3 Music V Manual

1. Introduction

This chapter contains a detailed description of the operation and structure of the Music V program. It provides reference material for users of Music V and source material for those who desire intimate knowledge of a sound-generating program in order to write their own.

Music V is the direct descendant of Music IV, a program that was widely used for five years and has been described in the literature.[1] Music V had to be rewritten to change from a second to a third generation computer (the IBM 7094 to the GE 645). However, in the process certain improvements were made, especially changes that made the program more easily adapted to other computers. It may be helpful to list these changes for the benefit of users of Music IV.

Principal Differences between Music IV and Music V

1. Music V is written almost entirely in FØRTRAN IV; it is much easier to use on a wide variety of computers. In addition, the FØRTRAN programs have been written so as to be easily modifiable to accommodate the different memory sizes and different word lengths of various computers.

[1] See Annotated References at end of chapter.

Despite being written in FØRTRAN, Music V is potentially as fast as Music IV. This potentiality can be achieved by writing the inner loops for certain computations (the unit generators) in basic machine language. Such programs are, of course, specific to a given computer, but at most only a few hundred instructions are involved.

FØRTRAN unit generators can be intermixed with basic machine-language generators. Initially, the program can be operated entirely with FØRTRAN generators. Gradually, the most frequently used generators can be coded in machine language. Exotic and infrequently used generators may remain in FØRTRAN at little cost. New generators can easily be added in FØRTRAN.

2. Instruments are defined as part of the score rather than in a separate program. (In Music IV the orchestra was assembled by the BE FAP assembly program.) In this way the entire composition—notes and timbres—is specified in a single document, the score. In addition, instruments may be redefined or changed at any point in the score.

3. A given instrument may play any number of voices simultaneously. Only one instrument of a given type need be defined; the composer no longer need worry about losing notes that overlap in time on an instrument. Unit generators are also multiply used; only one copy of each type of generator is in the memory; memory is thus conserved.

4. A free-field format for score cards is used. Successive fields are separated by one or more blanks or by commas. Mnemonics are used to denote operation codes and unit-generator types. This form of score is easier both to write and to read than the Music IV fixed-field score.

The score is interpreted by a completely separate subroutine READ1 and the output is entirely in numerical form. Therefore, it is possible to change the form of the score simply by replacing the READ1 routine. Moreover, since all subsequent parts of the program are strictly numerical, a maximum of machine independence is achieved in the FØRTRAN programs.

Overview of Music V

A block diagram of the over-all operation of the programs is shown in Fig. 48. The main programs, the principal subroutines, the flow of control, and the flow of data are indicated. The few basic machine-language programs are especially marked.

Pass I causes the score to be read by the READ1 subroutine. The score may be thought of as a sequence of data cards prepared by the user, although the actual medium could also be a computer-connected typewriter, a graphic computer, or a data file.

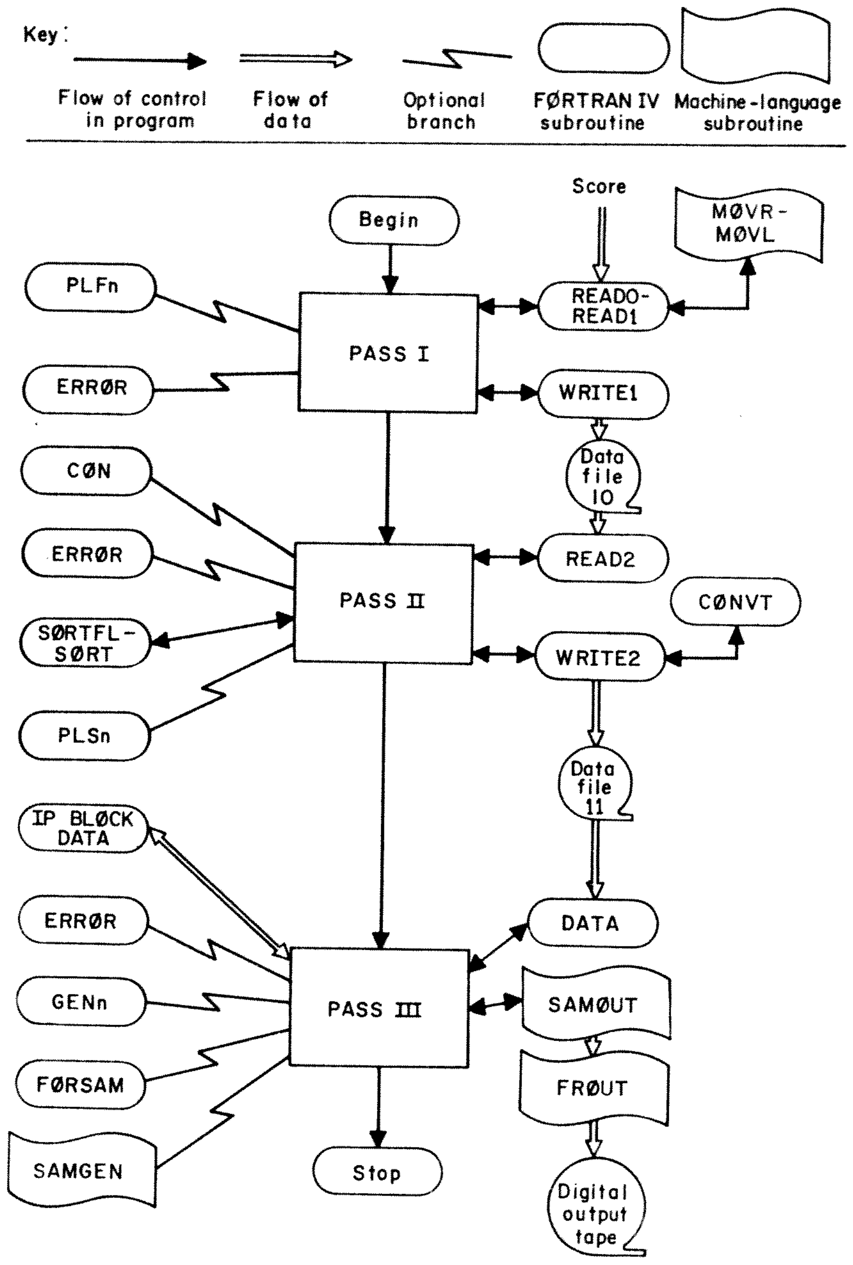


Fig. 48. Block diagram of Music V operation.

Cards are processed by Pass I in the order in which they occur in the score. Data are grouped into data statements which are terminated by a semicolon; a data statement need not correspond to a single card. The first field of the data statement specifies an operation code, and the second field specifies an action time when the operation is to be done. This time is measured from the beginning of each section of the music. The other fields may vary depending on the particular operation code.

The total number of fields may vary; no more than necessary need be used.

The principal operations are to

(1) Cause a note to be played
(2) Define an instrument
(3) Store data in Pass I, II, or III memory
(4) Call a subroutine in Pass I or II
(5) Generate and store a function in Pass III
(6) Terminate a section or a composition.

Pass I calls several subroutines. The function of the READ and ERRØR subroutines are obvious. The PLF subroutines are note-processing and generating routines which the composer has the option to provide if he wishes to make use of this possibility. MØVR and MØVL are two short machine-language routines that move a character to the right and left end, respectively, of the computer word. These are two of the few essential machine-language routines that must be provided.

Data statements are sent to Pass II via a data file recorded on disk or tape. Each statement is still labeled with an action time in the second field. The principal function of Pass II is to sort the data statements into ascending order of action times. (In Pass I, action times need not be ordered; in Pass III a strictly ascending order is required.) The sorting is carried out by two subroutines, SØRTFL and SØRT. These are provided as FØRTRAN IV routines; however, the sorting process can be substantially speeded by writing or obtaining machine language versions. Sorting programs are quite generally available.

After sorting the data statements for time, Pass II (optionally) applies a metronome function to distort the time scale. Subroutine CØN is used to read the metronome markings which are stored in the Pass II memory. Gradual accelerandos and ritardandos are possible, as well as sudden changes in tempo.

User-provided subroutines, called PLS subroutines, may be optionally supplied and applied to the data records after time sorting.

Just before each data statement is sent to Pass III, a CØNVT subroutine operates on all its fields. CØNVT must be supplied by the user; it replaces all the CVT routines in Music IV. For example, it is often given the job of converting frequency notation from some humanly simple scale like 12 tones—1 . . . 12—to the proper input numbers for oscillator frequency control. Inputs for attack and decay generators are conveniently computed here. Frequently, CØNVT adds parameters to the data statement.

The actual acoustic samples are computed in Pass III. The unit generators are encoded in SAMGEN (in basic machine language) and FØRSAM (in FØRTRAN IV). The Pass III program organizes these unit generators into instruments and plays the instruments as specified by the score. In addition, the GEN routines may be called upon to compute functions that are stored in the Pass III memory and are referred to by unit generators (e.g., ØSC).

Data statements which are the input of Pass III have action times written in their second field; these action times are now monotonically ordered; they determine the times at which all processing and generating in Pass III are performed.

Almost all information in Pass III is stored in one large array called I. It contains instrument definitions, parameters of notes currently being played, stored functions (from GEN routines), input–output blocks for unit generators, and certain other data. The size of I can be adjusted to a particular machine by an appropriate dimension statement.

Various other essential parameters—such as the length of a stored function, the number of stored functions, the length and number of input–output blocks, the maximum number of simultaneously sounding voices—will change with different computers and compositions. These parameters have been assembled into the IP array, which is compiled by a BLØCK DATA subprogram. Hence the parameters can be easily changed.

The usual unit generators and GEN functions use fixed-point arithmetic and store their results in the I array. (It would not be difficult to use floating-point routines instead, or to use both.) However, the routines do not produce FØRTRAN integers. Instead, FØRTRAN fixed-point numbers are multiplied by 2^n, which in effect puts their decimal points n places from the right end of the memory words. Values of 2^n for unit generators and for GEN functions are also compiled into the IP array. These values can be changed to accommodate different lengths of memory word.

Output samples are written on a digital output tape by a combination of SAMØUT and FRØUT subroutines. These are inherently machine-language operations, and there is no way to avoid so writing them. However, they can be brief and demand little programming time.

Chapter 3, which presents the Music V manual, is organized in the same manner as the program; it starts with a discussion of Pass I and its subroutines and then proceeds to the other passes. The actual FØRTRAN programs are, of course, the ultimate and best description of Music V; they should be read along with the manual.

2. Description of Pass I

The purpose of Pass I is to read the input data (score) and translate it into a form acceptable to the subsequent passes. The operation is diagrammed in Fig. 49.

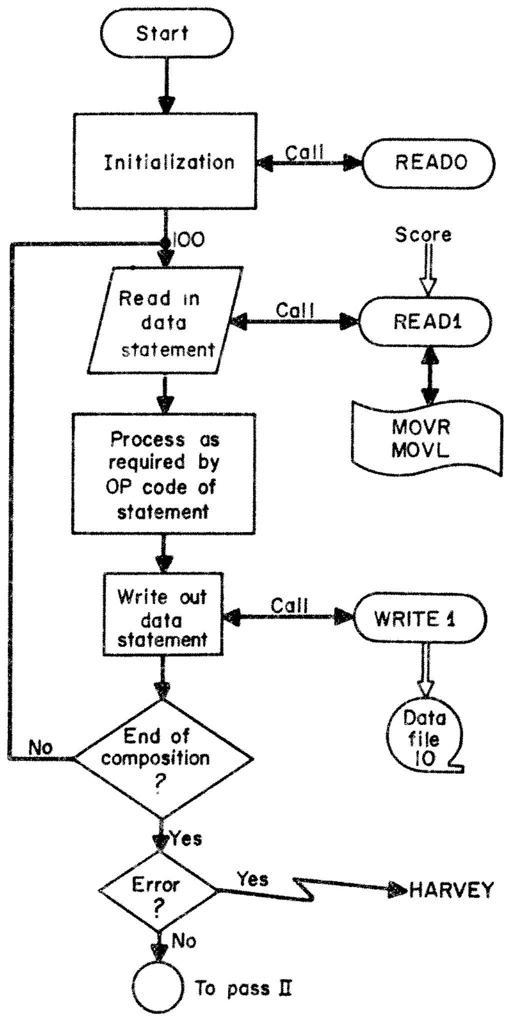


Fig. 49. Pass I.

The interpretive input routine READ1 (and READ0, which is used to read the first record) is written in FØRTRAN IV. It is designed for a computer with a word length of 36 bits. It requires two user-supplied subroutines (MØVL, MØVR) to be written in machine language for purposes of character shifting. Minor modifications to READ0 and

READ1 are necessary for computers of different word length and for different modes of input (see Section 7 for details).

The input data comprise a series of data statements punched in free format in columns 1 through 72 of cards. A data statement need not correspond to a single card.

A data statement begins with an operation code and is terminated by a semicolon. Other fields of information in the statement are separated by blanks (any number) or commas. Null fields, i.e., those denoted by successive commas, are assumed to have the value 0. With the exception of statements used in instrument definitions (see Section 4), the fields of a data statement are referred to as P fields since they load sequentially into the P array located in CØMMØN storage in Pass I.[2] The operation code, written as a three-letter mnemonic (see Section 3) is converted to a numerical equivalent and goes into P(1); the second field, containing an action time that specifies when the operation corresponding to the code is to be performed, goes into P(2). The other fields are interpreted according to the specifications of the various operation codes. If a field other than the OP code is written as an asterisk (*), the value stored in the corresponding position of the P array will be the value previously stored there. This feature can be employed to advantage when parameters remain constant over a sequence of data statements.

The input data are terminated with the data statement having the operation code of TER. Failure to provide this statement will result in an error comment.

The input program makes certain checks on the data statements and when errors are detected the value of IP(2), located in CØMMØN storage, is set to 1. Since this location is initially 0, Pass I can verify at its conclusion whether or not errors have been detected and, if so, the run is terminated without proceeding to Pass II. Termination is accomplished by calling a nonexistent subroutine named HARVEY.

As the data cards are read, they are printed, and any error comments are printed out after the offending statement. Data statements beginning with operation code CØM result only in printing and are not processed further. Such statements may be used to annotate the input data with comments.

In addition to establishing the appropriate values in the P fields, READ1 counts the number of P fields in the data statement and sets IP(1) (in CØMMØN storage) to this count. Pass I is then able to process the data statement as is required by the operation code and to write

[2] CØMMØN storage in Pass I is arranged according to the statement, CØMMØN IP(10), P(100), D(2000)

out the translated statement as N, (P(I), I = 1, N), where N = IP(1), to be read by Pass II.

Pass I contains a data array D(2000) which may be used for general storage and may in particular contain data for the PLF subroutines. SV1 and SIA data statements load the D array. (SV1 0 10 100; would set D(10) = 100.) The following D variables have special significance.

D(4) = Sampling rate

D(8) = Stereo-mono flag

D(8) = 1	for stereo
D(8) = 0	for mono

3. Operation Codes and Corresponding Data Statements

The operation codes are listed in the following table.

