MUSCIL
AN AUTOMATED MUSIC COMPOSITION DATA PROCESSOR

by

Jean Piché

B.A.C.C., Université Laval, 1974

PROJECT SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

M.A. (CMNS)

in the Department

of

COMMUNICATION

SIMON FRASER UNIVERSITY

SEPTEMBER 1981

APPROVAL

Name: Jean Piché

Degree: Master of Arts (Communication)

Title of Project: MUSCIL: An Automated Music Composition Data Processor.

Examining Committee:

Barry D. Truax
Assistant Professor
Senior Supervisor

Nick J. Cercone
Associate Professor
Computing Science

Jerry Barrenholtz
Lecturer in
Computing Science/Centre for the Arts

Martin Bartlett
Associate Professor
Faculty of Music
University of Victoria
External Examiner

Date Approved: September 29, 1981

Title of Thesis/Project/Extended Essay

MUSCIL: An Automated Music Compositon Data Processor.

Author: [illegible]
(signature)

Jean Piche
(name)

September 29, 1981
(date)

ABSTRACT

MUSCIL is a new language developed by the author for the input of musical scores in a computer-based music system.

The context of computer music composition is analysed in terms of a behavioural process where a number of specific tasks need to be accomplished for the production of music. The three main tasks are defined as the building of instruments, the composition of scores, and the organization of scores together in a final structure. Each of these tasks is analysed and conclusions are drawn as to their relative importance within a given system.

Based on these observations, and on a review of various computer-based music score editors, proposals are made for the elaboration of MUSCIL.

The MUSCIL language for music composition data processing is presented in the form of a user manual which explains the data structure and syntax of the language, and gives examples on the use of its features. The language is presently implemented in PASCAL and runs on Simon Fraser University's main computer.

THE MUSCIL PROJECT IS DEDICATED TO JEROME BARENHOLTZ
WHO INTRODUCED ME TO THE DELICATE ART OF PROGRAMMING AND
HAS SHOWN CONSTANT ENTHUSIASM FOR MY WORK.

ACKNOWLEDGEMENTS

The author wishes to thank Barry Truax and Nick Cercone for taking the time to read the manuscript, make suggestions and give essential criticism during the elaboration of the project. The author also acknowledges Donna Zapf, Hildegaard Westerkamp , Wayne Carr, Morrie Eaman, Ron Franklin, Frank Campbell, Dr. Tom Calvert, Susan Penner and Dr. William Richards for essential technical and emotional support.

TABLE OF CONTENTS

Approval........ii

ABSTRACT........iii

DEDICATION........iv

ACKNOWLEDGEMENTS........v

TABLE OF CONTENTS........vi

LIST OF FIGURES........viii

PREFACE........ix

A. CONSIDERATIONS ON THE MAKING OF MUSIC WITH COMPUTERS........1

I. DEFINING COMPOSITION........2

THE CONCEPT OF INSTRUMENT........4

THE CONCEPT OF SCORE........10

THE CONCEPT OF ORCHESTRA........14

II. A BRIEF SURVEY OF CURRENT MUSIC SCORE EDITORS........16

THE MUSIC V SYSTEMS........16

SCORE........18

4CED........19

THE POD SYSTEM........21

SSSP........22

III. PROPOSAL FOR AN INTERACTIVE REAL-TIME MUSIC DATA PROCESSOR........25

AN INSTRUMENTAL MODEL........27

INTERACTION........33

B. MUSCIL : USER'S MANUAL........35

I. THE MUSCIL WORKING ENVIRONMENT........36

MUSCIL LIBRARIES 39

INTERACTIVE COMMANDS 41

MUSCIL DATA STRUCTURE 44

II. THE HEADER 48

PITCH/TIME SCORES 49

LISTS 52

MASKS 54

III. THE PARAMETRIC COMMAND LIST 56

MUSCIL PARAMETRIC COMMAND LANGUAGE 61

IV. THE ORCHESTRAL COMMAND LIST 79

THE ORCHESTRAL COMMAND LANGUAGE 81

APPENDIX I 85

A SAMPLE MUSCIL USER'S FILE 85

APPENDIX II 92

A PASCAL IMPLEMENTATION OF MUSCIL 92

BIBLIOGRAPHY 167

LIST OF FIGURES

FIG.1....A MUSIC V INSTRUMENT USING 15 PARAMETERS......5

FIG.2....A MUSIC V NOTE TABLE..........................11

FIG.3....MUSCIL DATA STORES AND WORKING ENV...........38

FIG.4....ACTIVE LIBRARY STRUCTURE......................45

FIG.5....THE OCL DATA STRUCTURE........................94

PREFACE

The use of digital computers for the composition and generation of music has progressed greatly since the early experiments conducted by Lejaren Hiller(Hiller,1958) and Max Mathews (Mathews, 1969) in the early fifties. The availability of newer and faster digital computers brought the emergence of a new and very compelling way of producing music. Perhaps the very nature of music as a special application of physics and mathematics best explains the convincing results so far obtained in combining automated data processing and musical creativity.

For centuries music and numbers have followed a parallel path. The workings of Pythagorean arithmetics were partly based on proportions of vibrating string lengths in relationship to their sounding pitch. Principles of vibratory physics were initially formulated by the French physicist Fourier by examining sound as a compound of multiple harmonically related vibrations.

The more specific task of musical composition also involves processes closely related to mathematics. The Art of the Fugue illustrates the wonderous fascination J.S.Bach cultivated for the magic of numbers. More recently, from the time of Schoenberg's radical departure from tonal pitch theory, one can observe the almost overbearing presence of "data processing" as the main compositional concern of modern music practitioners.

While the extensively rigid rules in serial composition resulted in its downfall and dissaffection by most composers, perhaps the most influencial composers of our time (Boulez, Xenakis, Stockhausen and John Cage) have and in some cases still do define their craft as a selection process from all musical "possibles" by the application of compositional rules : the act of composing is an act of computing (Risset, 1980). The advent of the digital computer, and more generally of electronics, probably constitutes the most significant musical development in centuries. Very often in the history of music a given "golden period" can be linked to the arrival of a particularly apt tool. The pianoforte by virtue of its dynamic keyboard opened a fruitful venue to the composer of the late 18th century and its refinements have given us the expressive richness of romantic music.

The impact of computers will probably be deeper. For the first time now, the composer is given access to the generalized instrument. Or to put it in its true perspective, the generalized orchestra. Significantly the composer is also given the option to design his own personal rules of composition into a computer-based working environment.

A composer's task, in the traditional setting of instrumental composition usually consists of choosing instruments and writing scores for them. In a computer music-making environment, however, these tasks are not obvious.

Concepts of instruments and scores can still be used; but these become generalized in the sense that any instrument can be fabricated by the composer and can be made to play any score (Mathews, 1969). Digital synthesis procedures represent in fact a virtual instrument that can theoretically perform the tasks of any known acoustic instrument and offer entirely new instruments that don't have an equivalent in the acoustic world. The composer no longer merely chooses instruments but actually fabricates them by determining special computing procedures.

Perhaps more relevant to this presentation is the new sense of the act of composition. The writing of notes on musical staves, the testing on a keyboard, the erasure and recomposition of segments have successfully translated the genius of generations of composers and before the invention of automated data processing no other method of notating musical thought could have been used. In the current state of the software, one could discuss advantages and inconveniences of composing at the terminal as opposed to composing at the piano.

More than a few composers set in the traditional ways have cursed the computer as a form of sub-intelligence not always facilitating the composition of music. They see the composer as consistently being bent into thinking about music the same way the designer of the program was thinking. This problem is regularly experienced by composers who use computers in the making of their music. To a large extent an experience of this type is unavoidable to the uninitiated composer. The design and implementation of computer music software is an exercise in decision-making about what is important, most important and not so important. No two composers can come up with the same priority list. For this reason the final arbitrator in the usefulness of a computer-music program is the composer who uses it. When a composer is not satisfied, he designs his own programs. MUSCIL was designed by the author to implement first and foremost his own set of priorities in the exercise of his craft. The relevance of the solutions offered may or may not suit the needs of a composer with specialized demands. MUSCIL is not in that respect a program by a programmer, but a program by a musician. It is offered as a musical tocl to be used by the advanced composer who is comfortable with the compositional model on which it is based.

A. CONSIDERATIONS ON THE MAKING OF MUSIC WITH COMPUTERS

I. DEFINING COMPOSITION

The activity of music composition can be handily dissected into a number of subtasks.

1. The first of these tasks is to decide which instruments or which type of instruments will be used in the composition. Given a computer-based sound synthesis facility, the composer's work in selecting his palette of timbres will be much more complex than the mere choosing of available acoustic instruments. The composer will fabricate instruments and test them to familiarize himself with its general properties. Some attention will be given to what kinds of musical statements will be made with these instruments, but at this level the "sound" aspect is most relevant.

2. Once a group of sound-producing devices has been assembled the composer will compose scores for them. At this point our definition of score is very open. Let us simply assume that the composer intends to use a specific instrument to produce more than one sound. In other words the instruments will play something.

   As opposed to acoustic instruments, the scoring for digitally implemented instruments is more precise and more general. A score can be made to control every parameter of an instrument whereas acoustical instruments are usually given a score of pitch against time with some indications for a few of the secondary parameters, such as loudness and articulation.

3. When scores have been written for a number of instruments, the composer will likely want to hear theses scores - by themselves or playing together with one or a number of other scores. This task is accomplished by organising individual scores polyphonically and represents the most advanced stage of structural organization where the work's final form and shape is devised.

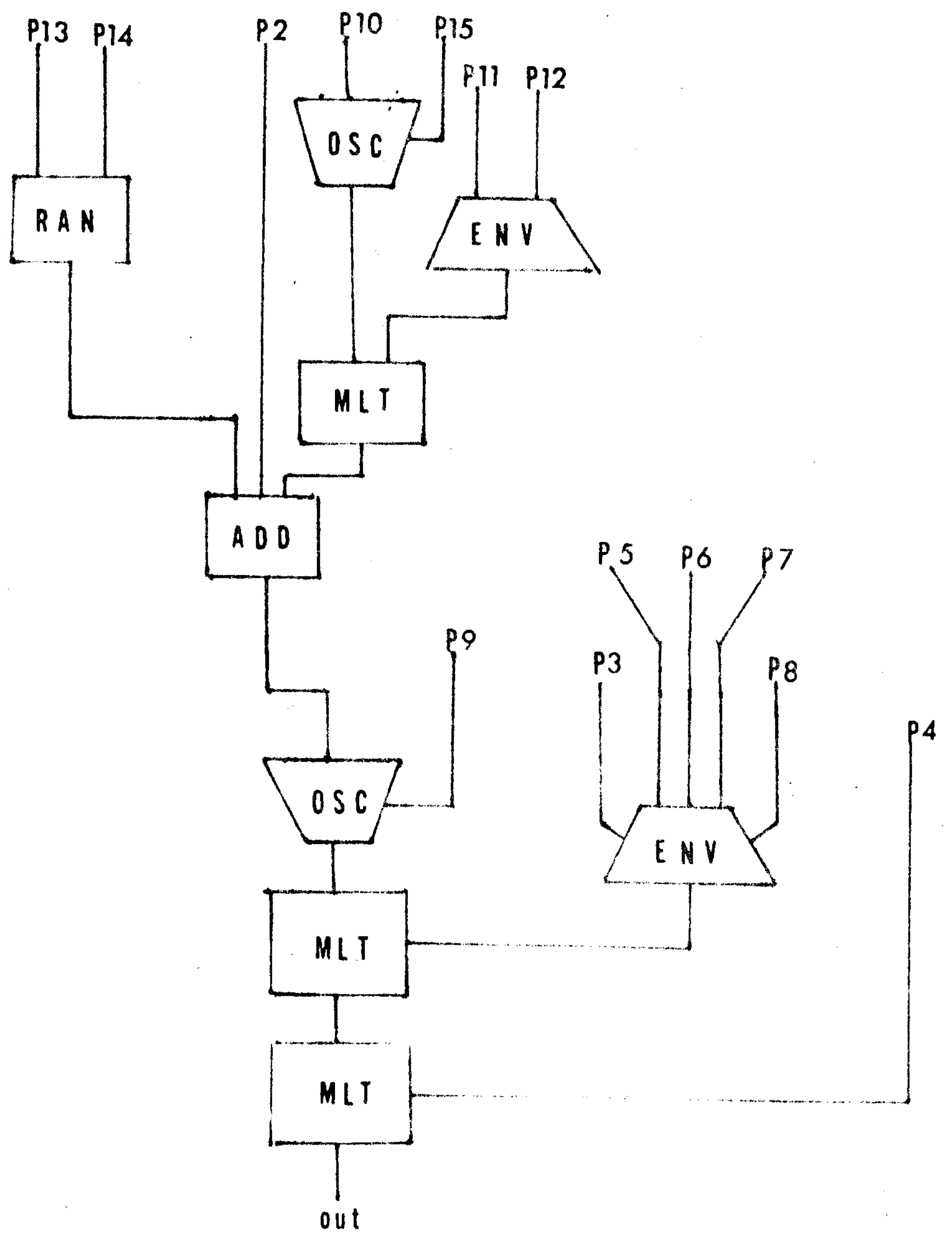
The production of music, as we have discussed briefly in our introduction, is subject to influences by the very means of production. If a procedure, musical or not, can be described with symbolic logic, computers can offer an implementation of the procedure. The problem with musical tasks is one of aesthetics. Of all possible musics, none can be said to be more desirable than the other. The very nature of experimentation in music or otherwise demands that experimentation tools be able to measure, synthesize, and analyse the unknown. This can only be achieved through the implementation of open systems based on extremely general models.

