be available. A **punch-in** function may be provided. This is used on a previously recorded track where a portion of the track is not acceptable. The track begins to play. When a preselected point is reached, the track material is muted and recording automatically begins. The newly recorded material replaces the prior contents of the selected portion of the track. This is the same type of punch-in available on multi-track tape recorders.

Often, only certain types of information, such as Note-On/Off messages, may be wanted. Controller messages typically use up a great deal of bandwidth. It may be desirable to filter these messages from the input stream. A **filter** function eliminates selected MIDI messages as they appear at the sequencer input.

Step-time entry is a means of entering MIDI data in "slow motion". The actual entry mechanism can take different forms. A MIDI instrument's keyboard can be used to specify the note numbers, or the sequencer may provide a means of direct entry without using an external keyboard. A single duration may be specified in advance with all input notes taking on that value until another is entered, or each note may require its duration be entered separately. Step-time is useful for entering passages too complex for the human performer.

Output functions allow the performed material to undergo transformations during playback without affecting the actual data stored within the sequencer. Tracks may be selectively muted or unmuted to allow review of only a portion of the recorded material. Looping causes a section of data to be repeated continually or for a specified number of hearings. This is useful for rehearsing a part which will be recorded in parallel (overdubbed) to the looping material. It may be desirable to direct output to a different receiving instrument. Typically, individual instruments are configured to receive on specified channels and are configured with different timbres. The ability to **channelize** an output stream allows the musician to compare a part played by different voicings by directing output to another instrument.

3.2.2. Event-Level Functions

The lowest accessible level available for editing is the event. The friendliness of event-level editing varies widely. The most primitive is a display of data as a list of hexadecimal numbers. Other implementations may display the data as a graph which shows relative durations and pitches, or infrequently, traditional music notation. In between these extremes is representing the data as a list, but instead of hexadecimal numbers, the data is translated into English and shows what the data represents. A hypothetical display of a MIDI data stream is shown in figure 3-2. The first event shows a C in the fifth octave struck with a velocity of 64 on the first beat of a measure. The second event shows the release of the previous note on the second beat. This is still rather primitive but is probably the most common method of displaying a MIDI data stream.

Fig. 3-2: Sample MIDI Data Display

```
1:00    C 5    64    NoteOn
2:00    C 5     0    NoteOff
```

Event-level operations include inserting an event into the stream, deleting an event, and modifying one of the data values such as changing the velocity value.

3.2.3. Track-Level Functions

A common and useful operation to apply to a track is **transposition:** the process of shifting all pitches up or down by the same number of half steps while maintaining their positions relative to each other. As the sequencer has no knowledge of tonality, the transposition is real rather than tonal. Internally, a constant is added to every note message. Since MIDI recognizes note numbers from 0 to 127, any transposition which would place a note outside of this range must be folded back within range. This may be handled by replacing the out-of-range transposed note with the same pitch-class but of the nearest octave which is in range.

Quantization is a very powerful function which is used to correct timing inaccuracies. It may be desirable to have a passage performed with rhythmic accuracy unobtainable by the performer or to clean up a rhythmically sloppy performance. Quantization will cause note onset times to fall on multiples of the quantization value. If the eighth note is the quantize value, then all notes will begin on exact multiples of the eighth note.

There are several types of quantization. Note-On commands may be quantized but the corresponding Note-Off is not. This will change the duration of the note. It is possible that a Note-On command moved backward in time will overlap its corresponding Note-Off. This is a situation that an 'intelligent' sequencer should be aware of.

The entire note may be moved in line with its quantization value. The duration remains as it was but the note is shifted in time. A problem arises with events such as Pitch-Bend which may occur between the Note-On and Note-Off. These intervening events can be shifted along with the note or retained in their original position.

Both Note-On and Note-Off commands may be quantized. This will either stretch or squeeze a note. Having both the attack and release times quantized may give the music a mechanical feel.

Merge is a function that combines several tracks into one. Some sequencers have only a limited number of available tracks. When it is necessary to make more recording passes than there are available tracks, the merge function can be used to compress the data from several tracks into one. More information can then be recorded on the free tracks. The ability to merge tracks can be useful in more creative situations than that described above. One pass may record only Note-On/Off information. A second pass may record Pitch-Bend or modulation wheel events. The controller information can be edited or recorded several times until it is satisfactory. At this point, it can be merged with the Note-On/Off information. This allows the musician to work with isolated types of data, making editing easier.

Track shift is a function which moves a track temporally in one direction or the other by a fraction of a second or fraction of a beat. This is sometimes necessary to compensate for timing discrepancies when synchronizing to tape or in large systems where the MIDI signal is passing through many instruments causing the signal to be delayed by a small amount. Track shift can be used to introduce an intentional delay effect by copying a track and playing both copies back simultaneously but with one track shifted with respect to the other.

A velocity value is part of the Note-On command. The velocity gives a relative dynamic to the note. The velocity values of an entire track may be modified to give a different dynamic contour. Corresponding to transposition of notes, scaling adds a constant factor to each velocity value, thereby raising or lowering the overall dynamic level of the track while maintaining its internal relationships. The velocity values may be clamped to a single constant value. The basic contour may be maintained but the overall dynamic may be compressed or expanded. Finally, the track may be modified to create smooth crescendos or diminuendos.

A **filter** function removes unwanted data from the data stream. Continuous controllers output a great deal of information. It is often desirable to filter out this information. Some sequencers allow a percentage of the specified MIDI message to be removed, thus thinning out the data. A few sequencers permit the entry of complex search criteria, such as removing every note above or below a certain range.

3.2.4. High-Level Functions

Track-level functions are implemented on virtually every sequencer. Many have some form of event-level editing. It is rare for a sequencer to have any but the most basic functions available for manipulating musical data at a high-level.

Much music is composed of sections that reappear throughout the composition. An advantage of high-level editing functions is that these sections may be recorded separately and then rearranged and repeated as necessary by a pilot track. A pilot track keeps track of the order in which sections will be performed [Garvin 1987]. Pilot tracks are usually called **link** or **chain** operations. Chaining will be discussed further in the case study.

3.2.5. Performance Directives

The sequencer performs music in real-time, so in addition to editing functions, it should include some means of directing the performance, much like a conductor does. The most basic performance directive found on a sequencer is a tempo control. At its most primitive level, a single tempo may be selected for the duration of the performance. More advanced implementations allow both sudden and gradual tempo changes to be specified throughout the music.