Numerical Value	Mnemonic	Purpose
1	NØT	Play note
2	INS	Define instrument
3	GEN	Generate function
4	SV3	Set variable in Pass III
5	SEC	End section
6	TER	Terminate piece
7	SV1	Set variable in Pass I
8	SV2	Set variable in Pass II
9	PLF	Execute subroutine in Pass I
10	PLS	Execute subroutine in Pass II
11	SI3	Set integer in Pass III
12	SIA	Set integer in all passes
13[a]	CØM	Print comment

[a] This code number is used only by READ1. A data statement beginning with CØM is printed but is not processed further.

Remarks

1. Only the first three characters of the operation code mnemonic are scanned; thus a user may write NØTE, INSTRUMENT, GENERATE, SECTIØN, TERMINATE, or CØMMENT in place of the three-letter codes if he prefers.

2. Integer-valued P fields may be written with or without decimal points.

3. Null fields, those denoted by successive commas, are assumed to be 0.

4. Fields specified as * are assumed to have the value previously stored there. This feature provides continuation over a sequence of data statements.

Description of Data Statements

Each statement begins with the mnemonic operation code (at least three letters). The second field must contain the time at which the operation is to be performed. Therefore the descriptions that follow the specifications will begin with the third field. All statements are terminated by a semicolon.

1. NØT—Play note
   P(3) Number of instrument on which note is to be played
   P(4) Duration of note (in beats)
   P(5)... As desired by instrument referred to in P(3)
2. INS—Define instrument
   P(3) Number of instrument being defined
3. GEN—Generate a function
   P(3) Number of generating subroutine (see Section 25)
   P(4) Number of function to be generated
   P(5)... As required by generating subroutine
4. SV3—Set variable(s) in Pass III, starting with variable N
   P(3) Number of first variable to set = N
   P(4) Value of variable N
   P(5) Value of variable N + 1
   P(6) ... (Number of variables to be set is automatically determined by the word count.)
5. SEC—End section and reset time scale to zero
6. TER—Terminate piece at specified time relative to last section
7. SV1—Set variable(s) in Pass I, starting with variable N
   P(3) Number of first variable to set = N
   P(4) Value of variable N
   P(5) Value of variable N + 1
   P(6) ... (Number of variables to be set is determined by the word count.)
8. SV2—Set variable in Pass II
   Fields are as in SV1
9. PLF—Execute subroutine in Pass I
   P(3) Number of subroutine: 1, 2, 3, 4, or 5
   P(4)... As required by subroutine referred to in P(3)

10. PLS—Execute subroutine in Pass II
    Fields are as in PLF
11. SI3—Set integer(s) in Pass III, starting with integer N
    P(3) Number of first integer to be set = N
    P(4) Value of integer N
    P(5) Value of integer N + 1
    P(6) . . . (Number of integers to be set is determined by the word count.)
12. SIA—Set integer(s) in all passes
    P(3) Number of first integer to be set = N
    P(4) Value of integer N
    P(5) Value of integer N + 1
    P(6) . . . (Number of integers to be set is determined by the word count.)

4. Definition of Instruments

An instrument definition begins with the data statement "INS t n;" where t specifies the time at which instrument n is to be defined. Subsequent data statements indicate the unit generators used in the instrument and their associated parameters. The data statement "END;" terminates the definition.

The unit generators that are recognized by name (i.e., three-letter mnemonic) by READ1 follow.

Name	Parameters	Type Number	Purpose
ØUT	I1, Ø ;	1	Monophonic output
ØSC	I1, I2, Ø, F, S ;	2	Oscillator
AD2	I1, I2, Ø ;	3	Two-input adder
RAN	I1, I2, Ø, S, T1, T2 ;	4	Random function generator
ENV	I1, F, Ø, A, SS, D, S ;	5	Envelope generator
STR	I1, I2, Ø ;	6	Stereophonic output
AD3	I1, I2, I3, Ø ;	7	Three-input adder
AD4	I1, I2, I3, I4, Ø ;	8	Four-input adder
MLT	I1, I2, Ø ;	9	Multiplier
FLT	I1, I2, I3, Ø ;	10	Filter
RAH	I1, I2, Ø, S, T ;	11	Random and hold function generator
SET	I1 ;	102	Set new function

See Section 5 for a more complete description of the unit generators.

Data statements that specify unit generators may begin with the three-letter mnemonic name or with the type number. READ1 recognizes the 12 types listed in the table above by name,[3] and makes a check on the proper number of parameters. If, for example, four or six parameters are listed for ØSC, which requires five parameters, an error condition will result, causing the job to terminate at the conclusion of Pass I after all input cards have been scanned. Since unit generators may be labeled by type number as well as name, it is possible to add units to the subroutines FØRSAM (coded in FØRTRAN IV) or SAMGEN (coded in basic machine language) used in Pass III without the need for modifying READ1. Data statements referring to these new units by type number will be accepted by READ1, but no check will be made for proper number of parameters.

The notation for these parameters used on the data statement is as follows:

Pn	refers to nth P field on note card
Vn	refers to nth location in variable storage of Pass III
Fn	refers to nth stored function
Bn	refers to nth I-Ø block used by units

For example, instrument No. 3 would be defined at t = 10 by the following data statements:

```
INS 10 3 ;
ØSC P5 P6 B2 F1 P30 ;
AD2 P7 V1 B3 ;
ØSC B2 B3 B2 F2 P29 ;
ØUT B2 B1;
END ;
```

READ1 translates each mnemonic data statement into an all-numerical data statement as follows:

(1) In all data statements, P1 contains 2, the numerical equivalent of INS, and P2 contains the action time (10 in the example).
(2) P3 contains the instrument number (3) in the first data statement.
(3) In the second through the last data statements, P3–Pn contains the numerical equivalent of the mnemonic data statement fields P1 . . . P_{last}, respectively. The name equivalents for the unit generators are their type numbers listed above. The equivalents of the P's, V's, etc., are as follows:

[3] The "named" generators change frequently. The table describes the state of affairs in April 1968 at Bell Laboratories.

$$\begin{array}{ll} Pm \rightarrow m & 1 \leq m \leq 100 \\ Vn \rightarrow 100 + n & \\ Fp \rightarrow -(100 + p) & \\ Bq \rightarrow -q & 1 \leq q \leq 100 \end{array}$$

The equivalents are unique because only 100 P's and 100 B's are allowed. P's are represented by positive numbers from 1 to 100, V's by positive numbers greater than 100, B's by negative numbers from −1 to −100, and F's by negative numbers from −101 to $-\infty$.

(4) The last mnemonic data statement, END, has only two fields, P1 = 2 and P2 = action time. It is recognized in Pass III by its word count of 2; this terminates the instrument definition.

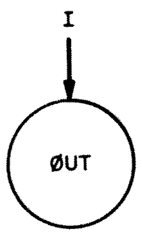
The example is translated into the following numerical data statements:

```
2 10 3 ;
2 10 2 5 6 −2 −101 30 ;
2 10 3 7 101 −3 ;
2 10 2 −2 −3 −2 −102 29 ;
2 10 1 −2 −1 ;
2 10 ;
```

All passes of the program operate exclusively on the numerical statements; all mnemonics are translated by READ1.

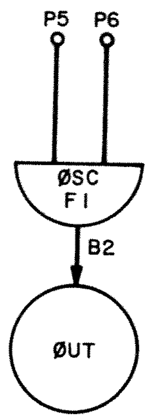
5. Unit Generators

ØUT: Output Unit (Numerical equivalent = 1)
Diagram:



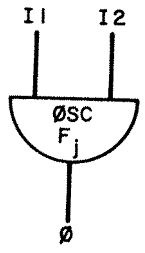
Data statement: ØUT, I, Ø ;
Operation: This unit generator adds the specified input into the specified output block thus combining it with any other instrument that concurrently uses the output block. $Ø_i = Ø_i + I_i$ where i denotes the ith sample.

Example: One of the simplest instruments is defined as
INS, 0, 1 ;
ØSC, P5, P6, B2, F1, P20 ;
ØUT, B2, B1 ;
END ;



B1 is often used as the output block. The location of the output block must be compiled into IP(10) (see Section 17).

ØSC: Oscillator (Numerical equivalent = 2)
Diagram:



Data statement: ØSC, I1, I2, Ø, Fj, S ;
Operation: The oscillator generates functions and oscillations according to

$$Ø_i = I1_i \cdot F_j([S_i] \bmod \text{function length in samples})$$

and

$$S_{i+1} = S_i + I2_i$$
$$S_0 = \text{initial value of sum}$$

where $Ø_i$ is output, $I1_i$ is amplitude, F_j is a (stored) function, S_i is the sum, $I2_i$ (increment) determines the frequency of oscillation, and i indexes the samples.

The frequency of the oscillation is

$$\text{Frequency} = \frac{\text{sampling rate} \times I2}{\text{function length in samples}}$$

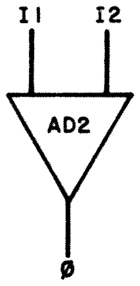
The length of the function in samples is equal to IP(6) − 1. n (which = IP(6)) samples of each function are stored. The first and nth samples represent the same point on the function and must have the same value. Hence the function is periodic with period n − 1 sample times. One note parameter P_n must be reserved for the sum. The value of this parameter on the data statement determines the initial value of the sum S_0. Usually n is selected to be one of the last locations in note parameter storage; if P_n is not written on the note card (cf. Section 4) P_n is automatically set to zero at the beginning of each note.

Example: The example for the output box is also appropriate for the oscillator. F1 determines the wave shape. P5 is the amplitude. P6 determines the frequency. Specifically

$$\text{Frequency} = \frac{\text{P6} \times \text{sampling rate}}{\text{function length in samples}}$$

See Chapter 2, section on ØSC Generator, and Chapter 3, Section 6, for more details about ØSC.

AD2, AD3, AD4: Two-, Three-, and Four-Input Adders (Numerical equivalent = 3; AD3 = 7; AD4 = 8)
Diagram:



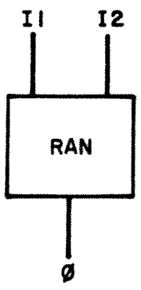
Data statement: AD2, I1, I2, Ø;
Operation: Output is generated according to

$$Ø_i = I1_i + I2_i$$

The other adding units (AD3 and AD4) work in a manner analogous to AD2.

Example: None.

RAN: Random Function Generator (Numerical equivalent = 4)
Diagram:



Data statement: RAN, I1, I2, Ø, S, T1, T2;
Operation: Output is generated according to

$$Ø_i = I1_i * R_i(I2_i)$$

RAN generates a low-pass random function whose peak amplitude is $I1_i$ and whose cutoff frequency is controlled by $I2_i$ and is approximately

$$B \approx \frac{\text{sampling rate}}{2} \cdot \frac{I2_i}{512}$$

More specifically, R_i is a function, varying from -1 to $+1$, obtained by sampling the line segments that connect independent random numbers, N_i. There are 512/I2 samples between each pair of independent random numbers (see Fig. 50). The N_i's are uniformly distributed from -1 to $+1$.

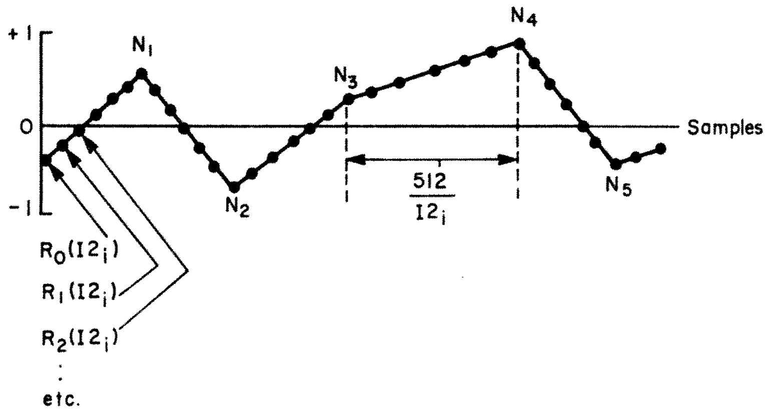
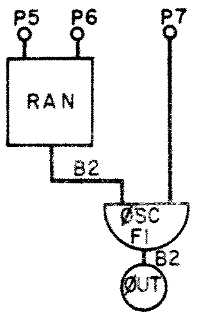


Fig. 50. Random function.

S, T1, and T2 are temporary storage locations which are normally kept in note-parameter locations. S holds a sum equivalent to the ØSC sum. T1 holds N_{i-1} and T2 holds $N_i - N_{i-1}$ where N_{i-1} and N_i are the last two independent random numbers.

Example: A typical instrument to produce a band-pass noise:

```
INS, 0, 1;
RAN, P5, P6, B2, P30, P28, P27;
ØSC, B2, P7, B2, F1, P29;
ØUT, B2, B1;
END;
```



Function F1 is assumed to be a sine wave. By means of the modulation inherent in the multiplication of the left oscillator input, B2 will contain samples of a band-pass noise whose center frequency is

$$\frac{\text{P7} \times \text{sampling rate}}{\text{function length}}$$

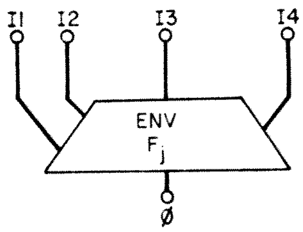
and whose bandwidth is

$$\frac{\text{P6} \times \text{sampling rate}}{512}$$

The peak amplitude is P5.

ENV: Envelope Generator (Numerical equivalent = 5)

Diagram:



Data statement: ENV, I1, F_j, Ø, I2, I3, I4, S ;

Operation: This unit scans a function F_j at a variable rate to produce an attack, steady-state, and decay amplitude envelope on a note.

$$Ø_i = I1_i * F_j \quad \text{(scanned according to I2, I3, and I4)}$$

The first quarter of F_j gives the attack shape, the second quarter of F_j gives the steady state, the third quarter of F_j gives the decay shape, the fourth quarter is unused and should be zero.

Specifically, the sections of F_j and the scanning rates are shown in Fig. 51.

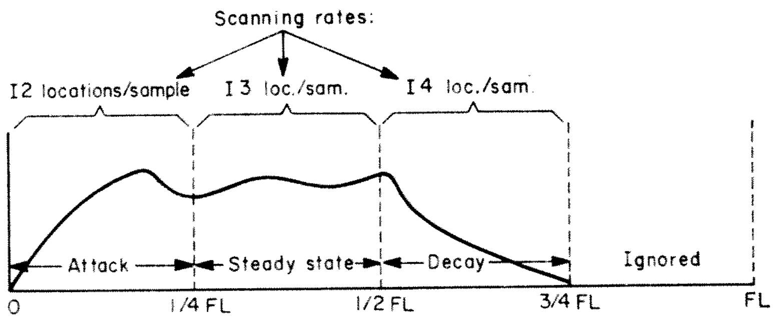


Fig. 51. Envelope function. FL = function length in samples.