However, generality rapidly becomes its own antithesis. Full cptions remain open only as long as we do not decide that some things are more important than other things (Truax, 1977). Many decisions were taken in the course of MUSCIL's elaboration to reduce the generality of music-making as an open-ended activity. This was done by applying specific models of compositional behaviour and of musical acoustics in the design of its main features. The models adopted, like the one described above, are widely, if not universally, accepted.

THE CONCEPT OF INSTRUMENT

For our purpose let us define the computer-implemented instrument as a virtual system capable of producing sound. These systems are defined by the types of sounds they produce and by the algorithms utilized to generate their timbre. Programs are currently available to assemble many types of algorithms. Such programs are MUSIC V (Mathews), MUSIC 360 and MUSIC 11(Vercoe) and SAMINS(CCRMA-Stanford). Since this project does not directly deal with techniques of instrument-building with computers, we will simply adopt the most wide-spread of these programs as a model for our own purpose of defining what an instrument consists of. MUSIC V provides a very elegant way of making instruments and most other programs were derived from it.

Fig. 1 — MUSIC V INSTRUMENT WITH 15 INPUT PARAMETERS



INSTRUMENT-BUILDING IN MUSIC V

We shall use the MUSIC V model (Mathews, 1969)in the following discussions as a basic structural reference for the elaboration of the MUSCIL input language. In MUSIC V a number of unit generators are offered. These units usually have a set of inputs and an output and can be connected together by simply listing calls to different units. The composer organizes different units together to obtain a network of signal-processors that have the potential of producing sound. The characteristics of the sounds they can produce depends on the types and number of units used, the networking of the connections between the units and the values of the parameters left open at the input points of the network.

A typical instrument might use a function generator, a modulating oscillator, a multiplier, an envelope generator and a random generator for a vibrato for example.

The diagram shown in fig. 1 is such a Music V instrument using nine unit generators. The RANdom generator uses two variable input parameters : P13 and P14. The function delivered by the RAN unit is added to the signal obtained by a first modulating oscillator whose pitch is obtained from P10 and whose amplitude is obtained through a multiplier controlled by an envelope generator with the function described by P11 and P12.

The waveshape of the modulating oscillator is specified by P15. The adder also received P2 which describes the pitch of the main oscillator. The sum of the adder is used to control the frequency of the oscillator. The main oscillator also knows the kind of waveform to output by parameter P9. Finally the instrument is given an amplitude contour with unit generator ENVelope, where P3, P5,P6,P7 and P8 give the attack time, initial decay, steady state and final decay plus a level for the steady state portion of the envelope function. The outputs of the ENV and the OSC are multiplied and sent to a second multiplier to scale the sound to a volume specified in P4. This instrument is strictly a signal-processing network and as such does not produce sound. All the "P" points need to be known for the instrument to produce sound. Each "P" point will provide essential information with regards to pitch, amplitude, timbre and duration of the sound. This information must be supplied for each and every sound the instrument is to produce.

A DISCUSSION ON THE HIERARCHY OF INSTRUMENT PARAMETERS

Over the years many different algorithms for instrument-building have been proposed. John Chowning's frequency-modulation technique (Chowning, 1973) and the linear distortion waveshaper of Mark LeBrun and Daniel Arfib(LeBrun, 1979 & Arfib, 1978) are pre-defined networks that have proven useful in the generation of rich families of sounds with relatively few control parameters. This reduction of control parameters will probably prove the most long-lasting advantage of powerful synthesis methods as opposed to the specific timbres they produce.

Most composers see a natural hierarchy in the parameters that define the syntactic value of a sound in relationship to other sounds. Pitch is always favoured, whether organized tonally or atonally. The succession of sounds in time is another primary parameter. Although this parameter is not part of any single instrument definition, it is present whenever we ask the instrument to produce more than one sound : a timing period must be known before a second sound is produced. This timing parameter allows the construction of rhythm, and of all time-related structures.

If, as above, we use the model of acoustic composition, we can discern a few secondary parameters like amplitude articulation and timbre. In a succession of sounds, a secondary parameter is one that is often found to be dependent or a function of the primary parameters, or which does not carry the same syntactical weight as the primary parameters. Most timbral parameters are usually in this group.

The actual duration of each sound will mostly be a function of the time available between each sound. This is always true in the case of monophonic instruments that must end one sound before they can produce a new one. The amplitude of a sound may be a function of the pitch a sound, as in almost all blown instruments like woodwind and brass. The higher pitches are louder.

Changes in loudness between sounds are usually done in broad gestures like a crescendo, a diminuendo, or a given accentuation pattern.

Certain timbral characteristics of the amplitude contour of sound often depend on pitch and loudness. The timbral and temporal character of percussive sounds, for example, will vary greatly in relationship to their loudness.

This discussion aims at understanding that when a sound is heard as the product of a device, the character of the sound is a result of the particular idiosyncracies of the system that produces it. The sonic manifestations of a vibrating or oscillating body are always reflective of the structure of the instrument and their character is the making of a physical system. In digital sound synthesis, this particular individuality of acoustic instruments is not easy to reproduce. No one amplitude, pitch or timbre parameter should be thought of as static. Credible synthesis of tones can only be achieved through a model that respects the inherent interdependency of parameters involved in the production of a musical tone.

As will be discussed later, these observations led to major decisions in the conceptualization of MUSCIL. This approach is also very relevant to reduction of data in the formulation of scores of parameter values to be performed by an instrument.

THE CONCEPT OF SCORE

Scores are defined as lists of parameter vectors needed by a given instrument to produce a succession of different sounds. The actual meaning or application of the parameters is not relevant as long as the first parameter in each vector is a waiting period before the next vector is performed by the instrument. The other parameter values in each vector will tell the instrument what sound to produce.

In traditional acoustic scoring the interpreter is asked to perform the given pitch(es) at the loudness prescribed with the waiting period(s) proportional to the duration of the note(s). In a practical sense a human performer can only be expected to process a limited amount of information regarding the sounds he is to produce with his instrument. This has been eloquently illustrated by the famous Klavierstucken by Karlheinz Stockhausen. Intended to explore the very precise parametric control of the generalized series, the pieces were in fact impossible to play as scored.

Computers brought with them not only an amazing capacity to synthesize sound, but also a disconcerting ease to perform the most complicated music devisable. Not only does the computer process instantaneous information about pitch, amplitude and timbre, but it can also control other parameters with great precision.

While the precision of control over all parameters is greatly enhanced, the amount of control information to be supplied to the instrument is also increased. The burden is on the composer to elaborate very complex tables of numbers to have the instruments produce music.

P1,P2,P3,P4,P5,P6,P7,P8,P9,P10,P11,P12,P13,P14 and P15 of our example must all be supplied for each and every sound. Fig. 2 shows one such table.

TIME	P1	P2	P3	P5	P6	P7	...P15
1.2	12.1	500	440.4	2.5	22	330	...11
2.4	8.1	300	880.8	3.2	24	660	...9
3.2	11.4	100	220.2	4.7	29	880	...7.2
4.4	10.1	50	330.4	3.3	20	300	...5.1
.							
.							
.							
6.6	11	77	220.	12.1	22	400	...2.2

....Fig. 2 - A MUSIC V NOTE TABLE....

A data table like this is a score. It is a list of control information and timers to be used one line at a time by an instrument with 15 parameters to produce one sound, and then wait the amount of time given in the first parameter before producing the next sound. The compilation and generation of control tables or scores of this type through a powerful high-level language is the first goal of the MUSCIL project.

If a composer must input scores one number at a time, he often faces a burden too complex to deal with efficiently. Most of the information supplied by the composer in this case is redundant, specially if many of the parameters are a known function of two or three primary ones.

We should also give consideration to the types of data processes used by composers in scoring for an instrument. A considerable amount of redundancy is usually present in a given musical statement. Daniel Charles, the noted French musicologist, defines music as the dialectic experience of change within permanence (Charles, 1976). A group of pitch values or a succession of durations will often be repeated within the same work; in the case of minimal music the same musical phrase is often repeated continuously with small variations. If a succession of parameter values is to be used consistently in a work, a composer should be given the means of describing the succession only once and referring to it subsequently by a name or a mnemonic reference.

In the early fifties, Iannis Xenakis formulated and demonstrated a pertinent principle for the generation of parameter values (Xenakis, 1963). The post-Webernian composers were dedicating enormous energies to calculating hyper-precise scores while never considering that the actual sounding result was so complex that the meaning of the individual events in a composition was lost. Xenakis stated that random generation procedures would much more efficiently generate the same kind of perceived structures. The use of random (stochastic) distribution methods in music is now wide-spread. The composer using them can specify probabilistic ranges for parameters of sounds instead of defining the precise value of each parameter for each sound event.

While experimentation with stochastic procedures as a style has probably shown its limits for the construction of scores, the use of random procedures to produce credible timbres cannot be overestimated. This particularity is mainly concerned with the micro-time structure of sound itself but it is also reflected in formal structures by the undefinable "human" edge of live music performance.

Randomization is in fact vital in the simulation of nature. Perhaps the most compelling power of precision in computer-based processing is also its killing grace. Computer music that does not acknowledge imperfection usually suffers from an unrelenting hard-edge.

MUSCIL is a language to generate control data tables, using algorithms to implement random and sequential processes easily that are often used by composers, thereby reducing redundant input. Parametric interdependency allows the formulation of scores in a hierarchical fashion.

THE CONCEPT OF ORCHESTRA

The last step in the production of complete musical works is usually the orchestration stage. Several ways of looking at orchestration are possible. One definition states that orchestration is the process (the "scorchestration" concept) by which a composer assigns different scores to be played by various instruments (Buxton et al., 1978).

Another definition, which is adopted by MUSCIL, states that orchestration is the organization of various instruments playing different parts into one polyphonic 'score'. In this task, the composer proceeds to the macro-assembly of individual parts scored for different instruments. This concept of orchestra in a computer setting is an expanded one, owing to the almost unlimited variety of instruments devisable with digital synthesis techniques and the theoretically endless variations possible in their juxtaposition and combination.

An analysis of most musics shows that there always are redundant elements in a work of music. The same instrument may for example appear at a later stage of a work playing the same score with variations in a number of parameters, e.g. softer, transposed by a fifth or simply faster. Such operations can be referred to as post-processing of scored data. These processes are most often employed in the general structuring of music.

An orchestra can be regarded as an open system capable of producing a variety of vertical structures. The number of variations possible is limited by the number of instruments comprised in the orchestra and the number of different scores playable by these instruments. The importance of orchestration and global structuring should not be underestimated. A final work of music will rarely consist of a lone line of sound. The composer must be given tools that ease the process of scoring many voices at the same time. He must also be given tools at this stage for the implementation of variations on the individual instruments he uses without having to rescore a full part.

MUSCIL makes this orchestral level of composition activity the most important focal point.

II. A BRIEF SURVEY OF CURRENT MUSIC SCORE EDITORS

In this chapter we shall look at a number of computer-based music score editors. In doing so, we will point out their advantages and inconveniences with the intent of integrating some of the more useful features into the design of our language. The choices made here reflect the author's experience with these systems without considering the intent of their authors. As was previously noted, the real test of any computer music system comes when users are trying to generate music. The features of systems highlighted as desirable reflect nothing more than this author's biases in accomplishing the tasks pertinent to music-making.

<u>THE MUSIC V SYSTEMS</u>

In the description of the MUSIC V instrument we mentioned its very crude score editing process. The overwhelming task of specifying scores in MUSIC V prompted several composers to develop new versions of the program by adding badly needed tools to facilitate input.