Some sequencers record the MIDI data that they receive without regard to a time signature. Others include measure information in the track data stream. Time signatures are important in at least two situations: when a metronome is used during recording and when editing options are specified by measure number. In these cases it is important to be able to vary the time signature. When recording a piece with multiple time signatures, it is desirable to set up a click track. The metronome can then produce an accent on the downbeat of each measure which helps the musician and machine stay synchronized.

3.3. Case Study: Texture by Roger Powell

Texture is a sequencer program written for the IBM-PC family of computers. For the computer-to-MIDI interface, Texture requires the Roland MPU-401 MIDI Processing Unit. Texture was selected for an in-depth look for two reasons. Most important is its functionality as a sequencer. It implements an impressive number of the standard sequencer features as well as going beyond most sequencers in its implementation of high-level structural manipulation functions. Second, it has achieved widespread popularity with both amateur and professional musicians. Jan Hammer, former keyboardist with Sarah Vaughan, currently uses Texture in producing the sound track to the <u>Miami Vice</u> television series [Milano 1987].

3.3.1. Event-Level Functions

Texture provides basic event-level operations such as insert, delete, and modify values within an event. It has a match function which will find the Note-Off event that corresponds to the selected Note-On event and vice versa. This is useful when deleting a note or modifying its pitch. There is also a locate function which moves the current editing position to the specified beat. The Texture event edit display is similar to the above example (figure 3-2).

3.3.2. Track-Level Functions

Certainly one of Texture's strengths is its implementation of a wide range of track operations. Following is a list of Texture's track operations with brief commentary where appropriate.

Advance (track shift)
Blend (merge) tracks
Block Copy: copy data to a new position
Block Move: move data to a new position
Copy
Erase
Fill: replicate data within a track
Filter unwanted MIDI data
Name
Play
Quantize
Scale event start times, note velocities and durations
Splice: append one track to another
Transpose
Undo previous track operation

3.3.3. High-Level Functions

Texture was chosen for this case study because it has the most complete implementation of high-level functions of all the sequencers studied. Texture recognizes the hierarchical nature of music as its organization shows. The highest level musical structure in Texture is called a **song**. Texture can work with one song at a time. A song is composed of **links**. Links are a list of **patterns** which are ordered by the musician. A pattern is made up of **tracks**. There are twenty-four tracks per pattern.

We can see that Texture goes beyond the simple multi-track tape recorder analogy which is so prevalent among sequencers. It allows the sectional organization of music. A pattern is the basic section. It can contain up to twenty-four parallel tracks. Sections can then be linked together in any order to produce a song. Linking is a sequential operation. Linked patterns form a list. The patterns in the list are performed in sequence.

It is appropriate that the highest level structure in Texture is called a song. Its limited implementation of the musical hierarchy, not allowing nested structures, lends itself to composing in the song form.

3.3.4. Performance Directives

Texture offers several performance directives associated with links. There is a global tempo that is in force when a song begins play. In addition, each link may include a relative tempo and rate. A relative tempo change modifies the tempo by some ratio, such as doubling or halving the tempo. The rate determines how quickly the change takes effect, from immediately to very gradually, thus permitting accelerandos and ritardandos to appear in the music.

A link may be repeated up to 255 times in succession. A track or group of tracks may be muted within the link. A link may also have a transposition constant associated with it.

3.3.5. Texture Application

A good tool gives its user a natural and intuitive means to express that which must be accomplished. Texture's format is geared toward composing in song form. The strophic nature of song form is readily compatible with the track/pattern/link/song format of Texture. It is instructive to apply Texture to the two previous examples, <u>Prelude</u> and <u>Canon</u>, to see what the difficulties are in using it to piece together music structured in other ways.

The first difficulty is at the phrase level. A Texture pattern can be between 1 and 545 beats long [Powell 1986]. The two sixteenth note motives, <u>x</u> and <u>y</u>, are each only one half of a beat in length. The originally conceived structure must be modified to contain no phrases of less than a single beat. If the low-level structures of listing 2-7, <u>a'</u>, <u>b'</u> and <u>c'</u>, can be created at the track level, either by real-time or step-time entry, then the single beat pattern length requirement will be satisfied. Because Texture does not allow nested structures, the high-level structure of listing 2-6 must also be modified. Essentially, the structure must be flattened. To simplify the resulting structure, recall that a link may be repeated up to 255 times. The resulting structure (the two A sections are omitted entirely) is given in listing 3-9.

Listing 3-9: Prelude, Texture version

```
PHRASE
  a',    --a' = x  + y
  b',    --b' = x2 + y
  c':    --c' = x3 + y

PROC Prelude =
  SEQ
    a'( repeat = 4 )
    b'( repeat = 4 )
    c'( repeat = 4 )
    b'( repeat = 4 )
    a'( repeat = 4 ):
```

The resulting structure is quite a bit shorter than the original. What is lost are several levels of abstraction, which does not make a very big difference in a piece of this size but would suffer in larger structures. Also, manual entry was complicated, more preliminary work outside the program was required.

It is difficult to represent overlapping structures, such as those found in the Canon of listing 2-8, in a strophic-based model. There is no way to do so while preserving the original structural model. Using the track copy function of Texture, it is possible to create Violin1 by replicating Violin until there are three concatenated copies. The track copy command can create Violin2 from Violin1 with the addition of adding four beats of silence onto the beginning of Violin2 using event-level editing commands. All abstraction is lost. Clearly, a more flexible sequencer model is needed to create and manipulate the full range and complexity of musical structures.

Chapter Four

Definition and Performance of Music Structures

4.1. Overview

A discussion of the hierarchical nature of music was presented in chapter two, where it was shown that a music structure is composed of a number of substructures, each of which is itself composed of other substructures until further decomposition is prevented by the occurrence of a structural primitive. In chapter three, current trends in sequencer design were examined. From that discussion, it is evident that the issue of the representation and creation of abstract music structures has not been adequately addressed. Yet, an effective environment for the development of musical compositions must allow the creation and manipulation of high-level musical constructs.