In a typical use

$$I2 = \frac{\text{function length in samples}}{4\cdot\text{attack time}\cdot\text{sampling rate}}$$

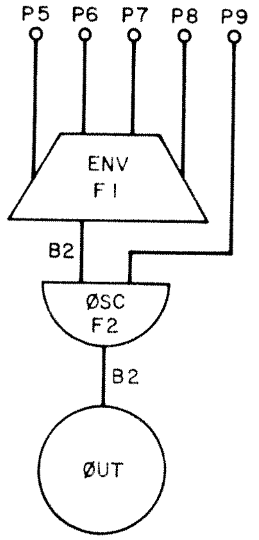
$$I3 = \frac{\text{function length in samples}}{4\cdot\text{steady-state time}\cdot\text{sampling rate}}$$

$$I4 = \frac{\text{function length in samples}}{4\cdot\text{decay time}\cdot\text{sampling rate}}$$

S is a temporary storage location (note parameter) to store a sum similar to the sum in ØSC.

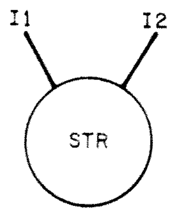
Example: The principal use is to generate envelopes for notes.

```
INS, 0, 1 ;
ENV, P5, F1, B2, P6, P7, P8, P20 ;
ØSC, B2, P9, B2, F2, P19 ;
ØUT, B2, B1 ;
END ;
```



P6, P7, and P8 determine attack, steady state, and decay times, respectively. P5 determines the maximum amplitude. P9 determines the frequency. F1 determines the envelope and F2 the oscillator waveshape. Typically P6, P7, and P8 are computed by an elaborate CØNVT function (see Chapter 2, section on Additional Unit Generators, ENV).

STR: Stereophonic Output Box (Numerical equivalent = 6)
Diagram:



Data statement: STR, I1, I2, Ø ;

Operation: This unit puts alternate samples from I1 and I2 into Ø

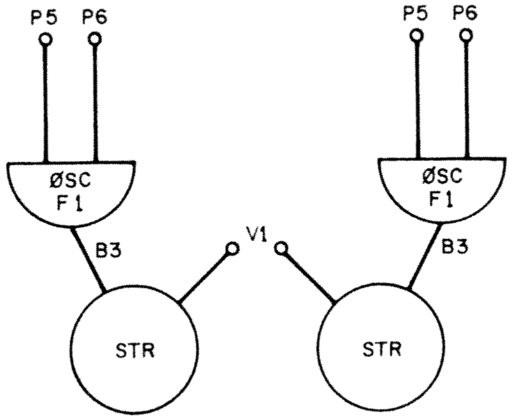
$$Ø_{2i} = I1_i$$
$$Ø_{2i+1} = I2_i$$

This arrangement is suitable for a stereophonic output conversion.

The stereophonic output requires an output block length equal to *two* input–output block lengths. Typically B1 and B2 are set aside for output storage.

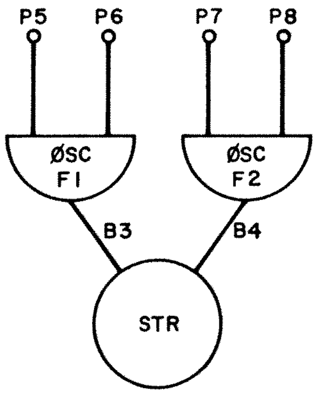
Example: Two instruments are defined which are identical except that one uses the left channel and the other the right.

```
INS, 0, 1;
ØSC, P5, P6, B3, F1, P20;
STR, B3, V1, B1;
END;
INS, 0, 2;
ØSC, P5, P6, B3, F1, P20;
STR, V1, B3, B1;
END;
```



V1 is assumed to be zero. Note that blocks B1 and B2 have been reserved for output.

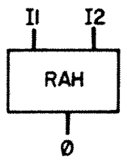
In another example, a single instrument produces sound in both right and left channels.

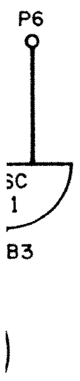


```
INS, 0, 1;
ØSC, P5, P6, B3, F1, P20;
ØSC, P7, P8, B4, F2, P21;
STR, B3, B4, B1;
END;
```

RAH: Random and Hold Function Generator (Numerical equivalent = 11)

Diagram:

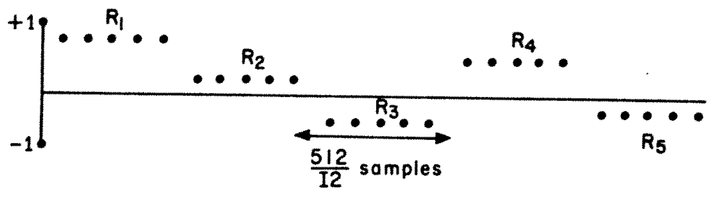




Data statement: RAH, I1, I2, Ø, S, T;
Operation: Output is generated according to

$$Ø_i = I1_i \times R_n(I2_i)$$

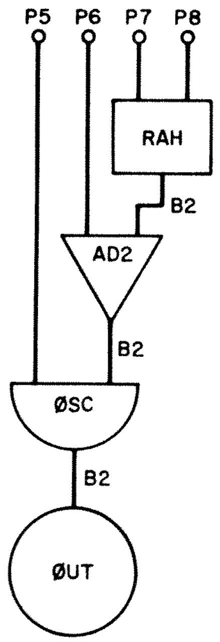
where $R_n(I2_i)$ is a succession of independent random numbers which change every 512/I2 samples. Thus this generator holds each random number for 512/I2 samples. $R_n(I2_i)$ is uniformly distributed from -1 to $+1$.



S and T are temporary storage locations which are normally kept in note parameter locations. S holds a sum equivalent to the ØSC sum. T holds the current R_i.

Example: A typical instrument to produce a succession of random pitches:

```
INS, 0, 1;
RAH, P7, P8, B2, P20, P19;
AD2, P6, B2, B2;
ØSC, P5, B2, B2, F1, P18;
ØUT, B2, B1;
END;
```



Function F1 can be any desired waveform. P7 should be at most equal to P6. The pitch frequency will assume a succession of random values between the frequencies

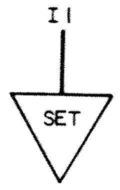
$$\frac{(P6 - P7) \times \text{sampling rate}}{\text{function length}}$$

and

$$\frac{(P6 + P7) \times \text{sampling rate}}{\text{function length}}$$

A new value of the pitch frequency is generated every 512/P8 samples.

SET: Set New Function Number in Unit Generator (Numerical equivalent = 102)
Diagram:



Data statement: SET, I1 ;
Operation: SET enables changing the function number in an ØSC or ENV unit generator by specifying the new function number as a note parameter.

In the instrument definition, SET must be just ahead of the unit generator on which it is to act; the input specifies in which P field of the note card the new function number is to appear. If this P field is given a negative or zero value, no change is effected; if it is given a positive integer value, this value is the new function number.

```
Example: INS, 0, 1 ;
         SET, P7 ;
         ØSC, P5, P6, B2, F1, P20 ;
         ØUT, B2, B1 ;
         END ;
```

With this instrument definition, all three of the following note cards

```
NØT, 0, 1, 1, 1000, 50, 0 ;
NØT, 1, 1, 1, 1000, 50, 1 ;
NØT, 2, 1, 1, 1000, 50, -2 ;
```

will leave function #1 in ØSC, whereas the note card

```
NØT, 3, 1, 1, 1000, 50, 2 ;
```

replaces function #1 by function #2 in ØSC.

6. Special Discussion of ØSC Unit Generator[4]

Probably the most basic and important unit generator used by Music V is the oscillator. Since the oscillator utilizes most of the basic

[4] This discussion of ØSC was provided by S. C. Johnson.

principles of Music V, a detailed description of its operation should prove useful in the design and implementation of additional unit generators.

The oscillator is a unit generator, meaning that it is a "device" that is useful in building "instruments." This device is simulated by a general computational *algorithm* which can produce any periodic function at any frequency or amplitude. This algorithm should be quite efficient, since it must compute 10,000–20,000 numbers for each second of sound.

Efficiency and generality are gained through the use of *stored functions*. The values of a stored function need be computed only once (by a GEN subroutine in Music V) and then may be referred to by any unit generator. By making the functions interchangeable among unit generators, we need keep only one copy of any function used and one copy of any unit generator in the computer memory.

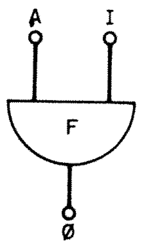
The mathematical algorithm for simulating an oscillator is described by the equation

$$Ø_i = A_i \cdot F(S_i \bmod FL)$$

and

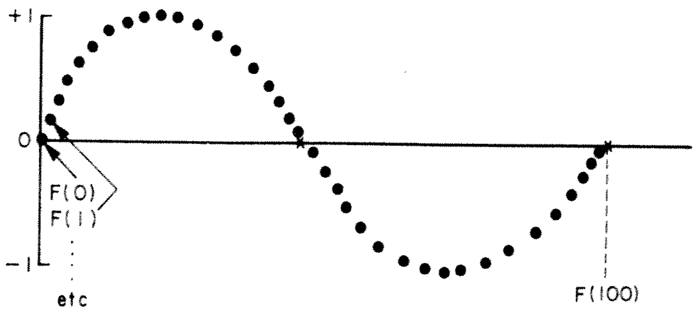
$$S_{i+1} = S_i + I_i$$

where



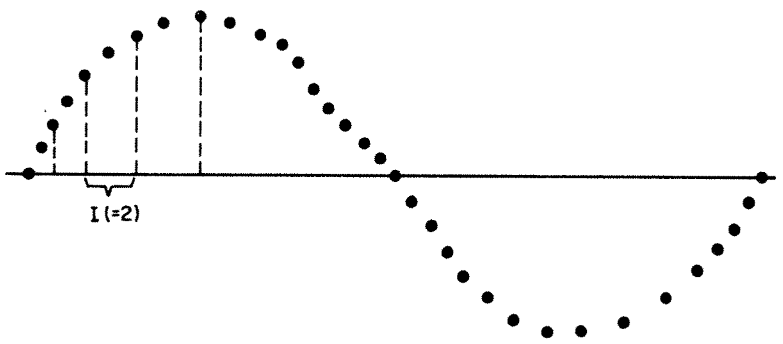
$Ø_i$ = the ith output sample
A_i = the ith amplitude input
I_i = the ith increment input (controls frequency)
F = a stored function (controls waveshape)
S_i = the ith sum of increments
FL = the length of the stored function (in samples)

Assume for a moment that the stored function is a representation of a sine wave occupying 101 computer locations, F(0), F(2), . . ., F(100).



The value of F(0) is sin (0/100 * 2π), F(1) is sin (1/100 * 2π), F(2) is sin (2/100 * 2π), etc. Since $0 \leq |\sin x| \leq 1.0$, we may multiply the values of the function by any amplitude A to produce output samples in the desired range, $0 \leq |Ø_i| \leq A$.

How does the oscillator reproduce this sine wave at any frequency? Assume that we have fixed the sampling rate at 10,000 samples per second. This means that the digital-to-analog converter will convert 10,000 samples into sound every second, and each sample number we output represents 1/10,000 second of sound. If we multiply the stored function shown above by an appropriate amplitude and output it directly, then each period of the wave will contain 100 samples, and it will be heard 10,000/100 or 100 times per second. This corresponds to a frequency of 100 Hz. Since the sampling rate is fixed, to double the frequency of the sound we must halve the number of samples per period of the wave. We do this simply by referring to *every other* value of the stored function.



Thus the output samples will be given by the relations

$$
\begin{aligned}
Ø(1) &= F(0) * A \quad (s = 0)\\
Ø(2) &= F(2) * A \quad (s = 2)\\
Ø(3) &= F(4) * A \quad (s = 4)\\
&\vdots\\
&\text{etc.}
\end{aligned}
$$

The output wave then has 50 samples per period and is heard at 10,000/50 or 200 Hz. To obtain the output $Ø_i$ in this case, the independent variable in the function F(s) is incremented by 2 each time the function is referred to. If the increment used was 4, we would output 100/4 = 25 samples per period, or a sine wave of 10,000/25 or 400 Hz. In general then

$$\text{Frequency in hertz} = \frac{\text{sampling rate}}{\text{samples per period}}$$

and

$$\text{Samples per period} = \frac{\text{function length}}{\text{increment}}$$

therefore,

$$\text{Frequency in hertz} = \frac{\text{sampling rate} * \text{increment}}{\text{function length}}$$

and

$$\text{Increment (in samples)} = \frac{\text{function length} * \text{frequency in hertz}}{\text{sampling rate}}$$

Modulus arithmetic is used in conjunction with the cumulative sum of increments S_i in order to achieve periodicity in the references to the stored function.

A final point concerns the sum of increments. Assuming a function length and sampling rate as above, the increment necessary to produce a 150-Hz tone I is (100 * 150)/10,000 or 1.5. Obviously any continuous function will have a value at S = 1.5, but we cannot directly talk about the 1.5th computer location of stored function F. Three approaches to this problem have been used: truncation, rounding, and interpolation. The fastest method is truncation, where the greatest integer ⟦S⟧ contained in the sum of increments is used as the S value. This is easily accomplished with fixed-point computer arithmetic, but may lead to some distortion of the output (see the table below). In the rounding method, we round the sum of increments to the *nearest* integer and use this as the S value. Although this takes a little more computation, it leads to better results.

In the interpolation method, the sum of increments is truncated to obtain a function value as in the truncation method. This function value is then corrected by linear interpolation: if y is the function value at F(⟦S⟧), y′ is the function value at F(⟦S⟧ + 1), and h is the amount by which the sum of increments exceeds ⟦S⟧ (= S − ⟦S⟧, or the *fractional part*), then the corrected value of the function is $y + (y' - y)h$. This method takes the most computer time but in practice produces the greatest accuracy. It can also effect a saving of memory space in the computer, since, as is shown in the table below, treating a stored function of 512 locations with truncation produces a greater distortion of the output than using interpolation on a function only 32 locations long.

The table shows the results of computing 500 values of sine x, using various methods and stored function lengths. The table entries are the percentage rms error.

Function Length	Truncation	Rounding	Interpolation
32	7.9	4.0	0.3
64	3.8	1.9	0.06
128	1.7	1.1	0.02
256	0.9	0.5	0.004
512	0.5	0.2	0.001
1024	0.24	0.12	0.0002

In general, rounding is about twice as accurate as truncation, and doubling the length of the stored function doubles the accuracy for both the truncation and rounding. Doubling the function length quadruples the accuracy for the interpolation method, however. Which method is used will depend on the availability of computer time versus memory space in a particular installation of Music V.

The distortion level of the oscillator depends on the function length and the particular numeral process used. It also depends on the particular increment used: distortion occurs only when the increment is not an integer. Finally, it depends on the waveshape used: the distortion level will increase when the slope of the stored function is steep at the point considered.

How much this distortion alters the quality of the sound is hard to predict; a function with steep slope should be expected to be more distorted than a sine wave, and yet in many cases the distortion will be more *audible* with sine waves than with complex waveforms. For instance, the synthesis of a frequency-modulated sine wave with the following parameters:

Function length = 512 samples
Sampling rate = 10,000 Hz
Frequency deviation = 3% (of fundamental frequency)
Vibrato rate = 25% (of fundamental frequency)

produces a clearly distorted sound when truncation is used, and an acceptable sound when interpolation is used. But if the sine wave is replaced by a complex tone with harmonies decreasing at 6 dB or 12 dB per octave, there is almost no audible difference between sounds synthesized with truncation and those synthesized with interpolation.