The MUSIC 360 and MUSIC 11 languages devised by Barry Vercoe at M.I.T. resort to graphic means for inputing notes on staves. These pitch-time structures can then be applied to a predefined instrument with the other parameters being input at the instrument-building stage. This approach suggests a number of comments on the use of graphics hardware for the input of scores. It also brings comments on the relevance of various "visualizations" of scored music.

Graphics are expensive, in both hardware and software. If they came for free there is no doubt that they would be useful. Computer systems should be used in an efficient fashion, without wasting computing power on the implementation of features that, although impressive, have never proved to be of crucial importance in the making of good music. Until the time comes that proper, human-engineered interfaces can be provided at little cost, there seems no reason for the composer to avoid dealing with numbers. It is faster, more efficient and inexpensive.

Another comment, related to the previous one, challenges the conviction on the part of music-program designers that composers cannot understand anything but notes on music staves. Not only does this attitude take a heavy toll on experimentation in scoring, but also limits precise scoring of every parameter of an instrument and practicaly eliminates one of the finer edges of computer music as a powerful "organizer" of compositions.

Common Music Notation (CMN) has its uses for the computer-music neophyte. It is a practical representation of two primary parameters of music, pitch and time, but it places an unnecessary bias on what the composer ought to be thinking of as important in his music.

SCORE

Many features of SCORE, developed by Leland Smith at Stanford University, were adopted in MUSCIL. In the author's experience SCORE takes a very sensible approach to practical input procedures and at the same time provides very powerful tools for the experimenter (Smith, 1972).

SCORE was designed to work in conjunction with MUS10, a MUSIC V type instrument definition program that leaves a number of parameters open for control during the execution of a score. Each "part" of a composition is specified one parameter at a time using a number of operations that will apply for the duration of the score. Parameter values can be defined by inputing lists of data or random number generator coordinates. A mathematical language is offered that applies to any parameter to implement parameter interdependence efficiently.

SCORE compiles a score from a user's compositional file. That file is input according to the SCORE syntax with a given computer text editor. SCORE does not interact with the user and simply compiles the total score for all the instruments present in a file. There are no "nesting" possibilities. Everything pertaining to the execution of an instrument score must be present in the parameter command list of that instrument. Once a file has been defined, one cannot refer to other files and, for example, produce the simultaneous execution of many files. SCORE also lacks in that the composer cannot pre-compose "cells" of data and later use them in his instrument scores.

The last criticism of SCORE is essentially its "batch" processing approach. SCORE does not run in real-time nor was it designed to. This rules out any possibility of SCORE being used with input devices like keyboards or potentiometers to interpret or conduct the execution of a score in real-time.

4CED

Curtis Abbott programmed the 4CED language to control the 4C synthesizer built by Pepino di Giugno at IRCAM in France. The output of 4CED feeds the input registers of the 4C synthesizer with parametric data every time a new sound is to be produced, according to the score worked out with the 4CED editor (Abbott, 1980).

4CED is a real-time score processor and instrument-builder that incorporates a MUSIC V type instrument definition, a function editor for envelopes, and a powerful score processing language. Data for individual scores must still be input in the form of note lists with all parameter values actually defined.

These scores can be cycled and treated in a number of ways through the processing part of 4CED. Since 4CED feeds the 4C in real-time, a total reconfiguration of the synthesizer is allowed "on the fly" as is the input of parameter values through a set of potentiometers that are read by the program just before a sound is output. This is by far the most interesting feature of 4CED. It is essentially an extremely powerful sequencer that processes many scores at the same time, offering a number of options at the performance level.

4CED tries to implement a wide variety of musical operations on a small computer(PDP 11/34). That it succeeds in doing so is remarkable. Its important lack is in the type of primary score definition used. It is very similar to a MUSIC V input file and as such is very awkward. Much typing is needed if one wants to specify large deterministic pitch-time scores.

The "orchestral" language of 4CED is its most interesting feature. Individual scores are seen as "units" that will cycle for a time given in a list of real-time command. At any point in the execution of a command list, the scores can change, and specific parameters can be changed. Some features of this part of 4CED were adopted in MUSCIL.

THE POD SYSTEM

The POD programs were developed by Barry Truax to run on mini-computers while providing an interactive setting for the user (Truax, 1977). POD6 is centered around a conversational program to aid the user through a number of musical processes used in defining a specific type of composition. The instrument offered by POD for synthesis is a frequency-modulation algorithm with a fixed number of parameters. The parameters pertaining to timbre are defined through "object definition" procedures and the score consists of a pitch-time-amplitude conglomerate calculated from a Poisson distribution and user-supplied limits. An auxiliary score editing facility allows a more general control of the main parameters by using a number of operators on existing score data files.

POD programs are not readily amenable to the composer with a bias towards deterministic procedures in composition. POD's important feature is probably its stress on interaction as a tool for letting the composer define his own priorities. That however is true of the main part of the system(POD6). In POD6, the composer organizes a monophonic distribution of events(notes) by specifying its structural features. POD offers programs for editing, mixing and orchestrating different scores. These programs run on precompiled POD6 scores. At this level, the advantages of interaction are lost to the profit of generality. Since most musics usually consist of more than one line of sound, it is vital that the final goal of a music program always be a polyphonic score.

The three main steps defined earlier in our compositional model are present in POD. While POD6 was conceived as a full compositional facility it is in fact seldom used for the composition of an entire work. The three main tasks of instrument-definition, scoring (if one does not wish to use Poisson Distributions) and orchestral structuring are in fact implemented in the form of three different programs(POD6, PDFIL and MERGE).

The present author was influenced by this structuring of tasks.

SSSP

The Structured Sound Synthesis Project, coordinated by Bill Buxton at the University of Toronto, has concentrated of lot of energy in the creation of appropriate software for the input of music scores. Supported by some of the most impressive graphics available, most input is done with a light pen on a variety of pitch -time coordinates like "piano-roll" notation or common music notation on staves.

The criticism given earlier to graphics oriented input tools apply just as well in this case. In this particular example the cost implied in graphics is specially felt owing to the relatively small amount of computing power available for the full system.

A number of relevant issues pertaining to computer-aided composition were addressed in the development stage of the SSSP system (Buxton,1978). Buxton correctly assessed that composition is essentially a recursive process whose different steps are rarely exclusive. He further observed that very few score editors allowed the composer to work with small units and later use instances of these in the conception of larger compositional units. This hierarchical concept of scoring is fundamental if one is to take advantage of the true "assembling" capacities of a computer based resource.

SSSP implements this concept with tree structures where a recurring unit in a composition need not be duplicated in actual storage. SSSP uses the event as its primary unit. This event can be of various levels of complexity (e.g. a score, a note, a section) and a score is defined as a number of events.

Although the nesting approach is very powerful, there seems to be limits to its practicality. It would be important to define how many levels of data nesting are relevant to music composition. The composer can easily get lost in tree structures that involve more than a few dimensions. Perhaps more important is the observation that if music repeats itself, it rarely repeats itself perfectly. A nesting approach to compositional data structures should therefore advocate the use of powerful parametric modifiers to facilitate the scoring of "variations" of nested data.

SSSP also assumes primary parameters as the exclusive domain of pitch and time. Scores are strictly defined with four parameters (pitch,time,amplitude and timbre). Timbre is specified through a call to an independent data structure, known as the "object". Parameters relevant to timbre (envelopes, spectra, etc.) are defined once in a separate program. When a score is performed these parameters are not accessible. As we have seen in the preceding chapter, this approach does not lead to a "credible" morphology of musical statements. Parametric interdependence must be reflected all the way down the parametric chain of a given instrument for it to achieve critical realism. The notion of hierarchy in parameters is acknowledged, but some of the most important secondary and tertiary parameters are totally separated.

One development of the SSSP project worth mentioning is the conducting programs of Bill Buxton. Similar to 4CED in their scope, they allow the simultaneous performance of a number of scores and the real-time interaction of the "performer-conductor"(Buxton et al., 1980).

III. PROPOSAL FOR AN INTERACTIVE REAL-TIME MUSIC DATA PROCESSOR

Many composers have turned to computers as a way of generating music for many years and in many cases the results obtained would not have been possible without their use. Very often composers with a particular musical idea resort to writing specific programs to calculate data from which scores are derived (Xenakis, 1971 & Koenig, 1975). No particular program for music-making has ever succeeded in offering all composers the possibilities for elaborating all possible music structures.

The scoring of music is nothing more than a special kind of data processing and the possible musics derived from all possible sets of data is beyond reach of the imagination.

The design and implementation of the MUSCIL language has gained from the criticism given to currently available music score editors and, in some cases, has borrowed directly from them. The author wishes to acknowledge their contribution.

The goal of the MUSCIL project was to establish a language that retains a measure of familiarity for the composer acquainted with other music score editors like SCORE or MUSIC V. At the same time MUSCIL is to offer greatly enhanced features to ease the production of complex musical scores. It is a difficult task for the designer to formulate languages that are very general in the scope of problems they can treat, and yet which allow the execution of these tasks in a relatively "natural" way.

In elaborating a language, the first step is to describe the types of problems we wish to solve with that language. A music processor in itself can be relatively simple, as we have seen with the MUSIC V note tables. Theoretically the composer could input any type of musical structure into a computer and have it performed. However this process is tedious. The level of redundancy is usually so high in almost all musics that their actual representation could be reduced to a few base elements and a good construction plan (music in "kit" form). The actual building can be automated.

The ultimate legitimacy of a score processor is that it saves a lot of work. The other main advantage is that it allows the composer to concentrate the bulk of his work at the level which suits him the most. Any experienced composer can formulate "personal" or "generalized" sets of rules in the composition of music. These rules, when formulated, can be implemented in the form of an intelligent score processor, where the entry of data is minimized as a primary task and more attention is given to what most composers actually do when they compose music.

MUSCIL does not attempt to be the most general of score processors. The compositional behaviour model adopted as a basis to the design of MUSCIL is specific. The "instrumental" approach, adopted by MUSCIL, has been time-tested by generations of composers. The "instrumental" process of MUSCIL, however, is greatly expanded and suits a number of alternative approaches to the task of creating music.

This chapter attempts to justify the main decisions that were taken in the design of MUSCIL. The MUSCIL user's manual describes the details of the implementation.

AN INSTRUMENTAL MODEL

An instrument is viewed as a sound producing device. The range, timbre and general amplitude characteristics of acoustic instruments is usually known. In the computer music idiom an instrument can be defined as a sound producing "unit" where range, timbre and amplitude characteristics are fabricated. The "construction" of this instrument is subject to many different views, specially in experimental music.

Since the days of the "Klangfarbenmelodie", the instrument is often used as a timbre before it is used as a "pitch-value" medium. Nevertheless an instrument will always be made to play something - whether it uses pitch or not as a primary syntactical vehicle.

What the instrument plays is the score. This score can be formulated with a list of numerical values that describe unequivocally what will be played. The "general" instrument as implementable with a computer should therefore be our reference when speaking of instrument.

Let us describe the task of composition as a succession of two recursive processes that cumulate in a completed work.

Think of an instrument. Compose a score of primary parameters for that instrument using rules and processes relevant to the compositional style used for the composition. Test the score with an available sound source, the piano or some other readily available instrument. Refine the score until satisfied with the results. Compose another part for a different instrument or the same instrument. This is the first recursive task.

Choose one of the previously composed scores and copy it on a multi-voice score at the appropriate place using modifications and transformations relevant to the compositional style. Repeat the preceding task until the work is completed. This is the second recursive compositional process.

In a computer-based environment these tasks are more precise and demand more attention from the composer. Some new tasks are also introduced by virtue of the computer's capacity to fabricate instruments.

Build an instrument (or a patch in the electro-acoustic terminology) and hear how it sounds with "dummy" parameter values. Refine the instrument by changing parameter values and/or reconfiguring the patch. Repeat this procedure until a family of instruments is available to the user's "orchestra".

Choose an instrument in the orchestra and write a score for it using rules and processes relevant to the compositonal style. Hear how the score sounds as played by the instrument. Refine the score until satisfied and/or hear what the same score sounds like when played by a different instrument. The essential difference of computer-based composition is perhaps most pronounced here. A score can be specified with absolute values with no margin left for human error during the interpretation stage of the work. The precision of scoring can be absolute and therefore necessitates special attention from the composer.

Creating complex multi-voiced scores is also a much more precisely defined task in a computer-based environment.

Choose a score and an instrument and enter it at the appropriate place in the score effecting modifications and transformations to the score relevant to the compositional style. Hear how the composite score sounds. Make modifications to the score, delete the score or choose another instrument to play the same score until the results sound satifactory. Repeat the preceding processes until the final results are obtained.