To further investigate this issue, the author has developed a sequencer which provides operations on high-level structures. This sequencer is an interactive tool that allows structures to be created and performed in real-time. Two subsystems, one that provides operations for the creation and manipulation of structures, and the other which traverses the structure during real-time performance, are of particular interest and are described below.

4.2. General Description

The manipulation of abstract music structures involves two main functions: defining the structure and performing the structure. In the author's implementation, the structure is defined interactively. A simple command language is recognized by the interpreter which then transfers control to the appropriate routine for execution. Upon receipt of a command to perform a structure, the structure definition is transformed into a tree data structure which contains the information necessary for performance.

The problem of creating a hierarchy of structures was alluded to at the end of the previous chapter. How should a structure which consists of three parallel tracks be joined to a structure of two parallel tracks? What is the result of the concurrent performance of a track whose duration is eight beats and a track of four beats duration? How will the above structures be connected to others?

To create a complex structure involves the ability to create simple structures, combine them into other structures, and use the resulting structures to form yet other structures. This is the same procedure used in creating complex program structures, where a high-level module may trigger the invocation of many lower level modules which must complete before returning to their parent module. In programming, certain details are known about module behavior, such as the point to which a called module will return control to its parent. The behavior of music structures must also be known in order to create well-behaved compositions.

A "black box" approach is used to define structure behavior. In computer science, the black box approach to program building means that the implementation details of a module need not be known by any but its implementer. The function of the module and its interface is all the information required to include it in a program structure. In building music structures, the black box analogy is taken literally; the structure forms a figurative box or rectangle in its two dimensions of duration and number of parallel tracks. A behavior has been specified for the duration and width of a structure whose components may not be of the same durations and widths.

For structures executing concurrently, the total duration is equal to the duration of the longest structure. The total duration of structures executing sequentially is equal to the sum of the individual structure durations.

Width has to do with the number of tracks which are being performed concurrently. For structures executing concurrently, the total width is equal to the sum of the individual structure widths. The total width of structures executing sequentially is equal to the width of the widest structure.

A substructure may occur more than once within another structure. An obvious example of this is the rondo with its ABACA formal structure. However, rarely is it desirable to repeat a structure verbatim. In the interest of variety, some variation is required. Thus, it is necessary to attach performance attributes to each occurrence of a structure. A performance attribute is a value given to a variable parameter that is specific to a particular occurrence of a structure. The attributes which are built into this sequencer implementation are described below. A specific occurrence of a structure is called an instance. An instance consists of a reference to the structure's formal definition and the corresponding attribute values. To summarize the "black box" approach to music construction, each box has two dimensions, duration and width, which are determined at definition time; and each box has attributes associated with it which specify performance details unique to that particular instance.

The data structures that are created while the composition is being defined are not in a form suitable for performance. When the "play" command is given, the specified structure is expanded and transformed into a tree data structure which has performance information in a form accessible for real-time performance. The tree represents the structure of the composition and is particularly suited for the type of traversal needed to realize the music. Traversal begins with the root node and proceeds down to the leaf nodes and then back up to the root again. The path taken varies according to the type of nodes encountered during traversal. If a structure is composed entirely of nodes which are sections, the traversal will be breadth-first. Since the children of a section are performed concurrently, the nodes of a tree of sections will be visited level-by-level. At the other extreme, a tree made up entirely of part nodes is traversed depth-first. The nodes of an individual branch are visited before moving on to the next branch.

A performance tree contains three types of nodes: track, part, and section. A track is the primitive element and contains the MIDI data which is transmitted to the instruments connected to the sequencer output. A track is always a leaf node. The other nodes, part and section, do not contain playable data. They are control structures which determine the traversal path. They also hold attribute information which is inherited by their children. The primary distinction between parts and sections is how their children are activated. A part plays its children sequentially; when the first child terminates, its next eldest sibling is begun. A section plays its children concurrently; all children are activated at the same time. Control is returned to the parent after all its children have terminated.

4.3. Structure Definition

A structure has a **name** so it may be referred to in subsequent operations, it has a **class** which determines how it and its children will be performed, and it contains a list of nodes which represent specific instances of other, predefined nodes. These instance nodes are the children, which are invoked by the parent during performance. The information which defines the parent is kept in a **FORMAL** node, as this is a formal definition, that is, the structure is defined in general, no attributes are attached to distinguish one occurrence of this structure from another.

FORMAL node

name	The label which is given to the structure. The name is used to identify the structure for operations directed to it.
class	One of track, part, or section. Indicates childrens' performance mode; sequential for a part, concurrent for a section. A track has no children.
definition	A list of instances which define the node. If the node class is a track, the definition consists of MIDI data.

Operations which are applied to FORMAL nodes are **CREATE**, **DELETE**, **COPY**, and **RENAME**. These operations effect the structure as a whole. The utility of the CREATE and DELETE operations is obvious while the RENAME operation is largely for convenience. The COPY operation is useful for two reasons: (1) a copy may be experimented upon without destroying the original, (2) it may be desirable to have two similar, yet different, structures appear in a work, as with an exposition and recapitulation.

Structure Creation Operations

CREATE(name, class)
Create a FORMAL node of type "class" with label "name". It is a precondition of this operation that an object "name" does not currently exist.

DELETE(name)
Symbolic information for an existing object, "name", is removed from the system. All references to instances of "name" in existing objects must be removed. If the object referred to by "name" is a track, the track data is destroyed along with its symbolic information.

COPY(old, new)
Duplicate the structure named "old" and call it "new". Only the top-level of the structure is duplicated; that is, each defining node is not recursively duplicated.

RENAME(oldname, newname)
Preconditions are: "oldname" must exist, "newname" must not exist. This operation replaces the label, "oldname", with the label, "newname".

The structural definition of a FORMAL node is a list of **INSTANCE** nodes. An INSTANCE node contains a reference to a FORMAL node and a list of attributes. A structure may have several occurrences of another structure in its definition, but each instance may be different. The differences are revealed in the list of attributes attached to each specific occurrence of a node in a definition list.

INSTANCE node

reference	An INSTANCE node is a list of attributes which modify an occurrence of a structure. The reference field indicates the specific structure of which this is an instance. It refers to a FORMAL node.
channel	Channel voice messages for this INSTANCE node are transmitted on the specified MIDI channel. The default is no modification of the channel information.
mute	This node is played silently. Only timing information is sent. The default is not muted.
repeat	Perform node this many times in sequence. The default is one performance of the node.
transpose	Adjust all note messages by the transposition factor. The default transposition constant is zero, no transposition.