7. Input–Output Routines for Pass I and Pass II

Input for Pass I: READ0 and READ1

The interpretative input routine for Pass I is a FØRTRAN IV subroutine named READ1. It has an additional entry point called READ0 used for reading the first record. The program, as supplied with Music V, is designed for a 36-bit word machine and accepts input data punched in the free format in columns 1 through 72 of cards, as has been described in Section 3.

READ0 reads an initial record into the input buffer ICAR (equivalent to CARD). The characters are stored one per computer word and are shifted to the right end of the word by the MØVR subroutine. (MØVR is one of the machine-language routines necessary for Music V.)

The operation of READ1, the main program, is diagrammed in Fig. 52. After writing out a record to set Music V to stereo or mono (which will be discussed below), the program (at 10)[5] scans to the end of the first data statement marked by ";". If necessary, more input records are read.

The characters are organized (at 21) into fields with exactly one blank character separating successive fields. The organized data are stored in IBCD and are printed out. The first field is compared with all possible mnemonics that may be written in it. If a match is found, the numerical equivalent of the mnemonic is found and one of a number of branches (at 29) is taken depending on the value of the first field.

If no match is found for the first field (at 40), it is taken to be a number if the data statement is inside an instrument definition. Otherwise, an error comment is made and the statement is rejected.

The remaining fields on the data statement are converted to numerical form by one of several sections of the program (218, 201–210, 100, 300–1200, 200, 217, 220, and 30) depending on what the first field is and whether the data statement is part of an instrument definition.

All score records have a mnemonic operation code as the first field and an action time as a second field *except* the second through the last cards in an instrument definition. In an instrument definition, such as the one given below

```
INS, Action time, Inst No ;
ØSC, P5, P6, B2, F1, P20 ;
ØUT, B2, B1 ;
END :
```

[5] The numbers cited in the descriptions of programs refer to statement numbers in the FØRTRAN program. These numbers are also shown in the block diagrams.

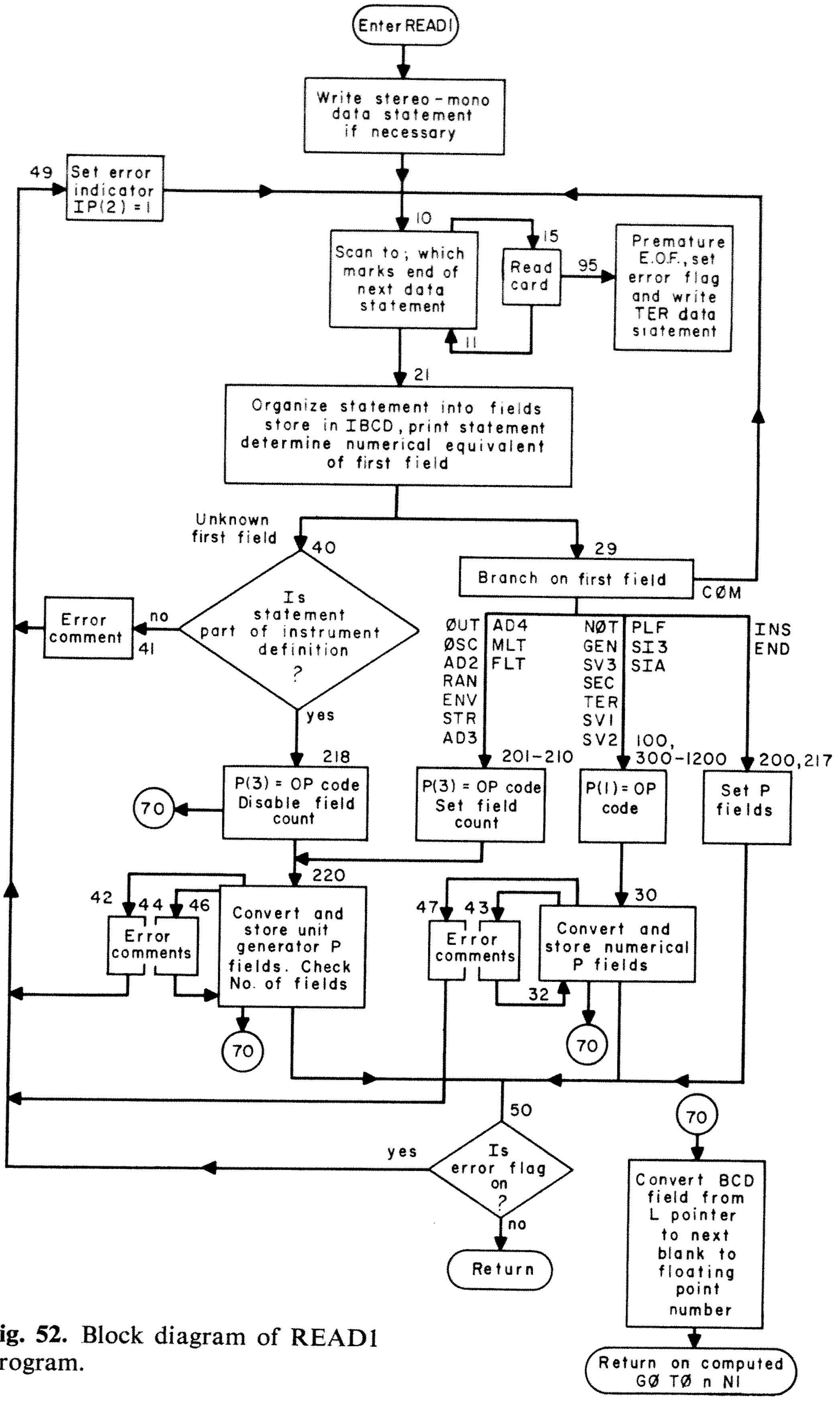

Enter READ1
Write stereo-mono data statement if necessary
49
Set error indicator IP(2) = 1
10
Scan to ; which marks end of next data statement
15
Read card
95
Premature E.O.F., set error flag and write TER data statement
11
21
Organize statement into fields store in IBCD, print statement determine numerical equivalent of first field
Unknown first field
40
29
Branch on first field
CØM
Is statement part of instrument definition ?
no
yes
Error comment
41
ØUT ØSC AD2 RAN ENV STR AD3
AD4 MLT FLT
NØT GEN SV3 SEC TER SV1 SV2
PLF SI3 SIA
INS END
218
201-210
100, 300-1200
200,217
P(3) = OP code Disable field count
P(3) = OP code Set field count
P(1) = OP code
Set P fields
70
220
42
44
46
Error comments
Convert and store unit generator P fields. Check No. of fields
47
43
Error comments
30
Convert and store numerical P fields
32
70
70
50
Is error flag on ?
yes
no
Return
70
Convert BCD field from L pointer to next blank to floating point number
Return on computed GØ TØ n NI

Fig. 52. Block diagram of READ1 program.

the operation code and action time appear only on the first data record. READ1 takes the operation code (= 2) and the action time from the first data record and stores these in P(1) and P(2) in all subsequent records connected with instrument definition. The value of P(3) is the type number of the unit, and the remaining fields are interpreted and converted to floating point and stored starting in P(4). Word (i.e., field) count for the statement is established in IP(1).

The conversion from BCD to floating point is done by a subroutine (at 70) which finds the position of the decimal point in the field of characters (or supplies it at the end if missing) and then multiplies the characters, which are expressed as integers, by the appropriate power of 10 and sums over all characters in the field.

Any errors that are detected cause an error comment to be printed below the printout of the data statement in which the error occurred. In cases other than that of an incorrect operation code, the entire statement is scanned so that all errors will be detected. Incorrect operation codes, however, prevent proper interpretation of the remaining fields in the data statement. When errors occur, a flag is set in CØMMØN storage (namely IP(2) is set to 1) so that Pass I may terminate the job at its conclusion. Furthermore, when errors are detected, the data statement is not returned to Pass I but control returns to the entry point of READ1 to obtain the next data statement.

It will be noted that the input array for the card data is named CARD which is (FØRTRAN) equivalent to ICAR. Also, IBCD is equivalent to BCD. This equivalence is necessary because the characters when read in with format A1 require a floating-point designation. However, for purposes of comparison, the data must be regarded as in integer form. Hence, the characters must be right adjusted (moved to the right and of the computer word). Similarly, when the organized data statement is to be printed out, it must be put back into left-adjusted form so that it may be printed out in A1 format. Consequently, this routine uses two subroutines, which must be written in machine language and, therefore, supplied by the user. READ1 (and READ0) makes the following calls:

```
CALL MØVR (CARD, NC)
CALL MØVL (CARD, NC)
```

The characters stored in NC consecutive locations of CARD are right (left) adjusted and replaced in the same locations. Calls to MØVR are found after the two "read" statements, and the "print" statement is preceded by a call to MØVL and followed by a call to MØVR.

READ1 inspects the output unit generators (ØUT or STR) in the instruments, and if a change from monophonic to stereophonic operation or vice versa is made, it writes out an appropriate stereo or mono control record at the end of the instrument definition. The record is

$$\text{SIA TA 8} \begin{pmatrix} 0\ (= \text{mono}) \\ 1\ (= \text{stereo}) \end{pmatrix} ;$$

The action time TA is the same as the action time for the instrument definition. Inspection is done by the first statement in READ1. END equals 1 at the end of an instrument definition and equals 0 otherwise. SNA8 equals 1 if the mono-stereo mode is changed and equals 0 otherwise. STER = 0 if the last out box is ØUT, and STER = 1 if the box is STR. Music V is assumed initially to be in the monaural mode.

If the program is to be run on a machine of different word length, the FØRTRAN DATA statements for arrays IBC, IVT, and LØP must be changed. These contain right-adjusted BCD characters for the break characters used in delimiting the input, the parameter types P, V, F, and B used in specifying unit generators, and the characters used in the three-letter mnemonic names for operation codes. In a 36-bit machine such data are entered as 6H00000X, in a 24-bit machine as 4H000X. If the input data are to be read in from any medium other than cards, the two "READ" statements at 15 and under READ0 need to be changed as required. The number of characters obtained by executing a "READ" command is a variable NC, established in this version of the program as 72. The arrays that hold these data have been dimensioned to accept a maximum value of NC equal to 128.

The break characters delimiting the fields of input data are the blank, comma, and semicolon; NBC, the number of break characters, is equal to 3. (If typewriter input is to be substituted, an additional break character may be needed equal to the carriage return.)

The most frequent change in READ1 is the addition of other OP codes or unit-generator names. The following steps will accomplish this change:

(1) Add the three-character mnemonic to the end of the LØP array. (The size of the LØP array may or may not need to be increased, the word count on the LØP DATA statement must be increased.)
(2) Increase NØPS by 1.
(3) Put another branch at the end of the GØ TØ at 29.

(4) Write appropriate code for the branch. The code for branches 201–210 or 300–1200 will usually serve as a model.

A few of the variables in the program are

NUMU	Normally equals 0. It is set to 1 to disable checking the number of fields in a unit generator.
NPW	The number of fields expected in a unit generator, not counting the name. For example, AD3 P1 P2 P3 B1 ; would have NPW = 4.
L	Scanning index for the IBCD array. It normally points to the character just ahead of the next field to be processed. L is changed by many parts of the program, including the BCD to floating-point converter.
I	Scanning index for the ICAR array.
J	Scanning index to store characters in the IBCD array.

Output for Pass I: WRITE1

The number of fields in a data statement is established by READ1 and stored in IP(1) located in CØMMØN. The fields of the data statement have been interpreted and the P array appropriately filled by READ1. Thus after Pass I has properly processed the data statement, it may call WRITE1 to write the data statement on tape or disc so as to be available to Pass II. The call from Pass I is as follows:

```
CALL WRITE1 (10)
```

WRITE1 sets N = IP(1) and writes the list N, (P(I), I = 1, N) onto data file 10 in binary format.

Input for Pass II: READ2

Pass II calls upon subroutine READ2 with the call CALL READ2 (10) to read N, (P(I), I = 1, N) from data file 10 and establish IP(1) = N.

Debug READ0 and READ1

For testing purposes, an all-FØRTRAN score-reading program is provided to replace READ0 and READ1. The program reads an all-numerical score. Each data statement has the following information:

```
N P(1) P(2) . . . P(N)
```

where N is the word count and the subsequent fields destined for the first N locations of the P array.

The calling sequences are the same as those of READ0 and READ1, namely

```
    CALL READ0
```

and

```
    CALL READ1
```

Debug READ0 does nothing.
Debug READ1 reads one statement into the P array according to the FØRTRAN statement

```
    READ1, K, (P(J), J = 1, K)
```

where the format statement 1 is

```
    1 FØRMAT(I6, 11F6.0/(12F6.0))
```

Thus twelve numbers are read from the first 72 columns of each card.

8. PLF Subroutines

A data statement of the type

```
    PLF 0 n D4 D5 . . . Dm ;
```

will cause the following call to take place during Pass I at the time the data statement is read

```
    CALL PLFn
```

where n is some integer between 1 and 5. PLFn is a subroutine which must be supplied by the user. These subroutines can perform any function desired by the user. Usually they will generate data statements for Pass II or manipulate Pass I memory (the D(2000) array).
The information of the data statement PLF, 0, n, D4, . . ., Dm will be placed in the P(100) array in P(1) – P(m) at the time PLFn is called.
The P, D, and IP arrays are kept in common storage and hence are available to the PLFn routine. The dimension and common statements in the PLF routine and in Pass I must, of course, agree. For examples and a further discussion of PLF subroutines, see Chapter 2, section on Composing Subroutines—PLF.

9. General Error Subroutine

A general-purpose ERRØR subroutine is used by all three passes. A statement

```
    CALL ERRØR(N)
```

will cause the following comment to be printed

ERRØR ØF TYPE N

where N is an integer.

The meaning of N is as follows

	N
Pass I errors	
Nonexistent OP code on data statement	10
Nonexistent PLF subroutine called	11
(i.e., in call PLF_n, $n < 1$, or $n > 5$)	
Pass II errors	
Too many notes in section, D or I array full	20
Incorrect OP code in Pass II	21
Incorrect OP code in Pass II	22
Nonexistent PLS subroutine called	23
(i.e., in call PLS_n, $n < 1$, or $n > 5$)	
Pass III errors	
Incorrect OP code in Pass III	1
Code is < 1 or > 12	
Too many voices simultaneously playing[a]	2
Too many voices simultaneously playing	3

[a] The maximum number of voices must be equal to or less than the number of note parameter blocks (see Section 16).

In addition to these error comments, the READ1 will print error comments if it detects errors in or cannot interpret any data statements. These comments are described in Section 7.

10. Description of Pass II

Pass II performs three general functions:

(1) Sorting the data records obtained from Pass I into forward chronological order according to starting times,
(2) Applying special conversions to some of the input data records by calling the user-supplied CØNVT subroutine, and
(3) Applying the metronomic time-scaling operations to starting times and durations.