As witnessed by this description, the computer setting of composition offers more flexibility by allowing the composer to hear the results of a particular compositional decision immediately. Above and beyond the advantage of making instruments, the composer is given instantaneous feedback. Another advantage is that of modifying instances of a particular score without having to restate the complete score, but instead only the parts which are subject to modifications.

RULES AND PROCESSES IN COMPOSITION

As we have discussed, most of the composer's work will go towards specifying or following a set of rules. It is reasonable to state that music history has yielded many sets of rules that have or have not resisted the onslaught of time. Traditional harmony is a set of rules, dodecaphonic music is regulated by general and specific rules, and so is process music. One would be hard pressed to formulate general rules that apply for each and every style of music.

However there are definite constants in almost all music. Without calling them universal rules, we can at least suggest the nature of their universality. For instance, few musics never repeat themselves within a composition. Repetition is fundamental to music everywhere but in specific areas of contemporary music composition. This suggests that music is usually composed with small pre-composed units that are used with a number of variations.

Following this observation, one can comment on determinism in music. With the exception of a specific phase of the avant-garde period, no music has ever been fully deterministic. While the means of being fully deterministic are now available, it has never been proven to be desirable. Music, whether in performance or composition, always uses a considerable amount of indeterminism. This suggests that random processes are important, at least for the simulation of indeterminism at the micro level.

PARAMETER HIERARCHY

MUSCIL divides parameters in three classes : Primary(pitch-time), secondary(amplitude, articulation) and tertiary(timbre). This division should not, however, dictate that certain parameters cannot be "primary" in their controllability. This particular hierarchy was designed as a convenience to establish what types of processing MUSCIL should implement.

MUSCIL assumes the pre-eminence of pitch-time conglomerates in the hierarchy of musical parameters. Attention will be given to these two parameters in the form of a special kind of data structure that will facilitate their input and allow the composer to interact with MUSCIL mainly in terms of pitch-time parameters. Secondary parameters like articulation and amplitude are seen, in the design of MUSCIL, as functions of the pitch-time conglomerates. A mathematical language should therefore facilitate the control of these types of relationships.

Parametric interdependence is a powerful concept in the description of musical sound events, since computers can easily perform the complex transfer functions from one parameter to the other. MUSCIL will provide a special purpose mathematical processor to treat this problem. The most important effect of that feature will be to allow the composer to concentrate his creative energies on the production of pitch-time scores while other parameters - by virtue of mathematical expressions - can be defined once as functions of them.

COMPLEX WORKS

MUSCIL also assumes that, whatever the musical idiom employed, a piece of music always consists in the final analysis of a number of sound events, with or without silence, played solo or simultaneously with other sound events. The multi-voice score therefore becomes the main product of MUSCIL.

Complex works of music are usually constructed from smaller parts. Redundancy is almost always present in some form or other as a work unfolds. The composer will often use an instrumental part at the beginning of a work and re-use the same part with modifications again later in the composition. As in the multi-stave orchestral score, MUSCIL will allow an "orchestral" area where many individual "parts" can be organized together. This feature acknowledges the fact that most musics use polyphony and will influence the composer to think polyphonically. Perhaps the most important part of MUSCIL is the algorithm that allows the sorting of many individual instruments playing individual sound events at the same time. The particular task of "orchestrating" different voices is seen as the most important in the elaboration of complex musical works.

INTERACTION

Interaction was and still is much discussed in computer music as a primary feature of efficient and useful languages. Interaction permits a novice user an easier introduction to music-making with a particular language and greatly enhances his chances of being successful.

Another vital advantage of interaction is to permit immediate execution of special sub-tasks for testing purposes. In a composition session at the computer terminal, one will wish to hear small fragments before incorporating them in a full-fledged composition. Once a user has mastered the important processes in the elaboration of a musical work, the most valuable feature of a system becomes the ease of execution of these processes. A user must be given full flexibility to move back and forth between various "musical levels" of the language.

MUSCIL was conceived as a high-level music language for composers who are already familiar with the tasks involved in producing computer music works. The type of interaction offered is ease of execution and economy of data entry.

A REAL-TIME PROCESSOR

MUSCIL compiles a score in real-time. This is implemented by outputing event data to an FIFO output buffer. This buffer can be read from the top by an interpreter feeding a digital sound synthesizer, while it is being filled at the bottom by the MUSCIL execution cycle. The output cycle of MUSCIL will ouput 'chords' of all synchronous events in an orchestrated score and provide a waiting time for the synthesizer or synthesis program before it reads the next 'chord'. The next 'chord' of synchronous events is calculated during the waiting period and output after the buffer has been read by the synthesizer.

The real-time feature of MUSCIL will prove very useful when real time input 'windows' are left in the instrument definition of a particular parameter. An input device like a joystick or potentiometer can then be read through the window in the buffer to obtain current data. The output stage of MUSCIL will calculate a score of up to 48 voices.

B. MUSCIL : USER'S MANUAL

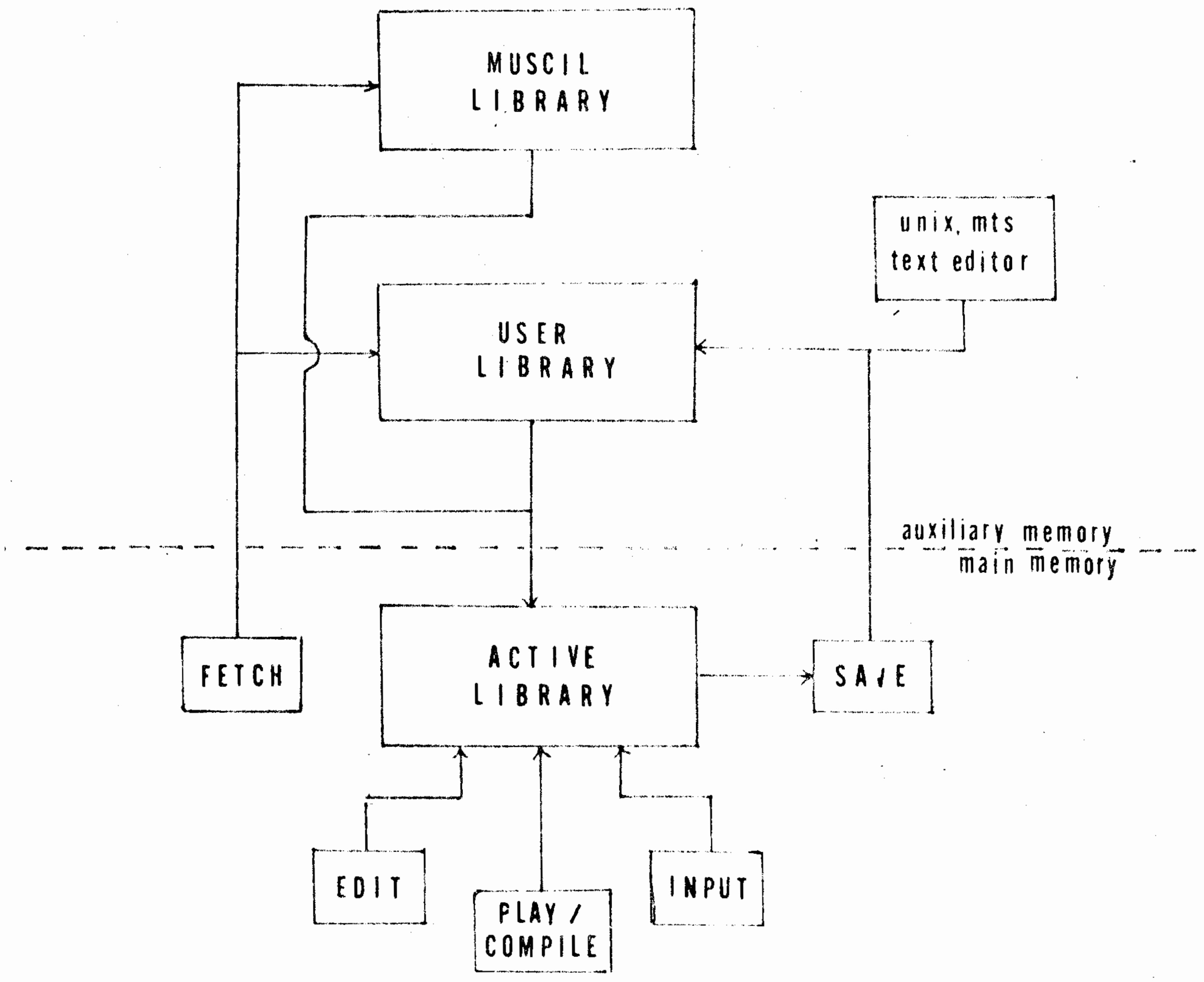
The following presents the syntax for the MUSCIL language. All options of MUSCIL are used and explained in the examples. Special symbols are defined and explained as well as the syntax of the MUSCIL Command Language.

I. THE MUSCIL WORKING ENVIRONMENT

Fig.3 shows the diagram of available resources in the MUSCIL working environment. The first section is a public use library that contains various data lists, instrument definitions and demonstration scores that can be accessed by any user to get useful "pre-defined" MUSCIL statements. This library will be of particular use to the unacquainted user who does not wish to immediately tackle all tasks involved the execution of a MUSCIL score. Standard instrument definitions, for example, can be borrowed from the public library and only a pitch-time score need be input in conjunction with that instrument to produce a playable score. The library is also used to store MUSCIL commands that have proven to be of general usefulness such as a crescendo function or a randomization function.

The second part of the library is the private library. This library is stored on disk space allocated to a user. In this file, the composer inputs MUSCIL data through a general purpose editor like UNIX or MTS or the editor available in the particular computer system. This file contains all the composed data relevant to the execution of a number of orchestrated scores by MUSCIL.

Fig. 3 — MUSCIL DATA STORES AND INTERACTIVE WORKING SPACE



When the MUSCIL program is started, a file name is asked for and all the data is read into the "active" storage of MUSCIL. The structure of the file must respect a strict input syntax that will be described at a later stage.

MUSCIL LIBRARIES

All interactive work is conducted through the active library, that which sits within immediate reach of the MUSCIL program. The active library is subject to changes in its contents at most points of a work session but it constitutes the only data from which the execution of a given score can be derived. Library management in MUSCIL is an important concept. At any given moment in a session, the user is given the option to redraw his compositioral intention completely with the use of a new set of definitions. The libraries are therefore exchangeable. The capacity of the "active" library, however, is limited. References to external libraries are subject to availability of space. The public library, for instance, might contain a vast number of definitions sufficient to exceed the alloted active storage. At the beginning of a session the user can choose to input his own library and parts of the public library. In the course of interacting with MUSCIL, the user can completely alter the contents of the library. The only time one is not allowed to do so is during a MUSCIL score compilation.

When a score is being compiled, all materials pertaining to that compilation must be in the active library.

Note: The composer may wish to work only in the interactive mode without reference to a particular user's file. In such a case,he can input all his data during a session and save the full active library in a personal file on disk.

INTERACTIVE COMMANDS

All user interaction is handled by a conversational subprogram that recognizes a number of commands and dispatches the necessary actions to have the commands executed. The interactive commands are explained below. <datalabel> indicates a user's defined name for a particuliar type of data structure. The various types of structures are explained later. <filename> refers to a user's private library from and to which data is being transferred.

FETCH <datalabel> <filename>.

The FETCH command is used to get data from the external libraries read into the active library. If the optional <datalabel> is omitted, the full contents of the file known as <filename> will be read.

INPUT <datalabel>.

The INPUT command is used to input any type of data into the active library from the computer terminal. Note that the type of data input is determined by the first letter of the <datalabel> and will be stored accordingly in the active library.

SAVE <datalabel> <filename>

The SAVE command will store the data contained in the active library under the <datalabel> into the appropriate store of the file known as <filename>. Here again the first letter of the <datalabel> determines the type of data. If no label is given the full active library will be saved.

EDIT <datalabel>

The EDIT command allows the user to change some of the data contained under the <datalabel>. The contents of the <datalabel> are printed on the screen. The edit functions are those of a common text editor.

PLAY <datalabel>

The PLAY command is the only command that is restricted to particular data structures. Only PITCH/TIME SCOREs, PARAMETRIC COMMAND LISTS and ORCHESTRAL COMMAND LISTS can be played. Refer to the next section for details.

MUSCIL DATA STRUCTURE

Fig.4 presents the various components of the MUSCIL active library. It is composed of three distinct parts.