Operations are needed which act upon INSTANCE nodes. The operations which effect part of a structure by modifying its list of instances are **INSERT**, **REMOVE**, and **SET_ATTRIBUTES**. These routines are used when a structure is being defined. That structure, referred to below as "editobj", is the object currently being edited.

Structure Definition Operations

INSERT(name, position, editobj)
An INSTANCE of the object, "name", is created and placed at "position" in the list of instances defining the current edit object, "editobj".

REMOVE(position, editobj)
The INSTANCE node at "position" is removed from the current edit object, "editobj".

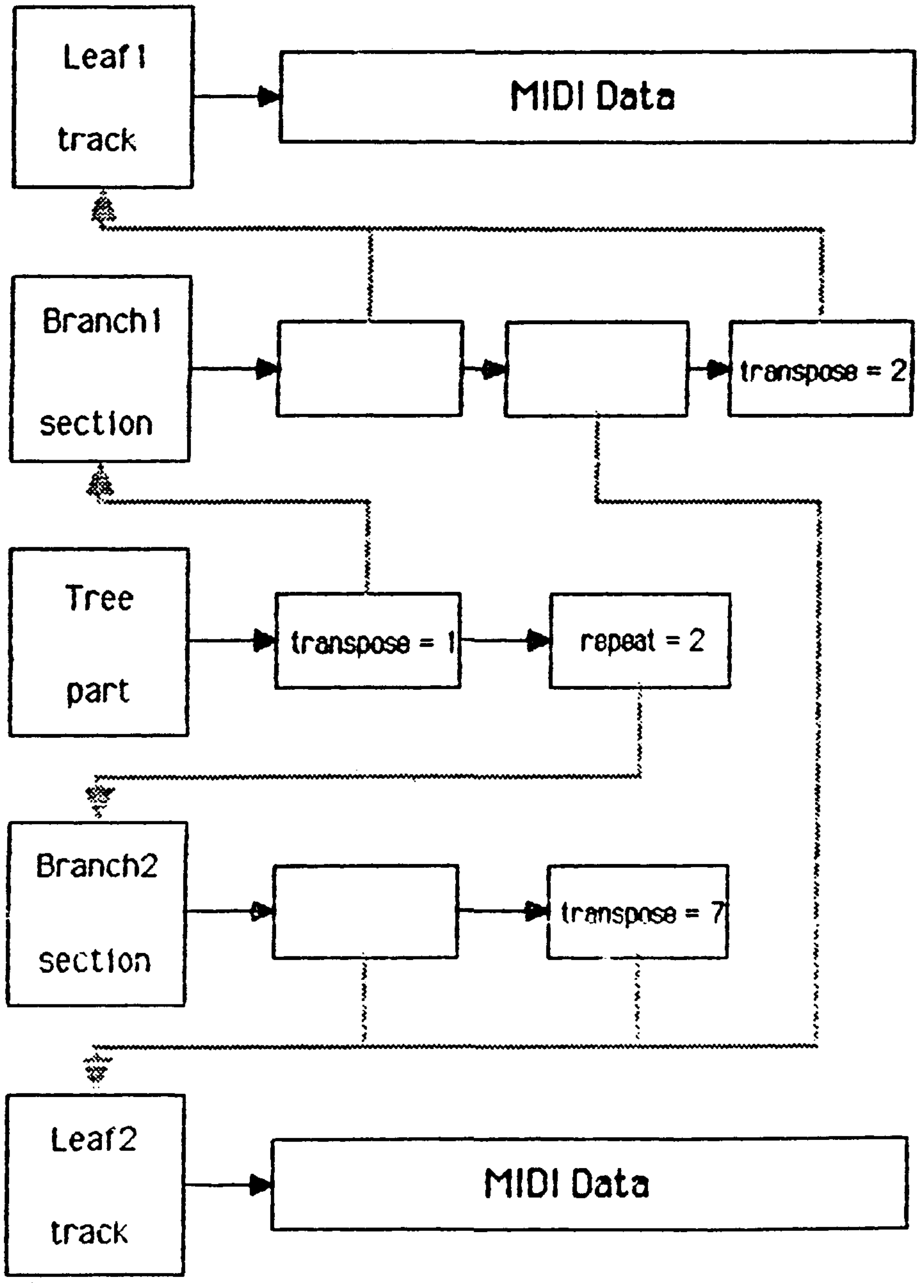
SET_ATTRIBUTES(attributes, position, editobj)
The list of attributes is installed in the INSTANCE node located at "position" within the current edit object, "editobj".

A graphic example of a music structure definition which illustrates the nodes that are created and how they are linked together is given below (figure 4-3). The example shows how listing 4-10 is represented using the data structures described in this chapter.

In the example, a FORMAL node is distinguished by the name field. It also contains a field indicating its class. An INSTANCE node has a reference, depicted as an arrow, to the FORMAL node of which it is an instance. Attributes are shown in INSTANCE nodes only if they are not default values.

The top-level of the example, the FORMAL node for <u>Tree</u>, is a part node, whose children are <u>Branch1</u> and <u>Branch2</u>. This instance of <u>Branch1</u> is transposed up a semitone. The repeat attribute indicates that <u>Branch2</u> is performed twice.

Fig. 4-3: Structure Definition



The FORMAL definitions for Branch1 and Branch2 are similarly

Listing 4-10: Structure Definition Example

```
PHRASE
  Leaf1,
  Leaf2:

PROC Branch2 =
  PAR
    Leaf2(               )
    Leaf2( transpose = 7 ):

PROC Branch1 =
  PAR
    Leaf1(               )
    Leaf2(               )
    Leaf1( transpose = 2 ):

PROC Tree =
  SEQ
    Branch1( transpose = 1 )
    Branch2( repeat    = 2 ):
```

structured. The PHRASE structures, Leaf1 and Leaf2, contain MIDI data and so the definition part of their FORMAL nodes contains that data.