After completing these functions, Pass II writes its output onto data file 11 for subsequent use by Pass III (if desired, a Pass II report is printed, see WRITE2 below). The entire operation may be diagrammed as shown in Fig. 53.

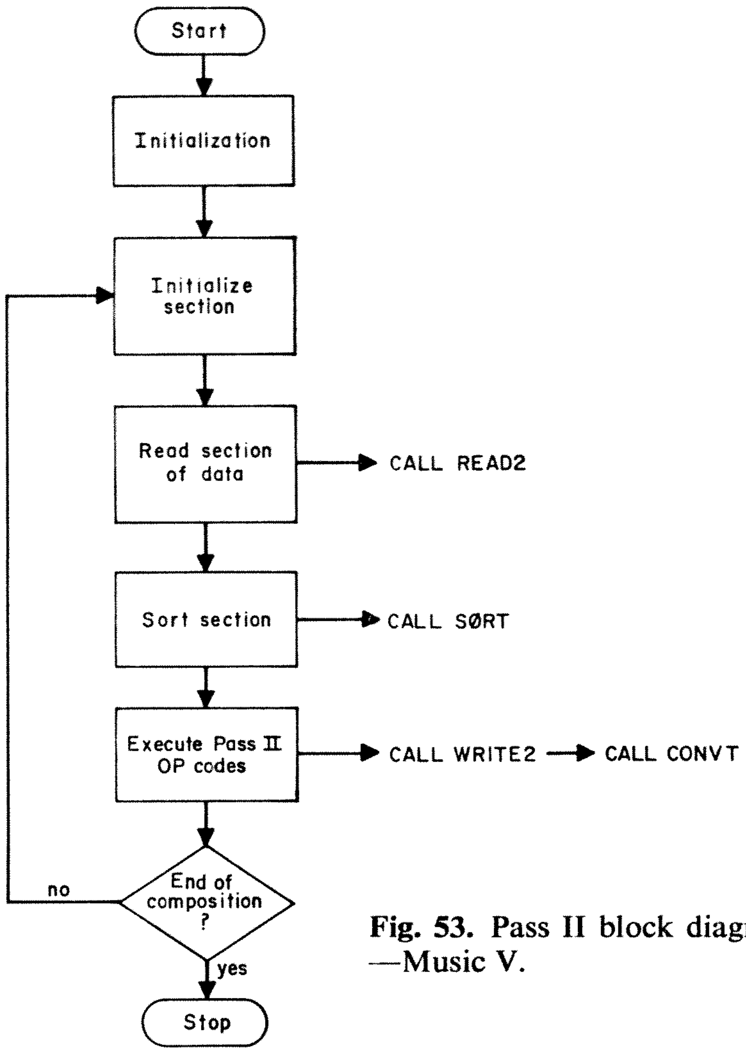


Fig. 53. Pass II block diagram —Music V.

Pass II maintains the following arrays in unlabeled common storage

CØMMØN IP(10), P(100), G(1000), I(1000), T(1000), D(10,000)

After setting certain variables (standard sampling rate, size of D and I arrays, and number of OP codes) to their initial values, Pass II calls on READ2(10). READ2 then reads information from data file 10 according to the format: (K, P(I), I = 1, K). The value of K is stored into IP(1). This call is repeated until an entire section has been read in and the data statements are accumulated in the D array. The I array is used to hold subscripts that point to the beginning of each data record in the D array. The T array is used to hold the action time of each data statement P(2). After an entire section has been read in, Pass II sorts the T array into ascending numerical order by calling SØRTFL and SØRT (SØRTFL is merely an initialization routine which informs the sort-program package that floating-point numbers are about to be sorted).

It also sorts the I array (pointers) as a passive list on T, so that after SØRT has been called this point list has been rearranged according to the starting times of the data statements.

Each data statement is then accessed from the D array according to the order specified by the I pointers (chronological order of action times). Each data statement is inspected to see:

(1) If the OP code is 8 or 12 (SV2 and SIA, respectively), then the variable list (i.e., P(4) through P(4 + n)) is stored into the G array starting at G (P(3)).
(2) If the OP code is 10 (PLS), then a call to PLS_n is generated, where n is the number stored in P(3).
(3) If the OP code is 7 or 9 (SV1 and PLF, respectively), an error message is printed (error of type 22). This error is not fatal, however, and Pass II merely ignores the offending instruction.
(4) If the OP code is 1, 2, 3, 4, 5, 6, 11, 12 (NØT, INS, GEN, SV3, SEC, TER, SI3, SIA), the subroutine WRITE2(11) is called. This subroutine applies the optional metronome time-scaling operations, prints the optional Pass II report, calls the subroutine CØNVT to modify any note parameters, and writes out a record on data file 11 for subsequent use as input to Pass III.

    The record is written according to statements

```
K = IP(1)
WRITE(11), K, (P(J), J = 1, K)
```

    K, which is kept in IP(1), is the word count. Data are in the P(100) array. Details of the operations done by WRITE2 are discussed in Section 11.

After a section has been processed, the next section is read. The section-reading sequence is terminated by a TER card via a flag IEND which is set to 1 when TER is encountered. IEND is checked after each section is processed.

The error comments produced by Pass II are printed by ERRØR and are discussed in Section 9.

Pass II contains a general-purpose memory, G(1000), which is primarily used by the PLS subroutines and by the CØNVT function. Blocks of locations starting at G(n) may be set with a SV2 AT n x . . . ; record. The setting occurs at the action time, AT, relative to the other data records.

Certain locations in the G array have special functions:

G(1)	Flag controlling Pass II report (= 0, print report; = 1, suppress report)
G(2)	Time-scaling flag (G(2) = 0 for no time scaling; G(2) = n for time scaling where metronome function starts at G(n))
G(4)	Sampling rate
G(5)	Starting beat of note
G(6)	Duration of note in beats
G(8)	Stereo-mono flag (= 0, mono; = 1, stereo)

The IP array contains certain other parameters:

IP(1)	Word count for current record in P array
IP(2)	Location in D array of beginning of data statement that is currently calling a PLS subroutine
IP(3)	Number of data statements in the D array

11. WRITE2

Pass II calls WRITE2(11) in order to:

(1) Invoke the optional metronome operations described below
(2) Produce the optional Pass II report on the printer
(3) Call CØNVT to modify data record parameters
(4) Write (N, P(I), I = 1, N) on data file 11 for subsequent use by Pass III

In order to utilize the metronome operations available in Pass II, a nonzero value must be stored in the array location G(2). This value is the beginning subscript in the G array of a tempo function such as the one shown in Fig. 54. This is a function constructed of any number of

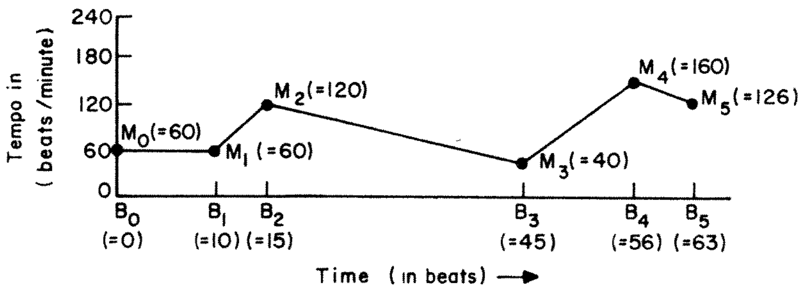


Fig. 54. Tempo function.

line segments. It is stored, beginning at G(G(2)), as an arbitrary-length list of number pairs, B_0 (= 0), M_0, B_1, M_1, B_2, M_2, . . ., B_n, M_n where M_i is the standard metronome marking (in beats per minute) at beat

B_i of the composition. See Chapter 2, section on Compositional Functions, for additional discussion.

WRITE2 uses the FØRTRAN function CØN(G, I, T) (see Section 12) to calculate the value at beat T of the function which is stored at G(I). WRITE2 then converts P(2), the starting time, and P(4), the duration,[6] from beats to seconds according to the following two relationships:

$$T_i = T_{i-1} + (B_i - B_{i-1}) \cdot \left(\frac{60}{F(B_i)}\right)$$

where

T_i is current time in seconds, which replaces the value in P(2)
B_i is current beat number, the value found in P(2)
$F(B_i)$ is the value of the tempo function at beat B_i and
T_{i-1} and B_{i-1} are the time and beat of the previous data record

and

$$D_i = L_i \cdot \left(\frac{60}{F(B_i)}\right)$$

where

D_i is the duration in seconds
L_i is the duration of note in beats and
$F(B_i)$ is as above

The tempo function itself may be placed into the G array via an SV2 instruction. The function shown in Fig. 54, for example, could be placed in the G array beginning at G(30) by the data records:

```
SV2, 0, 2, 30 ;
SV2, 0, 30, 0, 60, 10, 60, 15, 120, 45, 40, 56, 160, 63, 126 ;
```

These metronome operations can be turned off at any time by setting G(2) at 0. If the metronome operations are so turned off, P(2) and P(4) are not affected by WRITE2 and are assumed to be in seconds.

The Pass II report is printed automatically by WRITE2 if G(1) = 0. The Pass II report may be suppressed by setting G(1) $\neq$ 0 with an SV2 instruction (e.g., SV2, 0, 1, 1 ;). It consists of each data statement printed in order of ascending action times. Each data statement is shown exactly as it is presented to Pass III (if the data statements do

[6] Durations are given on NØT cards only. P(4) is affected if and only if P(1) = 1 (play a note).

not exceed 10 fields, they are printed one per line; longer data statements are continued on next line). In addition, if the metronome function is in use, P(2) and P(4) will have been converted into seconds, and the original values of these parameters (in beats) are printed to the right of each print line.

WRITE2 calls CØNVT immediately before it returns. G(5) and G(6) contain the original values of P(2) and P(4) if metronomic scaling was used.

12. CØN—Function Evaluator for Line-Segment Functions

CØN evaluates functions formed from a sequence of line segments. These functions are useful for representing compositional functions such as the metronomic marking. CØN is used by the time-scaling routines in Pass II. It may also be used by PLF and PLS subroutines.

CØN can be evoked by a statement such as

```
Y = CØN(G, I, T)
```

which will set Y equal to the value at time T of the function stored at G(I).

CØN expects to find a pair list in the G array beginning at subscript I. The form of the list is $X_1, Y_1, X_2, \ldots, X_n, Y_n$, where X_i and Y_i are the abscissa and ordinate values for the breakpoints of the function. As many breakpoints as desired may be used. Breakpoints do not need to be equally spaced along the abscissa. If T falls between two breakpoints (as it usually does), CØN computes Y as a linear interpolation between the adjacent breakpoints.

As an example, the function shown in Fig. 55 would be stored starting at G(30) in Pass II by the statement

```
SV2 0 30 0 1 10 12 20 1 ;
```

Its value at 13 would be obtained by

```
Y = CØN(G, 30, 13) = 8.7
```

13. SØRT AND SØRTFL

SØRT and SØRTFL are two utility routines in the Bell Laboratories utility library on the GE645 computer. They are called by Pass II when it arranges the data statements in chronological order according to action times.

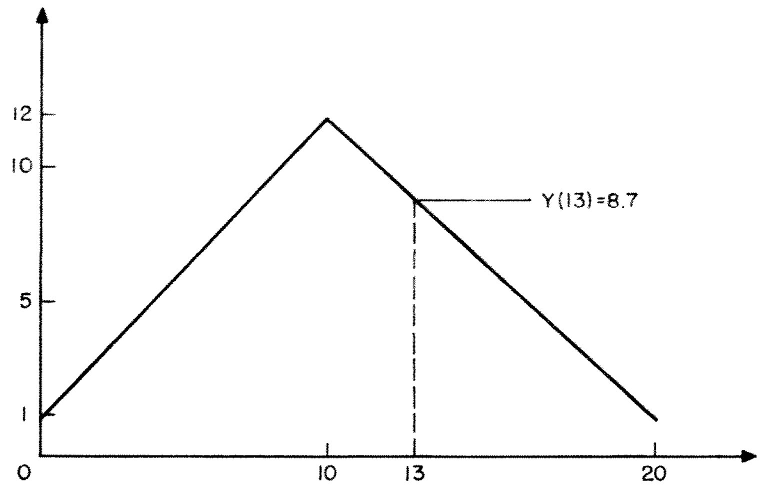


Fig. 55. Linear interpolation between two breakpoints.

SØRTFL is an initializing routine called to specify that floating-point numbers are to be sorted. The calling sequence for SØRTFL used in Pass II is simply

CALL SØRTFL

The calling sequence used in Pass II for the sort routine is

CALL SØRT (T(1), T(2), IN, I)

where T(1) and T(2) are the first two words on a list that will be sorted into monotonic increasing order, IN is the number of words to be sorted, and I is the location of the first word of a second list which will be rearranged in exactly the same manner as the T list.

In Pass II, T contains action times for data statements and I contains pointers to where the data statements begin in the D array. After sorting, the pointers in the I array will have been so rearranged that successive pointers point to data statements in the D array in their proper chronological sequence.

If two entries in the T array are equal, SØRT will *not* interchange their order. The preservation of order is essential for the data statements that define an instrument. All these have the same action time, but their order must be maintained.

14. PLS Routines

A data statement of the type

PLS AT n D4 D5 . . . Dm ;

causes Pass II to execute the statement

CALL PLSn

where n is some integer between 1 and 5. The call is carried out at action time AT relative to the processing of other data statements in Pass II. PLSn is a subroutine that must be supplied by the user. It can perform any desired function, but a typical use would be to change a note parameter, such as pitch, according to some composing rule. For more information on the use of PLS routines, see the tutorial examples in Chapter 2.

The data statement PLS AT n D4 . . . is stored in numerical form in the Pass II D(10,000) array at the time the call to PLSn takes place. The arrangement is

```
D(M) = word count
D(M + 1) = 10   (the numerical equivalent of PLS)
D(M + 2) = AT
etc.
M = IP(2)
```

Thus, for example, in order to find D5 from the data statement, PLS must look up M at IP(2) and then look in G(M + 6). Such a roundabout procedure is necessary because of the sorting.

The dimension and common statements in Pass II and PLSn must, of course, be identical.

15. CØNVT—Convert Subroutine

This subroutine is called by WRITE2 immediately before each data statement is written out to be used as input by Pass III. It must be supplied by the user and replaces CVT functions in Music IV. Special conversion of input parameters are possible, such as converting a frequency given in cycles per second to an appropriate increment, conversion of a special amplitude notation to a form acceptable to Pass III, and so forth. Attack, steady-state, and decay times may be converted to correct increments for driving the ENV generator.

The necessary FØRTRAN CØMMØN statement is

```
CØMMØN IP(10), P(100), G(1000)
```

CØNVT is called by the statement

```
CALL CØNVT
```

At this time the parameters for the data statement are in the P array, and the number of parameters is in IP(1). G(5) and G(6) contain the starting time and duration in beats, if the metronomic scaling has been used.

CØNVT may perform complicated logical functions. It may increase or decrease the number of parameters, changing IP(1) accordingly. For more information, see the tutorial examples described in Chapter 2.