1. The HEADER, where data is entered in storage units of the types PITCH/TIME SCORE, LIST and MASK. The HEADER is a data store that is accessed by PARAMETER COMMAND LISTS. The HEADER is optional in the executicn of a MUSCIL score compilation but is useful when identical strings of data are to be used more than once in a particular composition. Up to ten different entries of each type can be entered in the active library.
2. The PARAMETRIC COMMAND LISTS(PCL) or Instrument Definitions constitute the only compulsory data for a MUSCIL execution. The Active Library has space for 25 PCLs.
3. The ORCHESTRAL COMMAND LISTS(CCL). The Active Library will accept 3 CCLs. Each of these can consist of up to 48 tracks of instruments playing together.

The interpretation and execution cf sounds is not the responsibility of MUSCIL. It is therefore assumed that a MUSCIL file interpreter and a synthesizer are carrying out the actual playing of the instruments. It is alsc assumed that a 'patch' service program will define instruments to the interpreter.

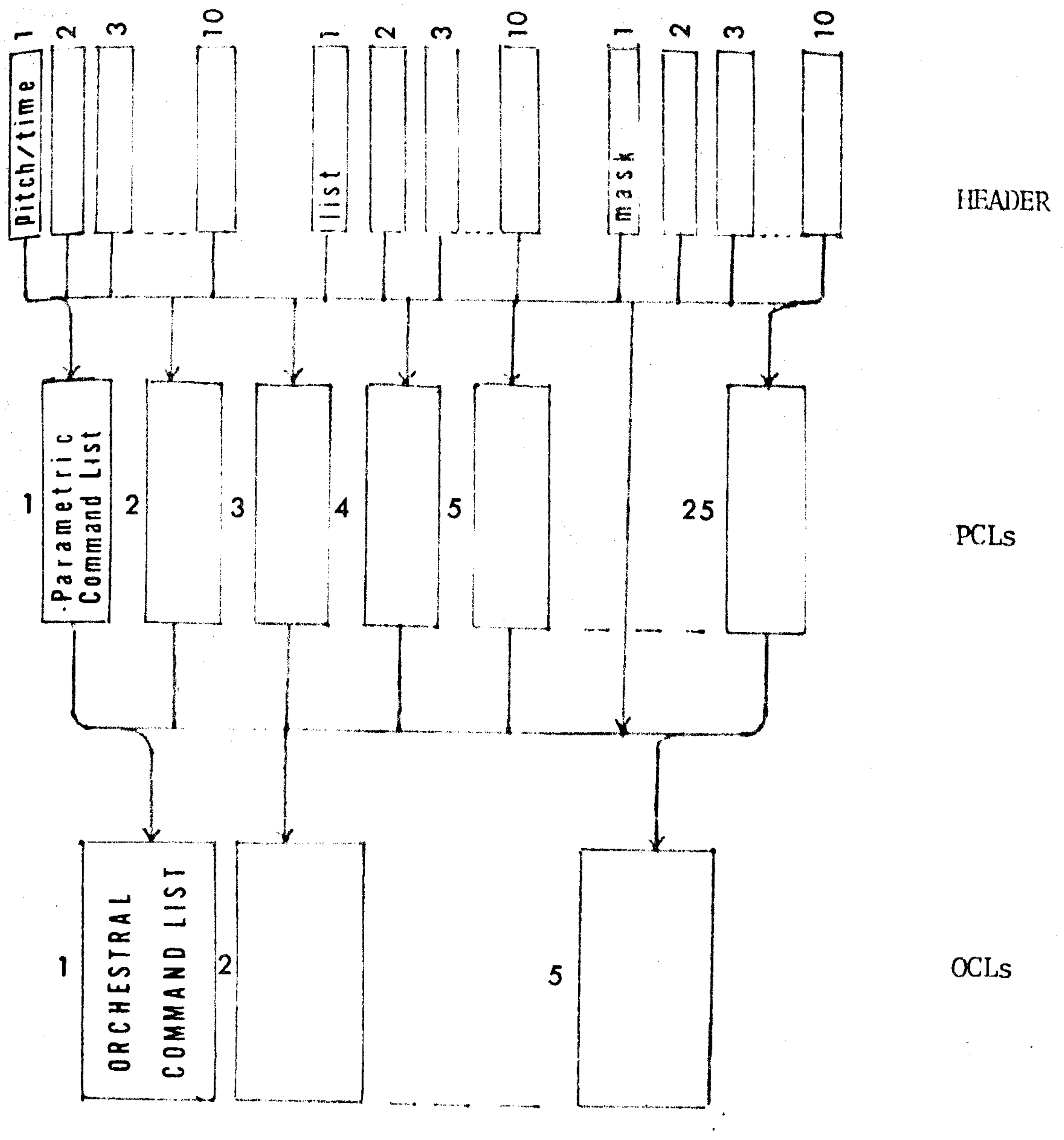


Fig. 4 The ACTIVE LIBRARY STRUCTURE

NOTE : For the purpose of our examples the instruments use a fixed waveform synthesis method and are controlled by fifteen input parameters similar the the MUSIC V instrument discussed in the first part of this text.

The ordering of parameters will be as follows :

P1 : Wait cycle(entry delay) before a new note in milliseconds.

P2 : Frequency in Centi-Hertz.

P3 : Duration of each note in milliseconds.

P4 : Maximum amplitude of each note in decibels.

P5 : Attack time in milliseconds.

P6 : Initial decay in milliseconds.

P7 : Duration of steady state in milliseconds.

P8 : Final decay in milliseconds.

P9 : Waveform number.

P10 : Vibrato rate in centi-Hertz.

P11 : Vibrato amplitude.

P12 : Vibrato attack time.

P13 : Random deviation for pitch warble in percentage.

P14 : Frequency of random fluctuation in centi-Hertz.

P15 : Waveform of vibrato oscillator.

II. THE HEADER

The storage area defined as the HEADER is divided in three sections. Each section stores a different type of data structure. Data contained in the HEADER area can only be accessed through the PARAMETRIC COMMAND LISTS. The HEADER in fact serves as an auxiliary storage for the PCLs, so that particular groups of data used often in a composition can be predefined, given a label, and recalled by using only the label. The three types of data structures contained in the HEADER are 1- The PITCH/TIME SCORE, 2- The LISTS and 3- The MASKS.

Note that HEADER data need not be present in the active library for the execution of a score. All HEADER data referenced from the PCLs must however be present.

PITCH/TIME SCORES

PITCH/TIME SCORES are a special type of data structure that allow a particularly efficient input of the two primary parameters of pitch and entry delay.

Each character on the computer terminal keyboard is assigned two values, one for pitch, the other for time. The configuration of different keyboard will vary but as a rule, the lowest pitch of the tempered scale (circa C1) is assigned to the leftmost/upperrow character (usually the character "1"). Moving to the right from that corner each key increments pitch by a semi-tone. One character represents one pitch. Therefore the full keyboard can be assigned a range of over eight octaves.

A future implementation of MUSCIL will offer alternatives to the tempered scale by allowing the composer to specify his own increment between keys. The base frequency(pitch) of the lowest key can also be changed to "modulate" the keyboard.

Time is also calculated by specifying an increment between the keys, the lowest key specifying the smallest indivisible time unit.

PITCH/TIME SCOREs are entered by first specifying the PITCH/TIME SCORE label. The first letter of the label must always be an "X". The next line will contain the time values in the form of characters and the third line will contain the pitch values. There need not be an equal number of pitches and times. The first time value is used in conjunction with the first pitch value and so on. The two lists cycle independently. The user must therefore ensure that, if that is desired, there will be a related number of pitches and entry delays.

PITCH/TIME SCOREs can only be used in a PARAMETRIC COMMAND LIST as a replacement for P1 and P2. This is done in the PARAMETRIC COMMAND LIST by listing a call to the label of a PITCH/TIME SCORE with the special "PLAY" command. The special form of this command will be treated in the following chapter on the MUSCIL COMMAND LANGUAGE.

EXAMPLE - THE PITCH/TIME SCORE

```
XVIO
703ygyg460tgtg469TGH69; (each character is an entry delay)
20392039gldgldgldgldgl; (each character is a frequency)


XVIA
8765;
NEWICNOPAMISN;


XVIE
T;
GLEGLS;


XVIF
TOCTOCBI;
$1;
```

LISTS

The LIST structure is simply a sequential list of data also to be used in the PCLs. A LIST can contain a number of elements of two types : an actual numerical value or a LIST label that contains LIST data. The first line of input is the LIST label which must always have the character "L" as the first letter.

The main advantage of the LIST structure is its potential for nesting LISTS within LISTS. There is no restriction to the level of nesting and this represents a powerful way of constructing complex structures.

LISTS are used sequentially by the "S" term in the PARAMETRIC COMMAND LIST. When a LIST label is encountered the full contents of that LIST is output before the next element in the original LIST is output.

The complete form of the term will be discussed in the next chapter on the MUSCIL COMMAND LANGUAGE. The user must of course be aware of where in the PARAMETRIC COMMAND LISTS a particular LIST will be used, as different parameters will expect different numerical value ranges.

EXAMPLE -SOME DATA LISTS.

```
LAMVIE (label on first line)
10 20 30 4 500; (data on next line)

LAMVIO
300 400 500 LAMVIE; (nesting allowed but not recursion)

LAMVIF
LAMVIO LAMVIA LAMVIO;

LAMVIA
100 200 300 400 LAMVIE;
```

MASKS

A MASK is defined as a two-dimensional geometrical construct specifying time-dependent limits within which a random number is generated. The specific use of MASKS will be discussed in the next chapter. The time co-ordinate is given in the PARAMETRIC COMMAND LIST; therefore, HEADER MASKs are strictly shapes that can be stretched in the time domain by a timing variable.

Masks are input in the HEADER the same way lists are input with the exception that only one mask comprising two,three or four coordinates can be entered under one label. Nesting is not allowed.

EXAMPLE - SOME MASKS.

```
MTRUM (label on first line)
10 15 ; (data on next line)


MTRUL
58 60 10 35 ;


MTRUP
100 500 2800;(if three coordinates are given the fourth is seen
             as equal to the third number)


MCLAP
0 1000 200 10;
```

III. THE PARAMETRIC COMMAND LIST

The next part of a MUSCIL input file is composed of Instrument definitions or PARAMETRIC COMMAND LISTS (PCL). The reader should at this point be concerned only with the structure of the PCL. The syntax of the command language will be described in the next chapter. Each instrument definition must begin with an instrument label, of which the first character is always an "I". It must also be terminated by the 'END;' control word. The PARAMETRIC COMMAND LIST is entered after the label line. One line of input must contain all the commands for the execution of one parameter. Command lines are input in order from P1 to P(n). The actual parameter number need not be given as MUSCIL will assume they are in order from P1 to P(n). The instruments here have 15 parameters to control. Each of these parameters must be supplied with a command string that will be used for the outputing of a parameter value as the instrument plays.

The active library contains space for 25 different instruments. One can simply input all command strings for the 15 parameters every time a new instrument is defined or one can use the 'COPY' option. This option uses the PCL of a previously defined instrument specified after the 'COPY' command. If the 'COPY' mode is used some of the parameter command strings may be substituted by inputing a new string preceded by the parameter to which it will apply.

It follows that at least one instrument must be defined in the MUSCIL active library. Other instruments can be copies playing different PITCH/TIME SCORE for example.
Note that all instrument definitions or PCLs must end with the control word 'END;'.

EXAMPLE - A PARAMETRIC COMMAND LIST USING THE PLAY OPTION.

```
IVICLIN
|4000 PLAY XVIO XVIO1 |3000 PLAY XVIO XVIO1 |4000 PLAY XVIO1;
P3:|P1 + 3000;
P4:|4000 S LAMVIE |3000 R MTRUM |4000 S LAMVIA;
P5:|R 20 25 30 60 | R 10 20;
P6:|P4 * R 0.25 1.35 - P1 * .025;
P7:|P3 * 2;
P8:|# 2;
P9:|# 6;
P10:|5000 s lamvie;
P11:|S 20 30 40 lamcla;
P12:|P3;
P13:|R MTRAP & 20 30 40 lamvie 400 2;
P14:|# 677;
P15:|P8 * R 0.98 1.03;
END;
```

EXAMPLE -A PARAMETRIC COMMAND LIST COPYING IVIOLIN AND CHANGING SOME PARAMETERS

```
ICLARINA
COPY IVIOLIN;
|8000 PLAY XCLA |3000 R MCLAP8 XCLA XCLA2 XCLA;
P3 : |8000 S LDUCLO |3000 S LDUCLO LDUCLA LDUCLE;
P4 : | R 50 60 10 5;
P5 : |# 1.255;
P6 : |4000 S 30 40 10 5 |7000 R 5 11;
END;
```

EXAMPLE -A PCL COPYING IVIOLIN AND SUBSTITUTING P1 & P2.

```
ITRUMPETT

COPY IVIOLIN

P1 : |5000 S LAMVIE |5000 S LAMVIA |2000 R MTRUP& 20 40 80 100;

P2 : | s 220 440 880;

P4 : |5000 R MTRUM |5000 R MTRUP |1000 R MTRUL;

P8 : P7;

END;

FIN;
```

<u>MUSCIL PARAMETRIC COMMAND LANGUAGE</u>

For every parameter of an instrument a command string is expected, whether duplicated from another instrument or input directly. The syntax for each command string is the same for every parameter with the exception of the optional 'PLAY' command where only certain types of processing are allowed. This case is treated separately.