4.4. Play Tree Traversal

4.4.1. Play Tree Node Description

The data structure used to represent the structure of the music for real-time performance is a tree. The network of references to other tree nodes contained in the data structure provides uniform and rapid access to information needed during performance. Each node contains three fields with which to reference other tree nodes; (1) all nodes may have children except those representing tracks. Each tree node has a field used to reference its eldest child. The eldest child is the child which is performed first in the case of a part node. (2) each non-leaf node in the tree may have any number of children. This precludes the use of a fixed number of child fields. Therefore, each child has a reference to its sibling. (3) the tree must be traversed up as well as down; a node must have access to its parent for obtaining information (parent's node type) and to perform operations (increment/decrement parent's semaphore). For these reasons, each node contains a reference to its parent.

Two references to the structure itself are provided. The reference to the **FORMAL** definition yields the node type; track, part, or section. The **INSTANCE** node provides attribute information. After a node has been performed its designated number of repeats, the repeat count is obtained from the INSTANCE node and used to refresh the count in the tree node. This solves the problem of nested repeats; an ancestor node with a repeat count will activate all its descendents that number of times. It is necessary for the descendents to maintain their respective repeat counts during each iteration.

The **semaphore** field is provided for the synchronization required by section nodes. All children of a section begin execution concurrently, but they may not all terminate at the same time. To preserve the music structure, a node's children must complete before the node may be reactivated. The semaphore count is incremented for every child as it is activated. When the child terminates, the semaphore count is decremented. The last child to terminate decrements the semaphore count to zero, allowing the traversal process to continue.

The **attributes** contained in a tree node are the same as those described above. However, the values appearing in the tree node may not match those found in the corresponding INSTANCE node. The case of the repeat count has already been described. The transpose and mute attributes may differ from the INSTANCE node because they inherit attribute values from their ancestors. Reasonable behavior dictates that if a node is muted, all its children must also be muted. Thus, the mute attribute is inherited. A muted node causes all its children to also be muted. Transposition is also inherited. Transposition constants are additive. The transposition constant of a node is the sum of the transposition constants of its ancestors.

TREE node

FORMAL
The reference to the FORMAL node provides access to needed information. Currently, the node class is accessed from the FORMAL node.

INSTANCE
The reference to the INSTANCE node provides access to structure attributes.

parent
Link to parent node.

child
Link to eldest child node.

sibling
Link to sibling node.

direction
Indicates the direction of traversal. Legal values are UP and DOWN. At the beginning of play, all nodes are set to DOWN.

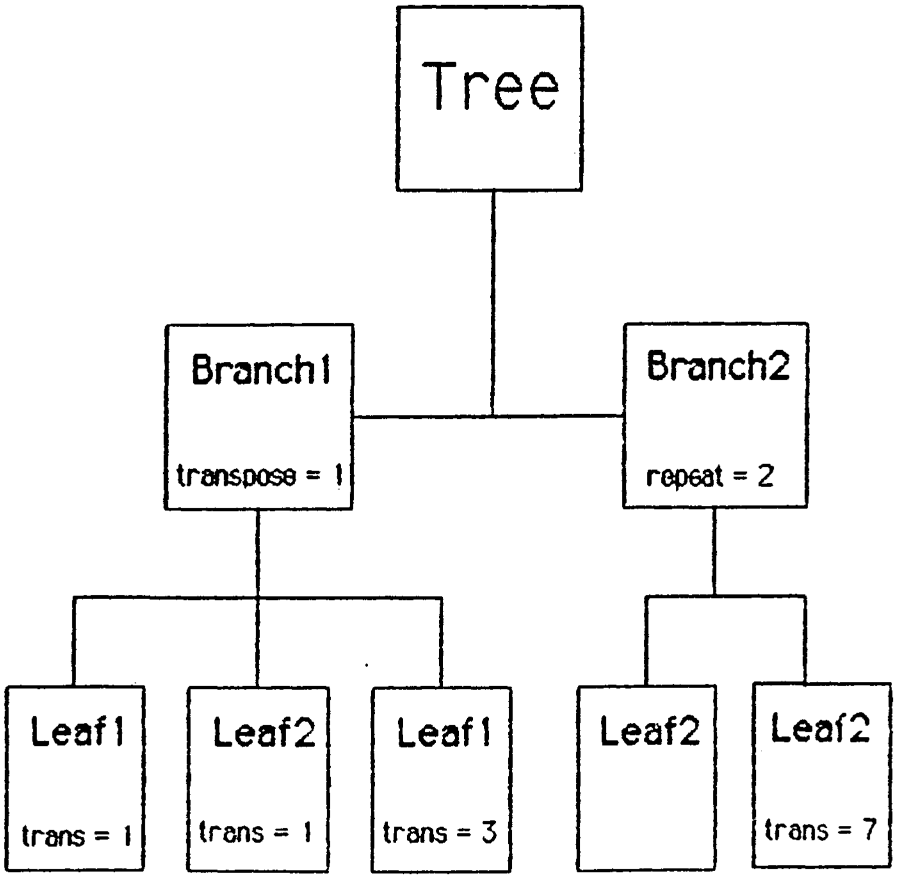
semaphore
A parent node must wait for its children to terminate before it may terminate. The semaphore keeps track of children which are still in play. The semaphore is incremented for each child when the child begins play and is decremented by each child when the child terminates.

attributes
The attributes are the same as those described for INSTANCE nodes. The transpose and mute attributes are inherited from ancestor nodes, along with any attribute modification contained in the current node.

The repeats attribute is copied from the INSTANCE node at the time the node begins play. It is decremented each time the node is repeated.

The example below (figure 4-4) is an illustration of the tree created for the previous listing. Note that the type of connection between nodes indicates whether they are performed sequentially or concurrently.

Fig. 4-4: Play Tree



The substructure of Branch1, illustrates inheritance of attributes. This instance of Branch1 is transposed up a semitone, therefore all its children are also transposed up a semitone. Its eldest child, the second occurrence of Leaf1, also contains a transposition factor in addition to its inherited transposition. Its total transposition is the sum of its inherited transposition and its own transposition.

4.4.2. Informal Algorithm Description

During real-time play, a table is maintained which has the same number of entries as the composition has tracks which will play concurrently. Each entry contains a reference to an active node in the play tree. A low-level play algorithm, which is responsible for transmitting the MIDI data, requests the data by track number. The track number is mapped to the corresponding table entry which points to the tree node currently active for that track. If the active node is a track, a MIDI event is obtained from the current track position. The MIDI event is modified by any attributes which affect it. It is then transmitted to the sequencer output. If the active node is not a track or the end of the track has been reached, more data must be found. The node stored in the table is used as a place marker with which to locate the material which must be performed next.