16. Description of Pass III

Pass III reads a sequence of data statements that have been ordered according to increasing action times, and it executes the operations specified by these data statements. The principal operations are defining instruments and playing notes. In addition, functions, variables, and numbers may be computed and stored in the Pass III memory for subsequent use in playing notes.

As mentioned in the introduction, most of the data in Pass III are stored in a large linear array I. Included are instrument definitions, input–output blocks for unit generators, functions, note parameters.

The size of the various parts of I will vary greatly, depending both on the specific computer being used and on the composition being played. Consequently the structure of I is described in the IP data array, which may be easily changed. Details are given in Section 17.

Over-all Operation

The over-all operation of Pass III is diagrammed in Fig. 56. The program is started by reading a few constants from IP, including the sampling rate IP(3) and the scale factor for variables IP(12).

A section is started by resetting the "played to" time T(1) to zero, since time is measured from the beginning of each section.

The main loop of Pass III consists simply of reading a data statement into the P array. As in previous passes, the P array is used exclusively for reading and processing data statements. The operation code always appears in P(1) and the action time in P(2). Samples of the acoustic output are generated until the "played to" time equals the action time. Then the operation code is interpreted and executed. The next data statement is then read and processed.

Instrument Definitions

If the operation code defines an instrument, the definition is entered in the I array starting with the first empty location in the table for instrument definitions. The location of the beginning of this instrument definition is recorded in the location table for instrument definitions. Different instruments are designated by being numbered.

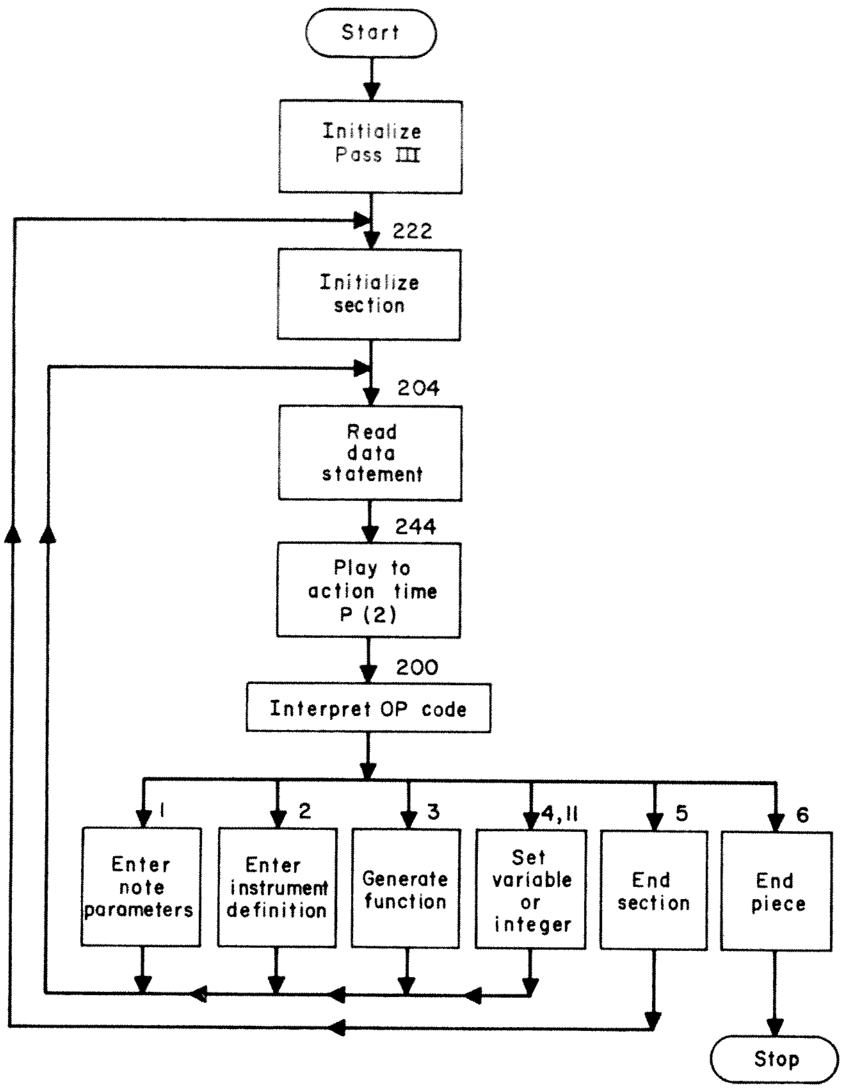


Fig. 56. Block diagram of main loop—Pass III.

An instrument definition consists of a list of the various types of unit generators used by the instrument, together with the inputs, outputs, and functions for these units. Inputs and outputs can be note parameters obtained for each note from a data statement, variables that are single numbers stored more or less permanently in the I array, or input–output blocks. These blocks are used for intercommunication between unit generators. Instrument definitions continue unchanged from one section to the next—unless they are redefined, in which case the latest definition applies.

Note Playing

If the operation code specifies a note to be played, the data from the P array are moved into the first unused block of locations in the note-parameter storage area. Unused note-parameter blocks have -1 in their first location; otherwise this location contains the instrument number. To scale amplitudes the P(2) $\rightarrow$ P(n) parameters are multiplied by IP(12) before storage in the note-parameter block (see paragraph on scale factors, p. 157). The *number* of note-parameter blocks determines the *maximum number of voices* that may be played simultaneously.

The termination time of the note is entered into the first unused location of the TI array, and the I array subscript of the note parameters is entered in the corresponding location of the ITI array. Unused locations in TI are marked with the number 1000000.0. TI and ITI are used to control the synthesis of samples of the acoustic waveform.

Play to Action Time

The most intricate part of Pass III consists in generating acoustic samples until the "played to" time T(1) equals the action time P(2) of the current data statement. In the process, any notes terminating before P(2) are turned off at their proper termination times. Several steps are involved. This part of the program is diagrammed in Fig. 57.

The action time P(2) is put in the current "play-to" objective T(2). The TI array that contains the terminating times of the instruments currently playing is searched for the minimum termination time TMIN. If TMIN $<$ T(2), acoustic samples are generated until T(1) $=$ TMIN, TMIN is removed from TI, and the whole process is repeated. If TMIN $>$ T(2), samples are generated until T(1) $=$ T(2) and control is returned to the operation-code interpreter (FØRTRAN statement 200). If TI is or becomes empty, a rest is generated until T(1) $=$ T(2). The algorithm just described starts at FØRTRAN statement 244 as indicated in Fig. 58.

The playing routines start at statement 260. The number of samples to be generated ISAM is computed as the product of the sampling rate I(4) times the time to be currently generated T(3) $-$ T(1). T(3) is the current objective for T(1). Sample generation proceeds by blocks. The length of the block is the minimum of (1) the length of a unit-generator input–output block, or (2) the number of samples remaining in ISAM. For each block of samples the program scans all note-parameter blocks (statement 268). For each voice that is turned on (first note parameter $\neq -1$) the program scans the instrument definition specified by the first note parameter (statement 271). Each unit generator specified in the instrument definition is called in the order in which it occurs in the definition. Either SAMGEN or FØRSAM is called, depending on

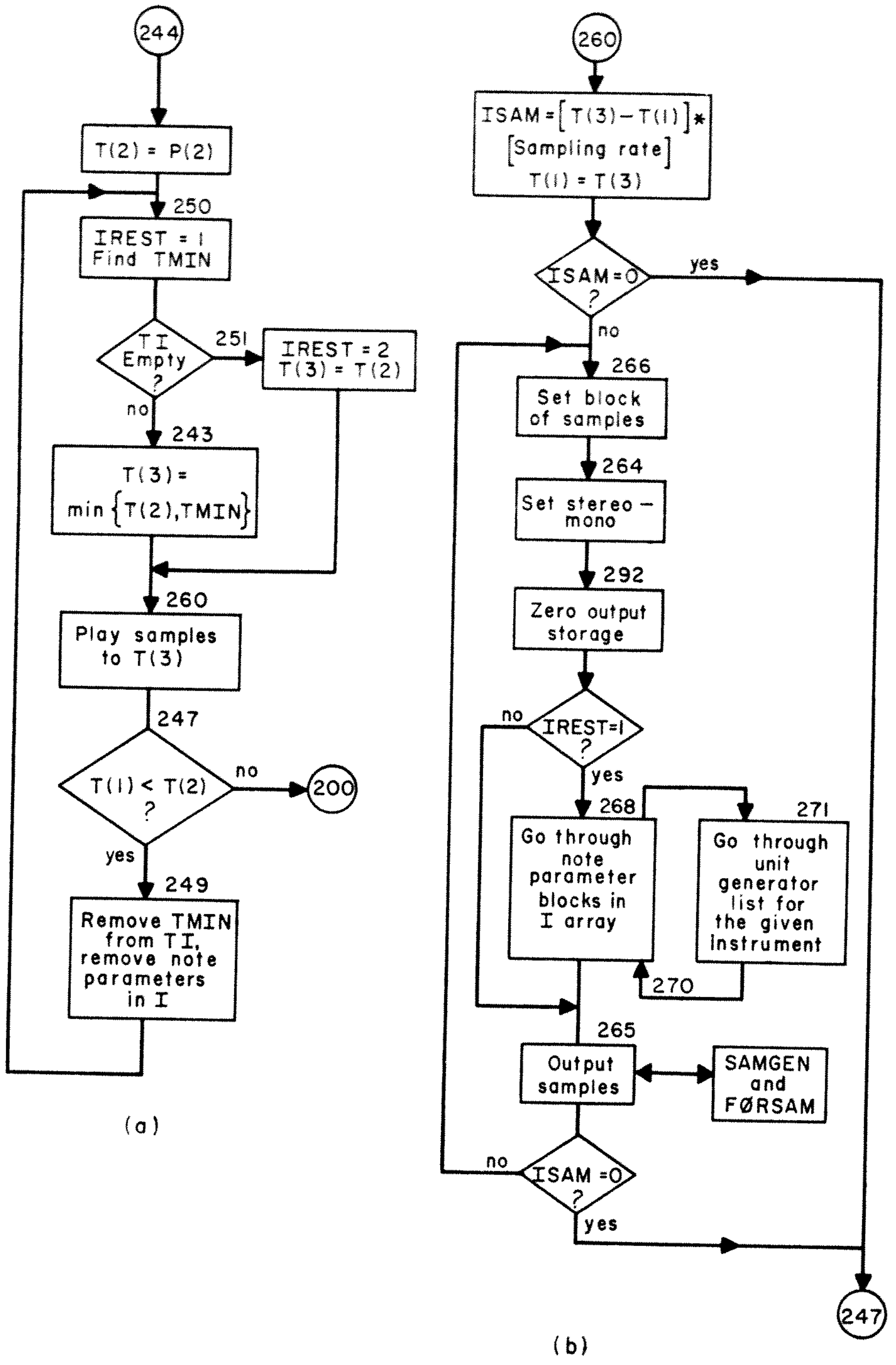


Fig. 57. Expansion of complicated parts of Pass III: (a) expansion of "play to action time," block 244; (b) expansion of "play samples to T(3)," block 260.

whether the unit-generator number is an integer less than or greater than 100.5.

After all unit generators in all instruments have produced a block of samples, the block is outputted with SAMØUT (statement 265). Another block is generated until ISAM is reduced to zero.

Function Generation

Some unit generators, in particular oscillators, use stored functions. These are computed and stored by FØRTRAN subroutines GEN1, GEN2, . . . which may be supplied by the user. Existing GENn functions for computing straight-line functions and sums of sinusoids are described in Section 25. Upon reading a data statement that requests function generation, Pass III calls upon the requested function. Space to hold the functions is provided in the I array.

Scale Factors

Because FØRTRAN stores only integers in fixed-point arrays such as I, variables that are inputs to unit generators are multiplied by IP(12), which is set equal to 2^n. This is equivalent to putting the decimal point n places from the right end of the memory word. For a machine with a 36-bit word, n is typically 18. Likewise functions are multiplied by IP(15) which is typically set to $2^{35} - 1$ in a 36-bit machine.

Variable and Number Storage

The operation codes SV3, SI3, and SIA cause numbers and variables to be stored in the I array. Variable number 1 is stored at I(101), number 2 at I(102), etc. The appropriate P field is multiplied by the scale factor IP(12) before storing. Thus I(m) equals IP(12) * P(n).

Integers are stored starting with integer 1 at I(1), integer 2 at I(2), etc. In general, these numbers are used to control the program. The following locations in I have special uses

I(4)	Sampling rate
I(7)	Master random number
I(8)	Mono-stereo control. I(8) = 0 for monophonic output, I(8) = 1 for stereophonic

These may be changed as desired. Otherwise I(1) through I(20) are reserved for program control and should not be changed. The SI3 and SIA operations do not use a scale factor.

Multiple-Use Instruments and Unit Generators

The structure of Pass III has been designed so that the same block of code embodying a unit generator is used in all instruments. Furthermore the same instrument can simultaneously (in the sense of time of the acoustic wave) produce many voices. This requires that no data specific to a given instrument or voice can be stored in the unit-generator code.

Note-parameter blocks in the I array are kept intact for the duration of a note. Hence certain quantities that must be continuous throughout the note, particularly SUM in the oscillator, should be kept in the note-parameter block.

Input–output blocks for unit generators must not be incorrectly overwritten inside an instrument. The same block may be used as input and output to a given unit generator, since the input is read before the output is written. However, a block cannot be used simultaneously for two different purposes, for example, as two inputs to a unit generator. That is, it should be kept in mind that an input–output block may contain only one set of values at a time (see Fig. 58).