THE TIMER - COMMAND SEPARATOR

If we consider the IVIOLIN (example), we find that each command is always preceded by the "¦" timer symbol. The optional number following the "¦" symbol is a timer value that will be counted down while the following command is executed. The times given as a timer are absolute. Every time the instrument plays a note or every time an event is output in the execution of an orchestration, the parameter timer is subtracted by the actual waiting cycle of the last event. When a timer has run out the next command in the command string is initiated with a new timer.

Some options are available in the use of timers. The "|" symbol however must always be present at the beginning of the command string. A numerical timer need not be given. If a numerical timer is not found, the command parser gives the subject command a timer equal to the timer given for the P1 parameter. This means that every time the command in P1 is changed, a similar update will occur in the parameter with no timer.

NOTE : a timer must always be given for the P1 parameter.

The timers also act as command separators. It is therefore important to understand that if a timer runs out and there are no further commands to process, i.e. the ';' terminator symbol is encountered, the command string will be recycled from the start. This does not hold for the P1 waiting cycle parameter. In that case when the end of the command string is met, the instrument stops playing, regardless of the timing values in the other parameters.

EXAMPLE -THE USE OF TIMERS IN THE PARAMETRIC COMMAND STRING

```
P(n):|200 command exp. |2000 command exp. |3000 command exp. ;

P(2-n) | command exp. | command exp. |command exp.;

P(2-n):| command exp.;
```

P(n) indicates that all parameters can use the following form. The other cases apply to all parameters except P1.

THE MUSCIL PARAMETRIC COMMAND EXPRESSION

A legal MUSCIL command expression is defined as an operation consisting of one or more 'terms' linked by 'operators'. A command delivers only one final value as a result of the expression given in the command. An expression must contain at least one MUSCIL term.

The user must ensure that the data input in a command is syntactically correct. The input parser only performs partial error reporting or correction.

1 - MUSCIL OPERATIONS

A MUSCIL command is always seen as an operation whatever the number of terms given. The command parser will read through the sentence and fabricate a linked list of operations between terms until a timer "|" or an end of string ";" is met. The operator symbols "*", "+", "-", and "/" are allowed. They respectively stand for : multiply, add, subtract and divide.

They also act as term separators. The execution will then look for an operator symbol and proceed to read the second term. When this is done, the parser looks for another operator and if found, another term, etc. This is carried out until the parser meets a "|" timer symbol. At that point the command is deemed complete and will be executed as such. The execution is always done linearly therefore not allowing the use of brackets. A product will be obtained from the operation of the first two terms and this in turn will become the first operand of the next operation involving the next term as the second operand. Here are some examples of legal operation chains.

EXAMPLE - SOME STRINGS OF MUSCIL COMMAND EXPRESSIONS.

```
|3000 term1 * term2 |3000 term3 + term2 ;
```

(each type of term is defined later. The syntax of an expression requires that each term of an expression be separated by an operator. Each expression must be separated by the "|" timer symbol)

```
|2000 term1 * term2 + term3;
```

```
|120 term1; (the one-term expression is allowed)
```

```
|3000 term1|2000 term2|5000 term4;
```

```
|term1|term2|term3;
```

```
|3000 term1 + term2 / term1 * term4 + term4;
```

2- MUSCIL TERMS

In an operation consisting of two or more terms, each term must be separated by an 'operator' symbol that also gives the kind of operation to be performed between two terms. There are four types of legal terms in MUSCIL; each one is discussed. Note that terms are parsed after a new command is initiated.

A)-THE S (for Sequential) TERM

The sequential command is invoked by the "S" symbol followed by the data to be cycled. The only legal types of data are numerical values and list labels. In the later case the list label must be present in the current 'Headerlist' library. The sequential processor will cycle all data given after the "S" symbol and before an operator or the next timing "|" symbol.

When a list label is given, the data of that list is sequentially read before the next value in the term data string is read. When using the PITCH/TIME SCORES, the S term is automatically assumed and is replaced by the PLAY command followed by the PITCH/TIME SCORE labels. Note that PITCH/TIME SCORES and the PLAY command can only be given as a replacement for P1 and P2.

EXAMPLE -USE OF THE SEQUENTIAL (S) TERM.

```
|3000 S 200 300 400 ;

|S lamvio lamvia;

|S lamvio 200 300 lamvia 400;




|3000 PLAY XVIO XVIE XVIA;(note how the PLAY replaces the "S")

|PLAY XVIA;
```

B)-THE R (for Random) TERM

The random generator is invoked by the control symbol "R" followed by either a mask label or a list of two, three or four numbers giving the coordinates of a tendency mask that will be used to calculate the time-varying minimum and maximum values used by the random number generator. The timing value for the stretching of the figure will be the same as the one given for the timer. This assignment is done after the timer has been obtained, therefore guaranteeing a timer will be known. The simplest case :

```
R 10 200;
```

In this case a random number (equal weight) will be generated between 10 and 200 every time the term is called upon to deliver data. The time variant factor here will not have any effect because both limits remain constant.

Three or four numbers can define a time-variant mask. A mask can also be defined in the header and called upon in the command string. The first two numbers always give the starting values of the random limits. The other number(s) gives the position of the two limits at the end of the timing period. If there are only three numbers the third number is used as maximum and minimum at the end of the timing period. Only one mask can be input in a single term.

EXAMPLE -USE OF THE RANDOM (R) TERM.

```
R 10 105; (two numbers provide constant limits)

R  10  200  400; (The fourth number when not given is made equal
              to the third number>

R 200 400 500 1000;

R MTRUM; (only one mask label can be referred  to  in  a  single
              R statement)
```

A particularly interesting way of using the random command is as a selector rather than a generator.In the latter case a value returned by the random number generator becomes the value of the term. When used as a selector, the random number is not returned as the value for the term but is used to select a value in a given list of values. The form of this special use of the "R" command expects a)- the R symbol, b)- mask, c)- the special control symbol "&" and c)- a list of values (list labels or numerical).

The list of data to be chosen from remains the same throughout the execution of the command. The items to be selected are counted and given a range. The number received from the random generator must be situated between 0 and 100. The first item in the data list is assigned the lowest range and the last item in the list is assigned the highest range. If the item selected by the random number is a list label, the full contents of the list is output before another selection is made.

EXAMPLE -The RANDOM SELECTOR.

```
R MTRUM& LVIC LVIA LVIE; (all data following the "&"
              is subject to selection)

R 0 10 10 80& 20 30 50 70 80 100 3 5 8 19 555.4;

R MTRUM& 30 40 50 LAMVIO 60 80 90 40 555.2:

R MTRUM & 40 60 70 80 90 100 110 120 300;
```

C)-THE PARAMETER TRANSFER TERM

A term can consist of a simple reference to a current parameter value that already has been calculated and output to the output buffer. Since parameters are processed one at a time it is possible to use a precedent parameter value as a term. The code symbol "P" is followed by the parameter number we wish to copy. If an attempt is made to copy a parameter that has not yet been output, an error will occur. Note that a parameter value can only be copied from the same instrument.

For example :

```
13000 P4;
12000 P2 * P3;
```

In these examples whatever parameter the expressions are applied to will use the previously calculated value from the specified parameter.

D)-THE CONSTANT TERM

A constant value can be given as a term by simply inputing that value. Special precautions must be taken, however, if the constant happens to be the first term after a "|" timing symbol. In such a case the constant will be confused with a timer value. If the timer value is to be copied from P1, the timer symbol should immediately be followed by the "#" symbol.

For example :

```
|4000 4.55;
|200 600 + 3.3;
|# 4.4;
|# 60000;
```

IV. THE ORCHESTRAL COMMAND LIST

The ORCHESTRAL COMMAND LIST of MUSCIL can process up to 48 instruments in parallel, and can best be thought of as a 48-track recorder for control data. One of the most important processes in MUSCIL is to sort up to 48 instruments playing different scores at the same time.

After having input a number of instrument definitions or PCLs, the user has the option to input a number of orchestrations of the predefined instruments. MUSCIL uses the ORCHESTRAL COMMAND LIST to compile a polyphonic score.

An orchestration is input by first stating the name of the orchestration (the first character must be an "O"). On the next line the first "track" is input. Subsequent lines all contain data for one track for up to 48 lines of input. When the number of desired "track-lines" is input, the "END;" control word is expected. All tracks are analoguous to a voice in an orchestral score.

EXAMPLE -A 10 voice ORCHESTRAL COMMAND LIST.

```
OSECTION1;
|20000 IVIOLIN(P2*1.01)|10000 ICLARINA|200 STOP|IVIOLIN;
|10 STOP|19900 IVIOLIN|100 STOP|ICLARINA|2000 ITRUMPETT;
|200     ITRUMPETT(P1+7.7)|200     ICLARINA|200     IVIOLIN|20000
STOP|IVIOLIN;
|IVIOLIN|ICLARINA|ITRUMPETT(P2*0.998)|20000
STOP|IVIOLIN|ITRUMPETT;
|2.2 STOP|IVIOLIN|2.1 STOP|ICLARINA;
|10000 IVIOLIN|10000 ICLARINA|200 STOP|IVIOLIN;
|10 STOP|19900 IVIOLIN|100 STOP|ICLARINA|2000 ITRUMPETT;
|200 ITRUMPETT|200 ICLARINA|200 IVIOLIN|20000 STOP|IVIOLIN;
|IVIOLIN|ICLARINA|ITRUMPETT|20000 STOP|IVIOLIN|ITRUMPETT;
|200 IVIOLIN|ICLARINA|ITRUMPETT;
END;
```

(Each line terminated by the ";" terminator contains data for one "track". All tracks are processed in parallel.)

THE ORCHESTRAL COMMAND LANGUAGE

Any instrument that is used in an orchestration must have been defined previously, otherwise an error will occur.
The sorting algorithm of MUSCIL ensures that all instruments will be playing their actual score regardless of other tracks.

1- TIMERS

Timers work the same way in the ORCHESTRAL COMMAND LIST as they do in the PARAMETRIC COMMAND LISTs. Different commands are separated by timer symbols "!" and the number following it is also optional. When a timer is not given, the timer will take the form of a "flag" that will be raised when the instrument label following the timer has played all its notes. When this happens the next command in that track is initiated. If a timer is given in the form of a number, and given that this timing value exceeds the time the following instrument would normally play, the instrument is re-initiated and starts playing at the beginning again until the track timer has run out.
If the timer is less than the normal playing time of the instrument, the instrument is discarded before the end of its part and a new command is initiated.

2- COMMANDS

The "STOP" command is used to introduce silence in a particular track at any given point. The "STOP" command must always be used in conjunction with an actual numerical timer.

A number of modifications can be made to parameter values received from the PARAMETRIC COMMAND LISTs. Essentially the PARAMETRIC COMMAND EXPRESSIONS apply as defined for the PARAMETRIC COMMAND LISTs. Parameters can therefore be changed inside the ORCHESTRAL COMMAND LISTs. There are a few particularities in their use. The parameter we wish to alter must clearly be stated, and the expression must be included between "()" brackets. If more than one parameter is to be altered, each parameter expression must be separated by the "$" symbol.