The **Traversal Algorithm** is called when more data is needed for a particular track. Input to the algorithm is a reference to a newly-terminated node which is obtained from the table of active nodes. The function of the algorithm is to locate the next node to be performed and to install it in the table of active nodes. A next node may not be found for either of two reasons, (1) the structure may have finished playing in which case a flag is set which signals the end of play, (2) there may be no more data for that particular track number in which case the corresponding table entry is marked empty.

There are two primary conditions which determine where the next active node will be found; (1) the direction of travel, UP or DOWN, (2) whether or not the node has children. The simple case is a node whose direction is DOWN and has at least one child. In this case, for a part node, its eldest child is activated; for a section node, all its children are activated.

A leaf node or a node whose direction is marked UP, indicates the same condition; an entire branch has been performed. The traversal now moves upward to find a new branch to traverse or, in the case of a node which repeats, to perform the branch over again. If a node repeats, the repeat count is decremented and that node is installed in the active table to be performed again. If there are no repeats and the node is the root of the tree, then the entire structure has been performed and play is terminated.

If there are no more repeats and the node is not the root of the tree, then the algorithm continues its search for a new branch to perform. The parent of the deactivated node determines the direction of the algorithm.

If the parent is a section, all its children were activated at the same time. Each activation incremented a semaphore marker. Therefore, when the child of a section node terminates, the semaphore marker of the parent is decremented. When it reaches zero, all children have completed and the parent becomes the next active node.

Part nodes have their children activated in sequence. Therefore, if the parent is a part, the next sibling must be activated. If there are no more siblings to perform, the parent is activated.

Preliminary Traversal Operations

ACTIVATE(t)
Begin the performance of node "t" by placing it in the table of active nodes. Set the direction of "t" to DOWN.

END_OF_LIST(t)
Returns TRUE if the node "t" is the last sibling in a

list of nodes.

ROOT(t)
Returns TRUE if the node "t" is the root node of the play tree.

In the **Traversal Algorithm**, "t" is the node from the active node table. It has just completed its activation, and is used to locate the next node to activate.

Traversal Algorithm

```
1.  if t.direction is UP or t is a leaf node then
    1.1  if t.repeats > 1 then
        1.1.1  t.repeats := t.repeats - 1
        1.1.2  ACTIVATE(t)
    1.2  else if ROOT(t) then
        1.2.1  done
    1.3  else
        1.3.1  t.direction := UP
        1.3.2  if t.parent is a part then
            1.3.2.1  if END_OF_LIST(t) then
                1.3.2.1.1  ACTIVATE(t.parent)
                1.3.2.1.2  t.parent.direction := UP
            1.3.2.2  else
                1.3.2.2.1  ACTIVATE(t.sibling)
        1.3.3  else
            1.3.3.1  t.parent.semaphore
                     := t.parent.semaphore - 1
            1.3.3.2  if t.parent.semaphore = 0 then
                1.3.3.2.1  ACTIVATE(t.parent)
                1.3.3.2.2  t.parent.direction := UP
2. else
    2.1  if t is a part then
        2.1.1  ACTIVATE(t.child)
    2.2  else
        2.2.1  if t is a section then
            2.2.1.1  for each child of t
                2.2.1.1.1  ACTIVATE(child)
```

4.4.3. Algorithm Example

To illustrate the traversal algorithm, a simple example is given in listing 4-11. In the following discussion, a number in parenthesis refers to a step as listed in the Traversal Algorithm.

In addition to building the play tree, another of the preliminary operations performed before play actually begins is placing the root node in the table of active nodes; so when the traversal algorithm is first called, it is to find the node which succeeds the root. In this example, <u>Traverse</u> is the root node. It is a part (2.1), and so, <u>Section</u>,

Listing 4-11: Traversal Example

```
PHRASE
  Track1,
  Track2,
  Track3:

PROC Section =
  PAR
    Track1()
    Track2():

PROC Traverse =
  SEQ
    Section(            )
    Track3( repeat = 2 ):
```

being the eldest child, is activated (2.1.1).

Section contains no MIDI data, so a new request is generated immediately. Since Section is a section node (2.2.1), its two children, Track1 and Track2, are simultaneously activated (2.2.1.1). As long as Track1 and Track2 contain data, the traversal algorithm is not called.

Assuming that Track1 completes first, the semaphore of its parent, Section, is decremented (1.3.3.1) and the traversal algorithm does not install a new node. When Track2 terminates, Section's semaphore is zero, so Section is activated with its direction set to UP (1.3.3.2.2).

The next invocation of the traversal algorithm receives Section as the input node. Its direction is UP and its parent is a part (1.3.2). It is not the last sibling, so

Track3 is activated (1.3.2.2.1). When Track3 has played all its data, its repeat count is decremented (1.1.1) and it is reactivated.

At the next node request, Track3 is found to be the last sibling (1.3.2.1) and its parent, Traverse, is activated. Since Traverse is the root node, the next node request signals the end of play (1.2.1).

Chapter Five

Evaluation and Future Enhancements

5.1. Overview

The author has implemented a sequencer which incorporates functions for creating and manipulating high-level music structures. A defined structure may then be performed in real-time. The program requires an IBM-PC or compatible computer equipped with a Roland MPU-401 MIDI Processing Unit or compatible interface.

The sequencer was built as an aid in developing this thesis. It has made several contributions. Its design was instrumental in developing the data structures and algorithm presented in chapter four. Its correct operation verifies the suitability of the material of chapter four. Finally, it provides a means to examine this paper's hypothesis which is stated below.

> An effective environment for the development of musical compositions must allow the creation and manipulation of high-level musical constructs.

Any legitimate evaluation of the program must be based upon significant use by more than one person. As the program has not been adequately tested for "an effective environment", its effectiveness in implementing the two musical examples presented throughout this text, Prelude and Canon, are discussed. In addition, future enhancements are discussed which include appropriate interfaces and performance directives.