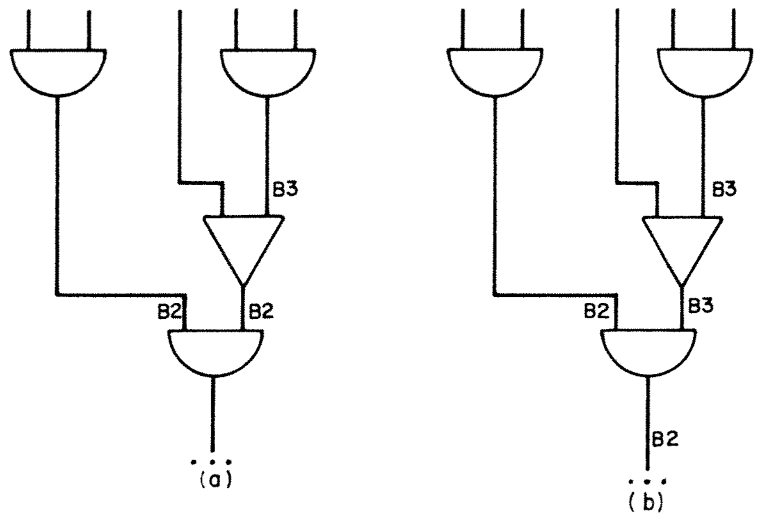
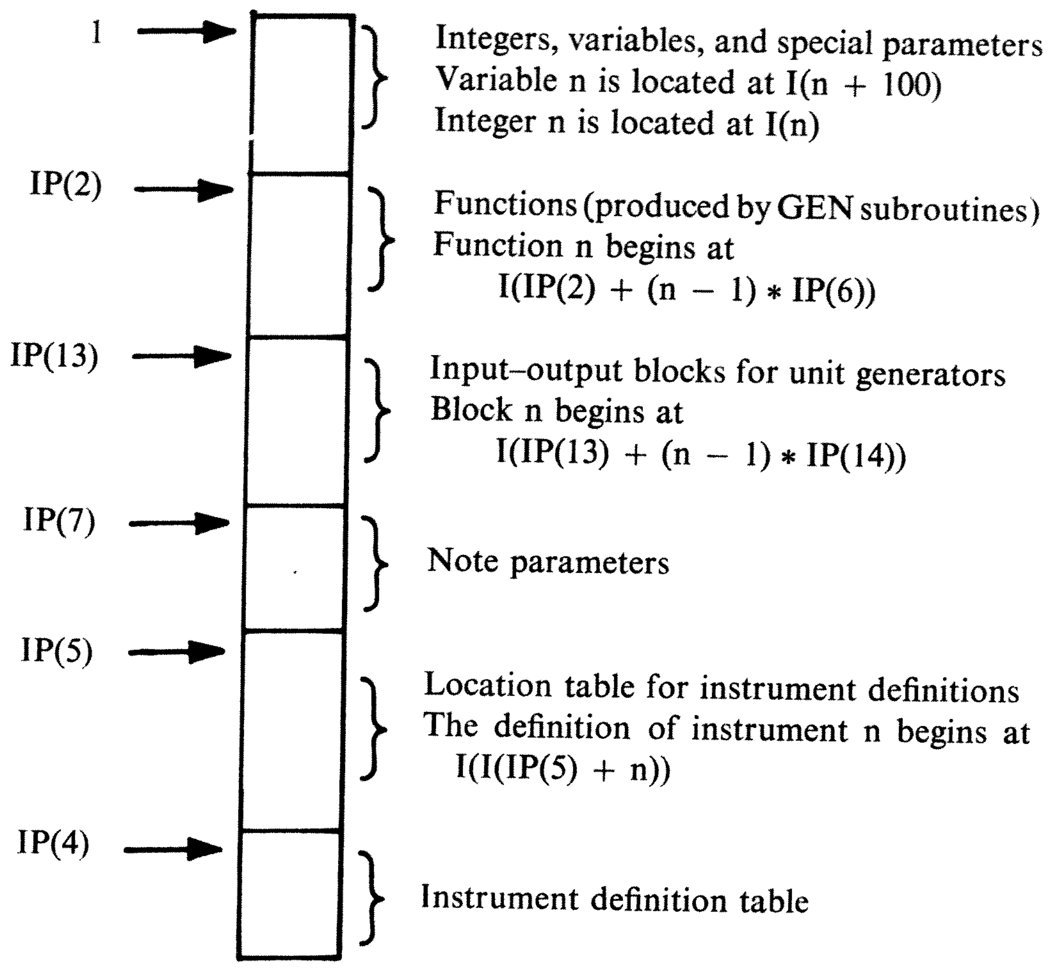


Fig. 58. Examples of (a) an incorrect and (b) a correct input–output block.

17. I and IP Data Arrays in Pass III

Most of the data in Pass III are kept in a large one-dimensional array I(n). Included are instrument definitions, note parameters, functions, unit-generator input–output blocks, and a few other miscellaneous data. Except for a few fixed locations which will be listed below, the data arrangement is flexible and is determined by parameters compiled into the IP(n) parameter table. IP contains the main Pass III constants which may change from time to time or from one computer to another—constants such as the number and size of the functions, scale factors for variables and functions, the sample value equal to zero pressure in the acoustic output wave, etc.

The I array is usually structured as shown in the diagram below. Values stored in IP give the subscripts in I(n) at which various quantities are stored.


1
IP(2)
IP(13)
IP(7)
IP(5)
IP(4)
Integers, variables, and special parameters
Variable n is located at I(n + 100)
Integer n is located at I(n)
Functions (produced by GEN subroutines)
Function n begins at
I(IP(2) + (n − 1) * IP(6))
Input–output blocks for unit generators
Block n begins at
I(IP(13) + (n − 1) * IP(14))
Note parameters
Location table for instrument definitions
The definition of instrument n begins at
I(I(IP(5) + n))
Instrument definition table

For example if IP(2) = 1000, functions will start at I(1000).

Certain special parameters in I have fixed locations and a particular meaning, as follows

I(1)	Number of words on the current data statement in the P(n) array
I(2)	Subscript of first empty location in instrument definitions
I(3)	Subscript of note parameters for the note currently being played
I(4)	Sampling rate
I(5)	Number of samples to synthesize in the current group
I(6)	Subscript of starting location in the instrument definition for the unit generator currently being played
I(7)	Master random number
I(8)	Monophonic-stereophonic signal I(8) = 0 for monophonic; I(8) = 1 for stereophonic

Any location in the I array may be set by an SV3, SI3, or SIA operation. In the set-variable operation the scale factor for variables is used so that

$$I(n) = IP(12) \cdot P(m)$$

whereas for integers no scale factor is involved

$$I(n) = P(m)$$

The following constants are compiled into the IP(n) array. The array is constructed by a BLØCK DATA subprogram and is stored in labeled CØMMØN memory, labeled PARM.

IP(1)	Number of operation codes in Pass III
IP(2)	Beginning subscript of functions
IP(3)	Standard (default) sampling rate
IP(4)	Beginning subscript of instrument definitions
IP(5)	Beginning subscript of location table for instrument definitions
IP(6)	Length of a function
IP(7)	Beginning subscript of blocks of note-parameter storage
IP(8)	Length of a block of note-parameter storage
IP(9)	Number of blocks of note parameters (equals the maximum number of voices that can play simultaneously)
IP(10)	Subscript of unit-generator input–output block which is reserved for storage of samples of the acoustic output waveform. SAMØUT puts out samples from this block
IP(11)	Sound zero. This is integer with decimal point at right end of the word
IP(12)	Scale factor for unit-generator variables (input–outputs, etc.)
IP(13)	Subscript of beginning of unit-generator input–output blocks
IP(14)	Length of a unit-generator input–output block
IP(15)	Scale factor for functions

18. Note Parameters

The word count and parameters P1 through Pn are read by Pass III from a data statement on the input file and are initially put into I(1) and into the P array. If P1 = 1 ≡ NØT, the parameters must be moved to a vacant block of note-parameter storage because other data statements will be read into the P array before the NØT is completed. Note-parameter blocks start at I(n) (n = IP(7)), each block is IP(8) locations long, and IP(9) blocks are available.

A block contains the following arrangement of the information

$$\begin{aligned} I(n) &= P3 \qquad \text{(the instrument number)} \\ I(n+1) &= P2 * IP(12) \\ I(n+2) &= P3 * IP(12) \\ &\vdots \\ I(n+m+1) &= Pm * IP(12) \end{aligned}$$

All subsequent locations to end of block are filled out with zeros. All locations are in fixed-point format. All locations except the first are scaled by the IP(12) scale factor. The first location I(n) contains the instrument number, unscaled. If a block is empty, I(n) contains −1.

When a unit generator is called to calculate part of a note, I(3) = n is set to the first location of the note-parameter block for that note. Consequently note parameter Pk may be found at I(n + k − 1).

19. Instrument Definition

An instrument in Pass III is defined by a *sequence* of data statements, which are read from the input medium. The description is stored in the I(n) array in the instrument definition table.

The format of the input data statements in Pass III is in the following table.

Record #	Word Count	P(1)	P(2)	P(3)	
1	3	2	Action time	Inst No	
2	n	2	Action time	Unit type	$D_1 \ldots D_{n-3}$
3	n	2	Action time	Unit type	$D_1 \ldots$
. . .					
last	2	2	Action time		

The description is terminated by a two-word statement. The quantities D_i specify the various inputs and outputs to the unit generators.

If $D_i < -100$, then $|D_i| - 100$ is a function number
If $-100 \le D_i < 0$, then $|D_i|$ is the number of a unit-generator input–output block
If $1 \le D_i \le 100$, then D_i is a note-parameter number
If $100 < D_i$, then $D_i - 100$ is a variable number

The mnemonic form of instrument definition as written on the score and read by READ1 has already been described in Section 4. Examples are given in Section 5.

The instrument definition is stored starting in I(n) where n = I(IP(5) + Inst number). Instrument definitions are stored in successive locations in I(n) according to their action times. The first unused location in the instrument definition table is kept in I(2). An instrument with a given number may be redefined at any action time. The new definition will be used subsequently. However, no "garbage collection" is done and the old definition will continue to occupy space in I(n).

The format of the description in I(n) is as follows:

I(n)		Type of first unit generator in instrument
I(n + 1)		Pointer to second unit-generator description = m
S_1, S_2, . . . , S_n		Subscripts and parameters pertaining to first unit generator
I(m)		Type of second unit generator
		Pointer to third unit generator
	⋮	
I(r)	0	Terminates description of instrument

The S_i's that specify inputs, outputs, and functions for the unit generators have the following meaning:

If $S_i < 0$, then $|S_i|$ is the subscript in I which specifies the beginning of a function or of a unit generator input–output block.
If $1 \leq S_i \leq 262{,}144$, then S_i is the number of a note-card parameter.
If $262{,}144 < S_i$, then $S_i - 262{,}144$ is the subscript in I of a variable. The number of the variable is $S_i - 262{,}144 - 100$.

20. FØRSAM

FØRSAM is a subroutine that contains unit generators written in FØRTRAN. These may be used either separately or together with SAMGEN which contains unit generators written in basic machine language.

FØRSAM is called in Pass III by the statement

```
CALL FØRSAM
```

The call causes FØRSAM to compute NSAM (= I(5)) samples of the output of unit generator, type J (J is given in the instrument description). Unit-generator types in FØRSAM are numbered 101 and up in order to differentiate them from SAMGEN unit generators, which are numbered 1 through 100.

The block diagram of the program is shown in Fig. 59.

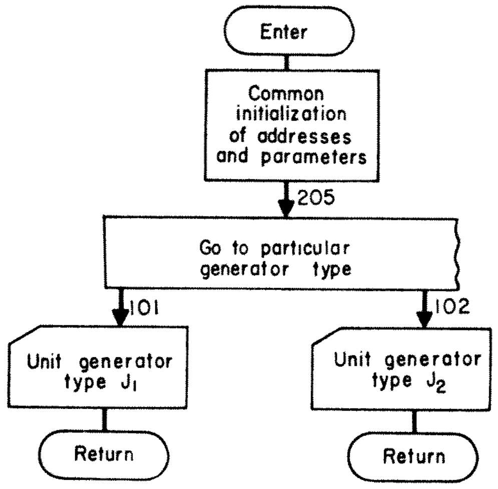

Enter
Common initialization of addresses and parameters
205
Go to particular generator type
101
102
Unit generator type J_1
Unit generator type J_2
Return
Return

Fig. 59. Block diagram of FØRSAM program.

Computation can be made in either fixed or floating-point arithmetic. Usually the scale-factor variables, IP(12) and IP(15), will be necessary to scale the results.

A listing of a small FØRSAM program with only one unit generator is shown below. The initializing routines in the program accommodate other unit generators which can be added to the program simply by extending the GØ TØ at 205 and writing the unit-generator code.

The dimension statement includes three arrays from Pass III, I, P, and IP, and two other arrays, L and M, which are used to address the unit-generator inputs and outputs. L and M are computed by the initialization procedure.

Specifically, the jth input or output will be found at I(m) where m = L(j). M indicates whether an input or output is a single number (note parameter or variable) or a block of numbers (function or I-Ø block). If M(j) = 0, the jth input is a single number; if M(j) = 1, the jth input is a block. For blocks, L(j) gives the subscript of the first number in the block. Inputs and outputs are sequentially numbered. Thus in the data statement

ØSC P5 P6 B2 F1 S ;

P5 is the first, P6 the second, B2 the third, F1 the fourth, and S the fifth. For more convenient referencing, an equivalence is set up so that $L(i) \equiv L_i$ and $M(i) \equiv M_i$.

```
      SUBROUTINEFORSAM
      DIMENSIONI(15000),P(100),IP(20),L(8),M(8)
      COMMONI,P/PARM/IP
      EQUIVALENCE(M1,M(1)),(M2,M(2)),(M3,M(3)),(M4,M(4)),(M5,M(5)),(M6,M
     1(6)),(M7,M(7)),(M8,M(8)),(L1,L(1)),(L2,L(2)),(L3,L(3)),(L4,L(4)),(
     2L5,L(5)),(L6,L(6)),(L7,L(7)),(L8,L(8))
C     COMMON INITIALIZATION OF GENERATORS
      N1=I(6)+2
      N2=I(N1-1)-1
      DO204J1=N1,N2
      J2=J1-N1+1
      IF(I(J1))200,201,201
 200  L(J2)=-I(J1)
      M(J2)=1
      GOTO204
 201  M(J2)=0
      IF(I(J1)-262144)202,202,203
 202  L(J2)=I(J1)+I(3)-1
      GOTO204
 203  L(J2)=I(J1)-262144
 204  CONTINUE
      NSAM=I(5)
      N3=I(N1-2)
      NGEN=  N3 -100
 205  GOTO(101,300,300),NGEN
C     UNIT GENERATOR 101- INTERPOLATING OSCILLATOR
 101  SFU=IP(12)
      SFF=IP(15)
      SFUI=1./SFU
      SFFI=1./SFF
      SFUFI=SFU/SFF
      SUM=FLOAT(I(L5))*SFUI
      IF(M1)210,210,211
 210  AMP=FLOAT(I(L1))*SFUI
 211  IF(M2)212,212,213
 212  FREQ=FLOAT(I(L2))*SFUI
 213  XNFUN=IP(6)-1
      DO223J3=1,NSAM
      J4=INT(SUM)+L4
      FRAC=SUM-AINT(SUM)
 216  F1=FLOAT(I(J4))
      F2=FLOAT(I(J4+1))
 217  F3=F1+(F2-F1)*FRAC
      IF(M2)218,218,219
 218  SUM=SUM+FREQ
      GOTO220
 219  J4=L2+J3-1
      SUM=SUM+FLOAT(I(J4))*SFUI
 220  IF(SUM-XNFUN)215,214,214
 214  SUM=SUM-XNFUN
 215  J5=L3+J3-1
      IF(M1)221,221,222
 221  I(J5)=IFIX(AMP*F3*SFUFI)
      GOTO223
 222  J6=L1+J3-1
      I(J5)=IFIX(FLOAT(I(J6))*F3*SFFI)
 223  CONTINUE
      I(L5)=IFIX(SUM*SFU)
 300  RETURN
      END
```

The number of samples to be generated is put in NSAM. Most of the unit generators will operate with a loop such as DØ 223 J3 = 1, NSAM.