EXAMPLE - SOME ORCHESTRAL COMMAND LISTS USING EXPRESSIONS.

```
|IVIOLIN(P2 * 1.333)|2000 ITRUMPETT(P1 + 100 $ P4 +10);
```

(IVIOLIN will play its score with all P2 values transposed by one third. ITRUMPETT will then play 2000 time units with P1 having 100 added and P4 having 10 added.)

```
|200 IVIOLIN (P9 + P5);
```

```
|ICLARINA (P6 + P8 + P9)|ICLARINA(P6 + P10);
```

APPENDIX I

A SAMPLE MUSCIL USER'S FILE

The following compositional file is known as "RS" . "RS" is a user's file that was written with the MTS time-sharing system editor (in the current implementation of MUSCIL). Note the special use of the "END;" control word to indicate that a particular section of the file is terminated.

```
HEADER

1.  XVIO
2.  7657657657657657657657765;
3.  gfgfgfgfgfgfgfgfgfg;
4.  XVIA
5.  GhvvKLl;
6.  tR;
7.  XVIE
8.  tref;
9.  acu)asc)abu)ghGHS6123123;
10. END;


11. LISTA
12. 20 40;
13. LISTB
14. 22 55 LISTA LISTC 44;
15. LISTC
16. 1.2 3.2 4.3 5.4 5.5 LISTD;
17. LISTD
18. 330 660 990 115 330;
19. LISTE PT>8.2 7.2 6.2 5.2 6.2 7.2 8.2;
20. LISTF
21. 220 440 880 440 220 880 1760 220;
```

```
22. END;


23. MCLA
24. 10 20 50 100;
25. MCLIP
26. 0.99 1.04;
27. MTRU
28. 200 300 100 900;
29. END;
```

INSTRUMENT DEFINITIONS - PARAMETRIC COMMAND LISTS

```
30. IVIOLIN
31. P1 |5800 S 80 40 80 40 20;
32. P2 |10000 S LISTF* R 0.99 1.01;
33. P3 |10000 R 5000 3200;
34. P4 |10000 S LISTA * R 0.9 1.1;
35. P5 |4000 R 1 5 10 908 LISTA LISTB 30 40 50 60 703 800;
36. P6 |500 S 100 200 300;
37. P7 |800 P1* P6;
38. P8 |1600 S 1 2 3 4 5 6 7 8 9+ R 0.01 1.02|1600 R 1 1 30 60;
39. P9 |R 500 1000;
40. P10 |100 S 400 500 600|200 S 301 401 |1000 2 / P7;
41. P11 |1600 4;
42. P12 |1200 5;
43. P13 |100 7;
44. P14 |1200 S 8 9 10 11;
45. P15 |5200 R 300 1800 1 3;
46. END;


47. ICLARINA
48. COPY IVIOLIN
```

```
49. P1 |5000 S LISTA 20 20 40 80 30 50 LISTA;
50. P2 |S LISTF LISTD* R 0.99 1.01 + 6;
51. P9 |5000 R 100 300 500 700;
52. END;


53. ITRUMPETT
54. COPY IVIOLIN
55. P1 | 2000 S 40 50 30;
56. P2 |S LISTD* R 0.99 1.01;
57. END;
58.


59. FIN;
```

ORCHESTRAL COMMAND LISTS

```
SECTION1;
|500 IVIOLIN|200 STOP| ITRUMPETT|40 STOP|400 ITRUMPETT;
|500 ITRUMPETT|350 STOP|500 ICLARINA|0.20 STOP|ICLARINA;
|10 STOP|1990 IVIOLIN|500 STOP|1000 ICLARINA;
|200 STOP |490 ITRUMPETT|10 STOP| IVIOLIN;
|500 STOP | IVIOLIN|300 STOP|100 ICLARINA;
END;


SECTION2;
|1500 IVIOLIN;
|1500 IVIOLIN;
|1500 IVIOLIN;
|1500 IVIOLIN;
|1500 IVIOLIN;
|1600 ICLARINA;
|1600 ICLARINA;
|1600 ICLARINA;
|1600 ICLARINA;
|1600 ICLARINA;
|1600 ITRUMPETT;
```

```
79. |1700 IVIOLIN;

80. |1700 IVIOLIN;

81. |1500 IVIOLIN;

82. |1600 ICLARINA;

83. |1600 ITRUMPETT;

84. |1500 IVIOLIN;

85. END;



86. SECTION3;

87. |IVIOLIN;

88. |ICLARINA;

89. |ITRUMPETT;

90. END;



91. FIN;
```

APPENDIX II

A PASCAL IMPLEMENTATION OF MUSCIL

FIG. 5 SHOWS THE DATA STRUCTURE OF THE ORCHESTRAL COMMAND LIST. ALL DATA PROCESSED BY MUSCIL IS DONE SO THROUGH A SIMILAR STRUCTURE.

THIS VERSION OF MUSCIL USES SPECIFIC EXTERNAL FUNCTIONS IN THE UBC IMPLEMENTATION OF PASCAL. THERE ALSO IS AN EXTERNAL CALL TO A FORTRAN RANDOM NUMBER GENERATOR IN THE NAG LIBRARY OF SCIENTIFIC SUBROUTINES.