5.2. Two Examples

Prelude and Canon, the two examples that have been developed throughout this paper, can be implemented on this project's sequencer directly from the listings given in chapter two. However, the implementation of each may be simplified. The KingsTheme and the Violin1 part of the Canon, each contain a phrase repeated three times. Instead of explicitly listing each repetition, the repeat attribute can be attached to the initial phrase invocation. This change simplifies and clarifies the structure. The result is given in listing 5-12. The four beats of rest, which begins Violin2, may be created in two different ways; (1) by going into record mode for four beats without receiving any MIDI data from an external MIDI instrument, (2) by using the built-in **rest** function. This function is useful whenever some duration of silence is needed. It creates a track which contains silence information. This track is subject to the same operations as any other track, although certain operations, such as transposition, are meaningless.

Listing 5-12: Canon, with repeats

```
PHRASE
  Theme,     --written by King Frederick the Great
  Violin,    --canonical voice
  FirstNote: --first note of the Theme, needed for ending

PROC KingsTheme =
  SEQ
    Theme( repeat = 3 )
    FirstNote():

PROC Violin1 =
  SEQ
    Violin( repeat = 3 ):

PROC Violin2 =
  SEQ
    rest(4)        --four beats of silence
    Violin1():

PROC Canon =
  PAR
    KingsTheme()
    Violin1()
    Violin2():
```

The implementation of attributes allows the low-level structures of the Prelude to be simplified. The phrases, x2 and x3, are merely transpositions of x. Utilizing the transposition attribute reduces the amount of raw material that must be created for this composition. The high-level structures shown in chapter two are modified by adding the repeat attribute. The resulting structures are given in listing 5-13.

This points out a strength of the hierarchical approach to composition using sequencers: major structural modifications are easy to make. A change made at any level is reflected throughout the composition. A useful application is in designing the structure of a composition, but only doing a rough sketch of the actual track material. When track data is perfected, it can be easily substituted for the original sketch. A function, **assign**, is included in the author's sequencer to facilitate this type of operation. It replaces the track data of one track with the data of another without changing the name of the original track. In this way, when an improved version of a track is created, one simple operation will globally substitute its data for the original throughout a composition.

5.3. Interface

Most sequencers are primarily multi-track tape recorders modeled in software. This is most evident in those with graphics-oriented interfaces; the screen is often a replica of a tape recorder, complete with buttons to

Listing 5-13: Prelude, with transposition

```
PHRASE
  x,    --left hand two note motive
  y:    --right hand two note motive

PROC a' =
  SEQ
    x()
    y():

PROC b' =
  SEQ
    x( transpose = 2 )
    y(               ):

PROC c' =
  SEQ
    x( transpose = 3 )
    y(               ):

PROC a =
  SEQ
    a'( repeat = 4 ):

PROC b =
  SEQ
    b'( repeat = 4 ):

PROC c =
  SEQ
    c'( repeat = 4 ):

PROC B =
  SEQ
    a()
    b()
    c()
    b()
    a():

PROC Prelude =
  SEQ
    B()
```

activate play, rewind, and record functions. An easily recognizable model provides a comfortable environment and reduces the time needed to learn to operate the sequencer. The limitations of these models as a compositional tool have been discussed.

This thesis arose from the need to resolve the conflict between creating a hierarchical music structure and the single dimensional concepts, which are so prevalent, for realizing these structures. A question must be asked: In the face of the apparent superior flexibility and organizational power of a hierarchically-based sequencer, why have these ideas not been implemented in commercial sequencer products?

During the course of this investigation, designing the interface in particular, a partial answer was uncovered. Representing multi-dimensional information in the confines of a computer monitor is not an easy task. The act of composition involves many conceptual jumps between foreground, background and intermediate levels of structure [Laske 1978]. The composer is able to make these jumps instantaneously, the computer, however, is not. It is not likely that all necessary information can be available on the computer screen at the same time. It is probable that the lack of a sufficiently intuitive interface between the composer and the computer is largely responsible for the dearth of sequencers which go beyond the tape recorder analogy.

The sequencer developed for this thesis does not adequately solve the above interface problems. Two alternative approaches to the project's interface are discussed below.

5.3.1. Programming Language Interface

The examples of music structures presented in this paper have been based on the Occam language. It was chosen because it has a syntax which represents concurrent, as well as sequential, processes. Also, as with other block structured languages, the program structure reveals a hierarchical organization such as that found in music structures. While Occam is probably not suited as a music structure language, it is desirable to have such a language. An integrated system where program text was entered with a text editor, compiled, and performed by a sequencer might be a suitable music development environment. All necessary information would be available in the program text. The problem is, of course, developing a music-definition language.

5.3.2. Graphics Interface

A more intuitive approach to representing a structure is to depict it graphically on the screen. Iconic interfaces are quite common on microcomputers. With the addition of a mouse as an input device, the command language can be removed from structure manipulations and replaced with a simpler point, drag and click input language. Since the music structure being represented is a tree, the structure creation process may involve constructing a tree, similar to the tree constructed in the previous chapter, on the computer screen. Each node of the tree is one of the structure types, phrase, part, or section. Vital information, such as structure name and associated attributes, may be displayed within the node.

A windowing environment may provide a convenient program interface. Each window contains the definition of a substructure. The definitions may be entered as a programming language similar to the listings presented in this paper or the tree may be constructed by dragging structure icons into the window. Multiple windows allow any portion of the structure to be visible or hidden as required by the composer.

A carefully designed input language or graphics interface could significantly enhance the control the composer has over shaping the musical materials.

5.4. Performance Directives

Performance directives were described in chapter three. The most rudimentary performance directive has been implemented in the sequencer built for this study; a master tempo control. A flexible and powerful set of performance directives would greatly enhance the usefulness of a sequencer.

Performance directives may contain MIDI messages that are not normally found in a recorded track, such as system exclusive information. They may contain MIDI messages, such as program change messages, that are more conveniently included after the track has been recorded. Conductor information, such as tempo control, are also included in performance directives. A list of useful performance directives is given below.

Performance Directives

control change
The control change command sends a control setting for a specified controller. At the end of a performance, this command may be sent to set controllers, such as modulation and pitch-bend, to their default settings in preparation for the next performance.

metronome
The audible metronome may be activated or deactivated

at a particular point in the music.

overdub
The overdub command turns recording on or off at a specified point during play. This is similar to the punch-in/punch-out command, but does not overwrite any existing track information.

program change
The program change command is useful for quickly changing voice settings on a MIDI instrument. Many signal processors also respond to program change messages.

system exclusive message
A system exclusive message may be sent. This may be used to change a parameter setting of an instrument or, before play begins, to send voice definition information to the instruments in the MIDI system.

tempo
The tempo command can set either an absolute or a relative tempo. An absolute tempo is specified in beats per minute. A relative tempo change sets a new tempo to some ratio of the current tempo.