In the computations performed by the unit generator, it is necessary to test to see whether an input is a single number or an I-Ø block. If $M_j = 0$, the jth input need only be obtained once from $I(L_j)$. If the jth input is an I-Ø block ($M_j = 1$), then each value is obtained with the help of the main DØ index J3. For example, the third input is located at I(J5) where

$$J5 = J3 + L3 - 1$$

The particular unit generator is an oscillator that interpolates between adjacent values of the function (see Section 6 for discussion of why interpolation is useful). Computations are carried out in floating-point arithmetic. Since the input data are fixed-point numbers, they must be floated and scaled by appropriate constants. Scale factors for I-Ø blocks and for functions are given in IP(12) and IP(15), respectively. The necessary scaling constants are computed at 101.

21. SAMGEN

SAMGEN is one of the few basic machine language programs in Music V. Consequently it must be written specifically for the particular machine on which it is to be used. The Bell Laboratories program is written in GMAP for a General Electric 635 computer. A few comments about the program may be of use in designing programs for other machines.

SAMGEN includes the unit generators of type numbers less than 100. The computation of the actual acoustic samples, which is the preponderance of the computation in Music V, is done by SAMGEN.

The general form of SAMGEN is shown in Fig. 60.

SAMGEN is written in such a way that one procedure can be used to set the parameters in all of its unit generators. This procedure accesses the I array in common storage during Pass III in order to find out

I(3)	the subscript in the I array of the note parameters for the note being played
I(5)	the number of samples to generate and
I(6)	the subscript in the I array for the instrument definition table of the unit generator being played.

The procedure then reads through the instrument description for the unit generator being played. (See instrument description, Section 19.)

For each unit generator, the procedure expects a certain number of inputs (S_i's) in a certain order, e.g., if unit type = 2 (oscillator), then

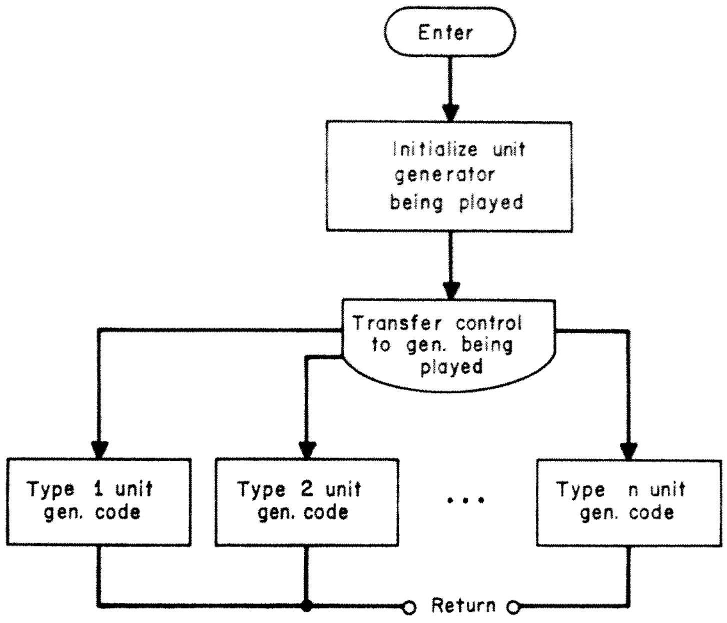


Fig. 60. Block diagram of SAMGEN program.

S_1 = amplitude, S_2 = frequency, S_3 = output, S_4 = function, and S_5 = sum. It then sets addresses in the specified unit generator according to the following conventions:

If $S_1 < 0$, then $|S_1|$ is the subscript in the I array of the beginning of a function or of unit-generator I-Ø block.
If $0 < S_1 < 262{,}144$, then S_1 is the number of a note parameter. Note parameter Px is located at I(I(3) + x − 1). (See Section 18 for more information on note-parameter storage.)
If $S_1 > 262{,}144$, then $S_1 - 262{,}144$ is the subscript in the I array of a variable: variable x is located at I(x + 100).

After the addresses are initialized, SAMGEN transfers control to the specified unit generator, which generates the number of samples specified in I(5).

The calling sequence is

```
CALL SAMGEN
```

Almost all information is supplied by the I array which is located in unlabeled common storage according to the statement

```
CØMMØN I
```

SAMGEN uses no subroutines.

22. SAMØUT

SAMØUT is another GMAP subroutine called by Pass III which (1) scales samples which are ready to be output, and (2) calls FRØUT to output these samples onto magnetic tape. Samples (S_i) are scaled according to

$$S_i = S_i/2^{18} + 2048$$

The calling sequence is

CALL SAMØUT (IARRAY, N)

where IARRAY = address of first sample to be output, and N is the number of samples to be output.

Other routines used by SAMØUT are

FRØUT4

No common storage is used.

23. SAMØUT for Debugging

This version of SAMØUT (cf. Section 22) is provided for debugging purposes only. It is called by Pass III with the call

CALL SAMØUT (IARRAY, N)

in order to *print* out N samples starting from location IARRAY. It must perform the same descaling operations as the normal SAMØUT, i.e.,

$$\text{sample}_i = (\text{sample}_i/2^{18}) + 2048$$

This version of SAMØUT is written in FØRTRAN and will print the sample values in any convenient format. It is recommended that in using this version of SAMØUT one should be careful of excessive output since it is easy to ask for a very large number of acoustic samples.

24. Acoustic-Sample Output Program: FRØUT

The subroutine package FRØUT is called by both Pass III and SAMØUT in order to write the actual acoustic samples generated by Music V onto magnetic tape. FRØUT is coded in assembly language rather than FØRTRAN (1) for efficiency and (2) because it must write special physical records onto tape in a form suitable for digital-to-analog conversion. This is usually not possible in a compiler language such as FØRTRAN.

The exact form of FRØUT will depend on the particular machine configuration of a computer installation. It is therefore necessary that this program be written by an experienced programmer at any computer installation that desires to run Music V.

There follows a general description of the FRØUT programs written at Bell Telephone Laboratories for use with the General Electric GE645 computer. It should act only as a model for such a program written for another machine.

Basically FRØUT simply takes sample values that are produced by Music V, packs several samples into one computer word, and writes them onto magnetic tape in a form suitable for digital-to-analog conversion.

At BTL, the digital-to-analog converters operate with 12-bit samples. Since the GE645 computer is a 36-bit word-length machine, FRØUT packs the acoustic samples three per word.

One packed computer word is of the form

```
┌──────────────── 36 bits ────────────────┐
aaaaaaaaaaaabbbbbbbbbbbbcccccccccccc
└─ sample 1 ─┘└─ sample 2 ─┘└─ sample 3 ─┘
```

Since the maximum integer value that can be represented in 12 bits is 4095_{10}, FRØUT screens the sample values it receives from Music V to be sure that it falls in the range 0 to 4095. Should any samples to be written by FRØUT be outside this range, they are clipped to 0 and 4095.

Pass III first calls FRØUT0 during its initializing sequence with the call

```
CALL FRØUT0 (66, 167)
```

where 66 is a file code (i.e., a logical file name of the tape file onto which packed acoustic samples are to be written), and 167 is the record length in 36-bit words to be written onto this tape (samples per tape record = 3 × words per tape record).

Whenever Music V has produced some samples that are ready to be output, subroutine SAMØUT is called by Pass III, which in turn calls FRØUT with the call

```
CALL FRØUT4 (IA, N)
```

which writes N samples onto tape starting from the location IA.

At the end of the composition, Pass III calls FRØUT with the call

CALL FRØUT3

FRØUT3 completes the output buffer, if it was only partially filled, with zero-voltage samples, empties this last buffer onto tape, and writes an end-of-file mark.

Packing of samples can be accomplished by machine-language shifting instructions and buffering. Acoustic sample tapes typically are unlabeled and unblocked, and use fixed-length records.

FRØUT3 prints a statement giving the number of samples out of range in the file which has just been terminated.

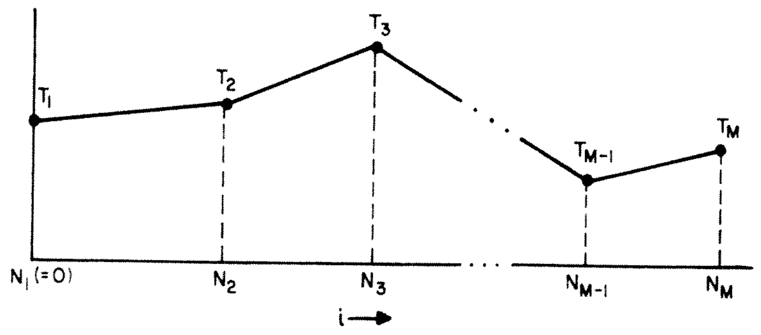
25. GEN—Pass III Function-Generating Subroutines

GEN1

GEN1 is a FØRTRAN subroutine to generate functions composed of segments of straight lines. The calling sequence is

CALL GEN1

Data are supplied by the P(n), I(n), and IP(n) arrays. The jth function $F_j(i)$ is generated according to the form shown in the diagram below.



Linear interpolation is used to generate the function between M points which are specified by the user. Thus between any two abscissa points N_m and N_{m+1} the function points are computed according to the relation

$$F_j(i) = T_m + (T_{m+1} - T_m)\cdot\frac{i - N_m}{N_{m+1} - N_m} \quad i = N_m \ldots (N_{m+1} - 1)$$

The number of corners M is arbitrary and is determined by the word count I(1). M = (I(1) − 4)/2.

In general the user will set $N_1 = 0$ and $N_M = IP(6) - 1$, so that the number of points in the function equals IP(6).

The parameters of the function are arranged as follows:

P(1)	P(2)	P(3)	P(4)	P(5)	P(6)	P(7)	P(8)	...	
3	Action time	1	Function No (j)	T_1	N_1	T_2	N_2	...	N_M

The function is stored starting in I(n) where $n = IP(2) + (j - 1) * IP(6)$ and is scaled by IP(15) so that, for example, $I(n) = T_1 * IP(15)$.

GEN2

GEN2 is a FØRTRAN subroutine to generate a function composed of sums of sinusoids. The calling sequence is

CALL GEN2

Data are supplied by the P(n), I(n), and IP(n) arrays.

The jth function $F_j(i)$ is generated according to the relation

$$F_j(i) = (\text{amp normalizer})\left\{\sum_{k=1}^{N} A_k \sin \frac{2\pi ki}{P-1} + \sum_{k=0}^{M} B_k \cos \frac{2\pi ki}{P-1}\right\} \quad i = 0 \ldots P-1$$

P (= IP(6)) is the number of samples in a function.

The parameters for the function are arranged as follows:

P(1)	P(2)	P(3)	P(4)	P(5)	...				P(−)	P(−)
3	Action time	2	Function No (j)	A_1	... A_N	B_0	...	B_M		± N

The number of sine terms is $|N|$. If N is positive, amp normalizer is computed so max $|F_j(i)| = .99999$. If N is negative, amp normalizer = .99999. The number of cosine terms M is computed from N and the word count I(1). $M = I(1) - N - 5$.

The number of samples in the function is IP(6).

The function is stored starting in I(n), and is scaled by IP(5)

$I(n) = IP(15) * F_j(0)$, etc.

where $n = IP(2) + (j - 1) * IP(6)$.

The *first* and *last* samples of the function are *equal*, $F_j(0) = F_j(P - 1)$, thus the period in samples is $P - 1$.

GEN3

General description:

GEN3 is a FØRTRAN subroutine which generates a stored function according to a list of integers of arbitrary length. These integers specify the relative amplitude at equally spaced points along a continuous periodic function. The first and last points are considered to be the same when the function is used periodically (e.g., by an oscillator).

Calling sequence:

CALL GEN3

Other routines used by GEN3:

none

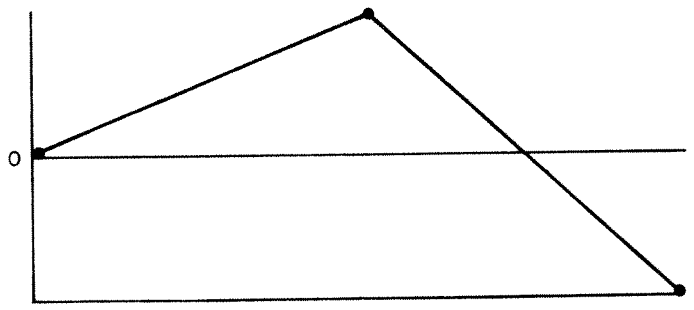
Data statement:

GEN, action time, 3, stored function number, $P_1, P_2, \ldots, P_{nj}$

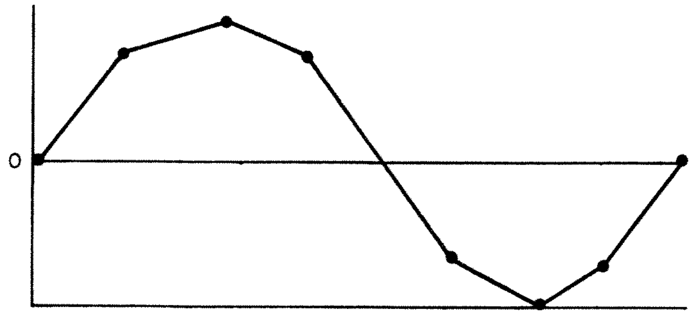
Examples:

The following P_i's will generate the functions shown below.

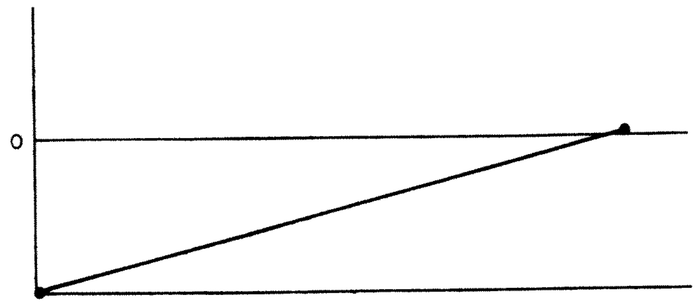
(1) 0, 1, −1
will generate



(2) 0, 8, 10, 8, 0,
−8, −10, 0:



(3) −1000, 0:



26. Data Input for Pass III—DATA

Subroutine DATA is called by Pass III with the call

```
CALL DATA
```

This causes one data statement to be read from file 11 into the P array in CØMMØN storage according to

```
READ (11) K, (P(J), J = 1, K)
```

I(1) is set equal to K (word count).

Annotated References by Subject

Music IV Program

M. V. Mathews, "The Digital Computer as a Musical Instrument," *Science*, *142*, 553–557 (November 1963). A semitechnical description of Music IV with some discussion of applications. This is a good introductory article.

M. V. Mathews, "An Acoustic Compiler for Music and Psychological Stimuli," *Bell Sys. Tech. J. 40*, 677–694 (May 1961). A technical description of an early version of a sound generating program. This is the first complete published description.

J. R. Pierce, M. V. Mathews, and J. C. Risset, "Further Experiments on the Use of the Computer in Connection with Music," *Gravesaner Blätter*, No. 27/28, 92–97 (November 1965). A semitechnical description emphasizing applications of Music IV. This is a good follow-up for the paper in *Science*.

J. C. Tenney, "Sound Generation by Means of a Digital Computer," *J. Music Theory*, *7*, 25–70 (1963). A discussion of Music IV as seen by a composer using the program. The article contains many details and is a good introduction for a musician.