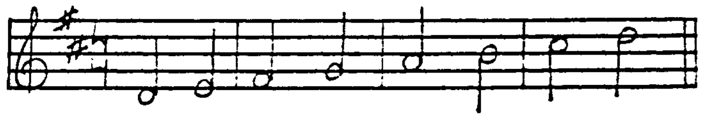
The requirements for the behavior of performance directives make them similar to attributes in some ways, but with significant differences. It is desirable to attach a performance directive, or a group of performance directives, to a structure just like an attribute. The performance directives are activated when the structure begins its execution. In this way, a program change message may select a different voice on a synthesizer at the beginning of a new section.

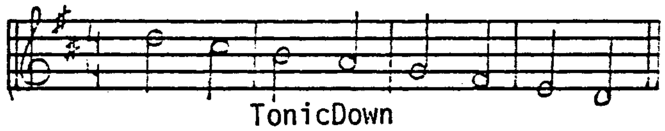
The attributes which have previously been defined all act upon existing data. Transposition modifies the pitch of recorded MIDI note events, muting substitutes silence information for note data, etc. Performance directives contain their own data to be output. This may be a MIDI message or it may be control information, such as a tempo change. Thus, their internal representation and their execution must differ from that of attributes.

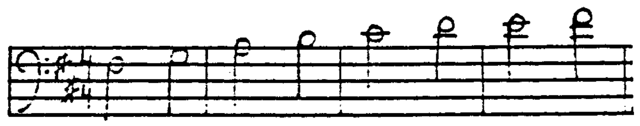
MIDI systems may be configured in many ways. The configuration of a large system may require that a great deal of setup data be transmitted. It is convenient to have the setup information for each instrument in the system grouped together as when configuring the system for a particular composition or series of compositions. For this application, performance directives may defined as independent entities. They may then be "performed" as a music structure is performed, but setup information rather than note data is transmitted.

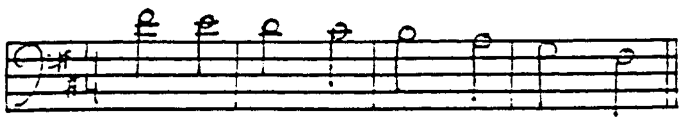
Appendix

Scale

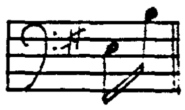

TonicUp


TonicDown

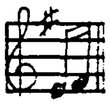

SixthUp


SixthDown

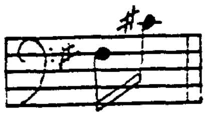
Prelude



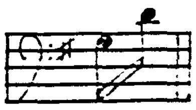
x



y

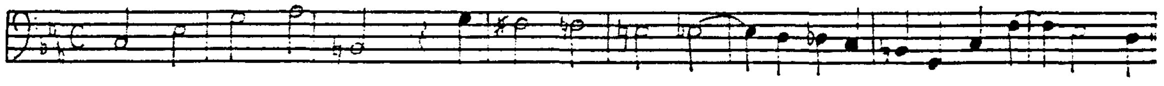


x2

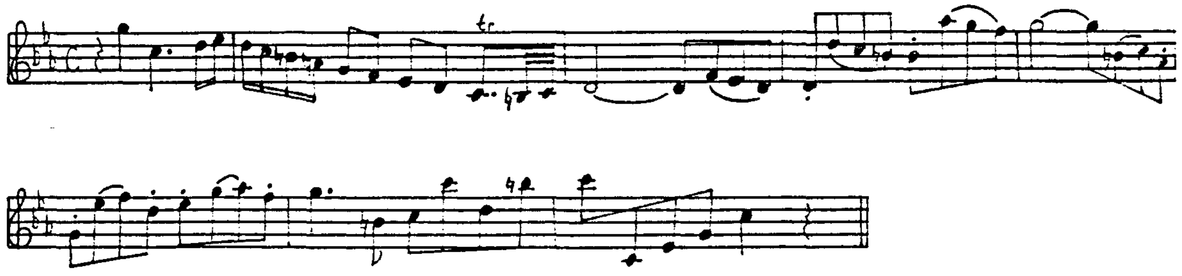


x3

Canon



Theme



Violin

Bibliography

Aikin, J. 1987. Sequencer Basics. Keyboard 13, 6. (June).

Apel, W. 1972. Harvard Dictionary of Music. The Belknap Press of Harvard University Press. Cambridge, Massachusetts.

Byrd, D. 1974. A system for Music printing by computer. Comput. Hum. 8. (May).

Cooper, J. 1986. An Insider's View of MIDI. Keyboard 12, 1 (Jan.).

Cooper, J. 1987. More On MIDI Continuous Controllers. Keyboard 13, 7. (July).

Cooper, J. 1987. MIDI Time Code. Keyboard 13, 8. (Aug.).

Garvin, M. 1987. Designing a Music Recorder. Dr. Dobb's Journal of Software Tools 12, 5. (May).

Gourlay, J. 1986. A Language For Music Printing. Communications of the ACM 29, 5. (May).

Greenwald, T. 1986. Texture 2.0. Keyboard 12, 10. (Oct.).

Laske, O. 1978. Considering Human Memory in Designing User Interfaces for Computer Music. Computer Music Journal 2, 4.

Many, C. 1987. Texture Version 2.5. Music Technology (May).

Maxwell, J. and Ornstein, S. 1984. Mockingbird: A Composer's Amanuensis. Byte 9. (Jan.).

MIDI Manufacturers Association, Inc. 1985. MIDI 1.0 Detailed

Specification. International MIDI Association, North Hollywood, CA.

Milano, D. 1987. Tips from the Pros. Keyboard 13, 6 (June).

Pountain, D. 1984. Microprocessor Design. Byte 9, 8 (August).

Powell, R. 1986. Texture 2.0 User Manual. Magnetic Music.

Smith, L. 1973. Editing and Printing Music by Computer. Journal of Music Theory 17, 2.