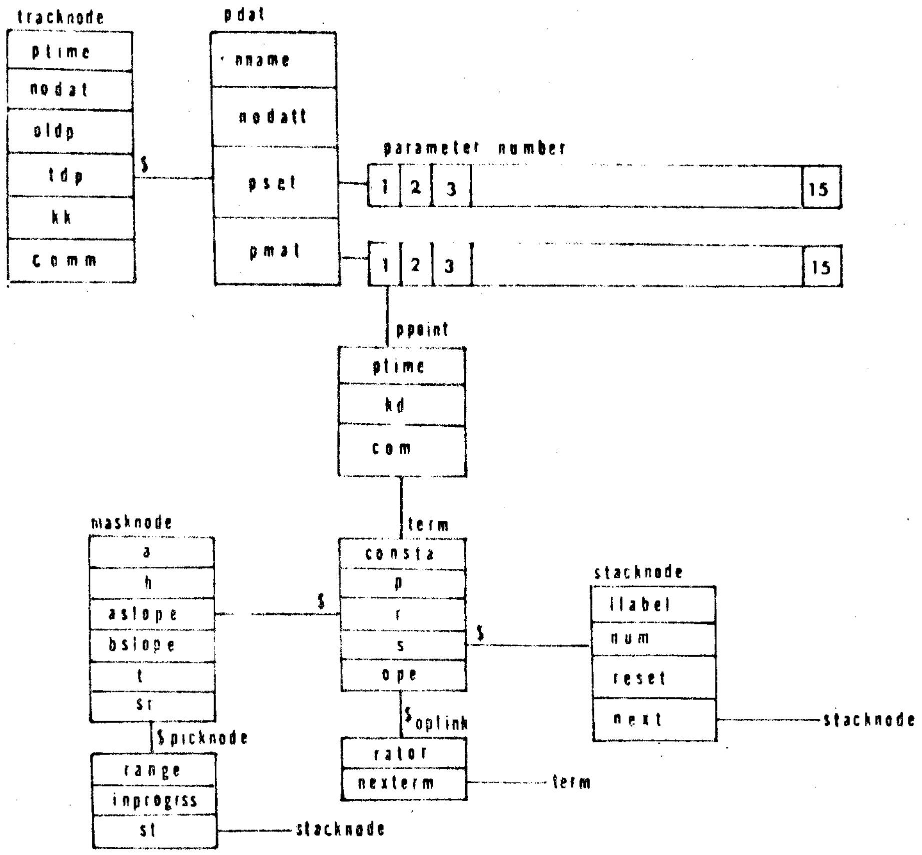


Fig. 5 The OCL Data Structure for the compilation of one track.

```
PROGRAM MUSCIL;

CONST

 MAX     = 999999;

 ONE     = 1;

 TWO     = 2;

 STOPCHARS  = (.'=','&','*','>',';','

 NUMS       =  (.'1','2','3','4','5','6','7','8','9','0'.);

 OPERCHARS  = (.'*','/','+','T','-','>',',

                                  CARS                                  =

(.'L','M','1','2','3','4','5','6','7','8','9','0'.);

 NOOFPARAM  = 15;
```

```
TYPE

      TP = TRACKNODE;

      ITEM = ARRAY (1..12) OF CHAR;

      FILEID = ITEM;

      DATAPOINTER = PDAT;

      OCLOUTREC = RECORD

                    TRACKNUM :INTEGER;

                    BUFSET : ARRAY(1..15) CF SHORT;

                    END;

      OCLOUTPUT = ARRAY(1..50) OF CCLOUTREC;

      OCL     =     RECORD

                     SMALLP   : SHORT;

                     OUTBUFFER :  OCLOUTPUT;

                     TRARRAY    : ARRAY  (1..48)  OF TP;

                     NOOFTRACK : INTEGER;

                     END;

      TRACKNCDE   =   RECORD

                       PTIME      : SHORT;

                       NCDAT      : BOOLEAN;

                       OLDP1      : SHORT;

                         CCM      : ITEM;

                         TDP      : DATAPOINTER;

                          KK      : INTEGER;

                       END;

      OCLPTR   = OCLINPUT;
```

```
CHARLINE = ARRAY (1..100) OF CHAR;
OCLINPUT = RECORD
           OCLNAME : ITEM;
           OCLDAT  : ARRAY (1..48) OF CHARLINE;
           END;
OCLTREE = ARRAY (1..4) OF OCLPTR;
OPPTR = OPLINK;
TERMPTR   = TERM;
STKPNT    = STACKNODE;
LISTPT = LISTINPUT;
LISTINPUT = RECORD
            LLNAME : ITEM;
            LDAT   : CHARLINE;
            END;
LISTREE = ARRAY (1..25) OF LISTINPUT;
PCLA = RECORD
       PCI : ARRAY (1..NOOFPARAM) OF CHARLINE;
       PCNAME : ITEM;
       END;
PCLTREE = ARRAY(1..10) OF PCLA;
STACKNODE = RECORD
            LLABEL  : CHARLINE;
              NUM   : INTEGER;
            RESETPT : INTEGER;
            NEXT    : STKPNT;
```

```
               END;
PICKNODE   =  RECORD
                RANGE : SHORT;
                INPROGRSS: BOOLEAN;
                ST    : STKPNT;
                END;
MASKNODE   =  RECORD
                T2      : SHORT;
                ASLOPE  : SHORT;
                BSLOPE  : SHORT;
                SR      : PICKNODE;
                A,B     : SHORT;
                END;
TERM       =  RECORD
                ACTUAL : CHAR;
                CONSTA :SHORT;
                PI     : INTEGER;
                S      : STKPNT;
                R      : MASKNODE;
                OPE     : OPPTR;
                END;
OPLINK     =  RECORD
                RATOR : CHAR;
                NEXTERM :TERMPTR;
                END;
```

```
PPOINT     =   RECORD
               PTIME    : SHORT;
               COM      : TERM;
                KD      : INTEGER;
               END;
   PDAT    =   RECORD
               NODATT : BOOLEAN;
               NNAME : ITEM;
               PSET   : ARRAY (1..NOOFPARAM) OF SHORT;
               PMAT   : ARRAY (1..NOOFPARAM) OF PPOINT;
               END;
  STACKARRAY = ARRAY (1..46) OF STKPNT;
  TERMPTRARRAY = ARRAY (1..46) OF TERMPTR;
  OPPTRARRAY   = ARRAY(1..46) OF OPPTR;
```

```
VAR
   TESTDAT : PDAT;
   PCLLIB : PCLTREE;
   OCLLIB : OCLTREE;
   OCLSOURCE : OCLINPUT;
   FF,EVENT,PARAM,H,HH    : INTEGER;
   TCTIME : SHORT;
   HEADLIST : LISTREE;
   OCLA     : OCL;
   DATAFILE : TEXT;
   FILENAME : FILEID;
   COMMAND : ITEM;
   FINISHED : BOOLEAN;
   NUMOFINST : INTEGER;
   STORESTACK : STACKARRAY;
   STORETERMPTR : TERMPTRARRAY;
   STOREOPPTR   : OPPTRARRAY;
   PASSPOINTER : TERMPTR;
   LLAST : INTEGER;
```

```
FUNCTION G05DAF(X : REAL;Y : REAL;): REAL; SLINKAGE;
PROCEDURE MASKSET(VAR ADRSS : CHARLINE;VAR I :INTEGER;
                  VAR MPASS : MASKNODE);FORWARD;
PROCEDURE  PCOMINIT  (VAR  INLINE  :  RECINT;  VAR  PCLAB  :
CHARLINE);FORWARD;
PROCEDURE TIMEUPDATE(VAR OCLA : OCL); FORWARD;
PROCEDURE OUTPUT(VAR OCLA : OCL) ; FORWARD;
PROCEDURE TRACKREAD (VAR KK : INTEGER;
                     VAR ITEMX : ITEM);FORWARD;
PROCEDURE COMTIMINIT(VAR TRANSFER : TRACKNODE;
                     VAR INTAKE : CHARLINE);FORWARD;
PROCEDURE FINDSMALL(VAR OCLA : OCL); FORWARD;
PROCEDURE GIVSET(VAR INDAT : DATAPOINTER); FORWARD;
PROCEDURE SEARCHPCLTREE(VAR INNAME : ITEM; VAR LCRETU : PCLA);
VAR INDEX : INTEGER;
    FOUND : BOOLEAN;

              (*SERVICE ROUTINE TO SEARCH THE INSTRUMENT LIBRARY
                FOR AN INSTRUMENT NAME. IF FOUND THE INSTRUMENT
                DATA IS RETURNED VIA 'LCRETU'.*)

BEGIN
  FOUND := FALSE;
  INDEX := 1;
  WHILE NOT FOUND AND INDEX
   BEGIN
    WITH PCLLIB(INDEX) DO
     IF PCNAME = INNAME THEN
      BEGIN
        FOUND := TRUE;
        LCRETU := PCLLIB(INDEX);
      END;
     ELSE INCR(INDEX);
   END;
  IF NOT FOUND THEN LCRETU.PCNAME := 'NOT FOUND';
END;
```

```
PROCEDURE NEWS(VAR SS : STKPNT);
VAR A : INTEGER;

        (*SERVICE ROUTINE TO ALLOCATE A NEW STACKNODE
          POINTER FROM THE STACKNODE POINTER STORE.*)

BEGIN
 IF STORESTACK(1)  NIL THEN
   BEGIN
    SS := STORESTACK(1);
    FOR A := 1 TO 45 DO
    STORESTACK(A) := STORESTACK(A+1);
   END;
ELSE NEW(SS);
END;
```

```
PROCEDURE DISPOSS(VAR SS : STKPNT);

              (*SERVICE ROUTINE TO STORE A DISCARDED STACKNODE
                POINTER IN THE 'STORESTKPNT'. THE DATA POINTED
                AT IS NULLIFIED*)

VAR INDEX : INTEGER;
BEGIN
IF SS  NIL THEN
  BEGIN
   INDEX := 1;
   WHILE STORESTACK(INDEX)  NIL AND INDEX
   IF STORESTACK(INDEX) =NIL AND INDEX
    BEGIN
    STORESTACK(INDEX) := SS;
    WITH STORESTACK(INDEX) DO
      BEGIN
       LLABEL := '                                   ';
       NUM    := 1;
       RESETPT:= 1;
       NEXT := NIL;
      END;
    END;
  END;
DISPOSE(SS);
```

```
END;
```

```
PROCEDURE DISMANTLSTACK(VAR SS : STKPNT);

V          (*SERVICE ROUTINE TO DISMEMBER A LINKED LIST
             OF STACKNODES AND SAVE THE POINTERS FOR
             LATER USE.*)

BEGIN
 IF SS  NIL THEN
  BEGIN
   IF SS.NEXT = NIL THEN
    DISPOSS(SS);
   ELSE
    BEGIN
    DISMANTLSTACK(SS.NEXT);
    DISMANTLSTACK(SS);
    END;
  END;
END;
```

```
PROCEDURE NEWTERM(VAR SS : OPPTR);
VAR A : INTEGER;
                        (*SERVICE ROUTINE TO ALLOCATE A NEW
                          POINTER 'OPPTR' FROM THE POINTER STORE.
                          IF THE STORE IS EMPTY, A NEW POINTER IS
                          RETURNED*)
BEGIN
 IF STORECPPTR(1)   NIL TEEN
    EEGIN
     SS := STCRECPPTR(1);
     FOR A := 1 TO 45 DO
     STCREOPPTR(A)  := STOREOPPTR(A+1);
    END;
ELSE NEW(SS);
END;
```

```
PROCEDURE DISPOSTERM(VAR SS : OPPTR);

               (*SERVICE ROUTINE TO STORE A DISCARDED 'OPPTR'
                IN THE 'STOREOPPTR' AND MAKE IT AVAILLABLE
                FOR LATER USE.*)
VAR INDEX : INTEGER;
BEGIN
IF SS  NIL THEN
  BEGIN
   INDEX := 1;
   WHILE STOREOPPTR(INDEX)  NIL AND INDEX
   IF STOREOPPTR(INDEX) =NIL AND INDEX
    BEGIN
    STOREOPPTR(INDEX) := SS;
    WITH STOREOPPTR(INDEX) DO
      BEGIN
       RATOR  := ' ';
       NEXTERM.OPE := NIL;
      END;
    END;
  END;
DISPOSE(SS);
END;
```

```
PROCEDURE DISMOUNT(VAR SS : TERM);


       (*SERVICE ROUTINE TO DISMEMBER A LINKED LIST OF TERMS*)
BEGIN
 IF SS.OPE  NIL THEN
  BEGIN
   IF SS.OPE.NEXTERM.OPE  NIL THEN
      DISMOUNT(SS.OPE.NEXTERM);
   ELSE DISPCSTERM(SS.OPE);
  END;
END;
```

```
PROCEDURE PUSHSTACK (VAR STACK : STKPNT; VAR NAME :ITEM);

                (* SERVICE ROUTINE TO SEARCH THE LIST
                   LIBRARY FOR A LIST LABEL, CREATE A NEW
                   NEW STACKNODE AND PLACE THE NEW NODE
                   AT THE START OF THE LINKED LIST*)
 VAR NEWNODE : STKPNT;
     INDEX : INTEGER;
     FOUND : BOOLEAN;
BEGIN
 FOUND := FALSE;
 INDEX := 1;
 WHILE NOT FOUND OR INDEX > 25 DO
     IF HEADLIST(INDEX).LLNAME = NAME THEN
        FOUND := TRUE;
     ELSE
      IF INDEX >= 25 THEN
         WRITELN (' ERROR IN STACK. LIST DOES NOT EXIST');
      ELSE INCR (INDEX);
  NEWS(NEWNODE);
  NEWNODE.LLABEL := HEADLIST(INDEX).LDAT;
  NEWNODE.NUM := 1;
  NEWNODE.RESETPT := 1;
  NEWNODE.NEXT := STACK;
  STACK := NEWNODE;
```

```
END;
```

```
PROCEDURE POPSTACK(VAR STACK : STKPNT; VAR NAME : ITEM);

          (*SERVICE ROUTINE TO DISPOSE A STACKNODE AND
            REESTABLISH THE LINK TO THE NEXT STACKNODE IN
            A LINKED LIST OF STACKNODE*)

VAR TEMP : STKPNT;
 BEGIN
  TEMP := STACK.NEXT;
  DISPOSS(STACK);
  STACK := TEMP;
 END;
```

```
PROCEDURE ITEMREAD(VAR LINE : CHARLINE;VAR K : INTEGER;
                         VAR ITEMR : ITEM);

                  (*SERVICE ROUTINE TO READ SEPARATE DATA ITEMS
                    AND CONTROL SYMBOLS IN A PCL COMMAND STRING.*)

     VAR  J,I : INTEGER;

     BEGIN
       FOR J := 1 TO 12 DO ITEMR(J) := ' ';
       I := 1;
       WHILE LINE (K) = ' ' DO
       INCR (K);
       IF LINE(K) IN STOPCHARS THEN
         BEGIN
           ITEMR(I) := LINE(K);
           INCR (K);
           INCR (I);
         END
       ELSE
         BEGIN
                 WHILE        LINE(K)         NOT         IN        (.'
',',','|','&',';','/',':','-','*','+',',','/','='.) DO
            BEGIN
             ITEMR(I) := LINE(K);
            INCR (K);
            INCR (I);
           END
        END
END;
```

```
PROCEDURE PTRINCR (VAR LABA :CHARLINE;VAR KO : INTEGER;
                   VAR OPF : BOOLEAN);
             (*SERVICE ROUTINE TO SEARCH FOR AN
               OPERATOR IN A LINE OF CHARACTERS*)

BEGIN
 WHILE LABA(KO) NOT IN STOPCHARS DO INCR (KO);
 IF LABA(KO) IN OPERCHARS THEN OPF := TRUE ELSE OPF :=FALSE;
END;
```

```
PROCEDURE LISTSET(VAR INS : STACKNODE; VAR KK :INTEGER;
                  VAR LABE : CHARLINE);
             (*SERVICE ROUTINE TO INITIALIZE A STACKNODE*)
BEGIN
    INS.LLABEL := LABE;
    INS.NUM := KK;
    INS.RESETPT := KK;
    INS.NEXT := NIL;
END;
```

```
PROCEDURE SETERM (VAR COMDAT : TERM; VAR KT :INTEGER;
                    VAR LABS : CHARLINE; VAR ITEMN : ITEM);
VAR          ITEMTI : ITEM;
Z,I,J,L : INTEGER;H : SHORT;
OPFIND : BOOLEAN;

            (*SETERM WILL INITIALIZE THE TERM BY READING
             IN THE CHARACTER LINE 'LABS'. THE FIRST
             SYMBOL RETURNED WILL DETERMINE WHICH OF
             OF THE SETTING ROUTINES WILL BE CALLED.
             ONCE A TERM HAS BEEN INITIALIZED, SETERM
             WILL READ FURTHER SEARCHING FOR AN OPERATOR.
             IF IT IS FOUND, AN OPERATOR LINK IS ESTABLISHED
             AND SETERM IS CALLED RECURSIVELY TO INITIALIZE
             THE NEXT TERM OF THE OPERATION. THIS PROCESS
             IS REPEATED UNTIL NO OPERATOR CAN BE FOUND.*)

BEGIN
 ITEMTI := ITEMN;
     WITH COMDAT DO
      BEGIN
        CASE ITEMTI(1) OF
               'S'  :  BEGIN
                       ACTUAL := 'S';
                       DISMANTLSTACK(S);
                       NEWS(S);
                       LISTSET(S,KT,LABS);
                       ITEMREAD(LABS,KT,ITEMTI);
                       END;
              'R'  : BEGIN
                       ACTUAL := 'R';
                       MASKSET(LABS,KT,R);
                     END;
'0','1','2','3','4','5','6','7','8','9' :
                      BEGIN
                      ACTUAL := 'C';
                      READSTR(ITEMTI,ONE,H);
                      CONSTA := H;
                      END;
              'P'      : BEGIN
                           ACTUAL := 'P';
                           READSTR(ITEMTI,TWO,Z);
                           PI := Z;
                         END;
           : WRITELN ('ERROR IN COMMAND PARAMETER :',PARAM);
                         END;
    PTRINCR(LABS,KT,OPFIND);
    DISPOSTERM(OPE);
    IF OPFIND = TRUE THEN
            BEGIN
```

```
                ITEMREAD(LABS,KT,ITEMTI);
                NEWTERM(OPE);
                NEW(OPE.NEXTERM);
                OPE.NEXTERM.OPE := NIL;
                OPE.RATOR := ITEMTI(1);
                ITEMREAD(LABS,KT,ITEMTI);
                SETERM(OPE.NEXTERM,KT,LABS,ITEMTI);
                END;
       END;
   END;
```

```
PROCEDURE PCOMINIT (VAR INLINE : PPOINT; VAR PCLAB : CHARLINE);
               VAR  ITEMX : ITEM;
                    GG : INTEGER;

              (*PCOMINIT IS CALLED FROM GIVSET WHENEVER
               A PARAMETER IN AN INSTRUMENT MUST BE
               INITIALIZED.IT WILL FIRST LOOK FOR A TIMER
               IN THE CHARACTER LINE IT RECEIVES AND THEN
               CALL 'SETERM' TO COMPLETE THE PROCESS OF
               INITIALIZING THE TERM. THE GLOBAL VARIABLE
               'PARAM' IS USED TO DETERMINED IF UPON MEETING
               AN END OF LINE SYMBOL ';' THE PARAMETER
               BEING PROCESSED IS THE 'P1' TIME DELAY
               PARAMETER. IF SO THE STOP FLAG IS RISEN
               OTHERWISE THE CHARACTER LINE IS REPROCESSED
                RECURSIVELY  WITH  A  POINTER  SET TO THE BEGIN
NING OF THE LINE. BEFORE 'SETERM' IS CALLED
               THE PREVIOUS TERM RECEIVED THROUGH INLINE
               IS DISMEMBERED TO REGAIN ACCESS TO THE
               POINTERS THEY USED.*)

 BEGIN
  WITH INLINE DO
         BEGIN
          WHILE PCLAB(KD) NOT IN (.'|',';'.) DO INCR(KD);
          ITEMREAD(PCLAB,KD,ITEMX);
    CASE ITEMX(1) OF
     '|'   : BEGIN
              ITEMREAD(PCLAB,KD,ITEMX);
              IF ITEMX(1) NOT IN NUMS THEN PTIME := MAX;
               ELSE
                 BEGIN
                 READSTR(ITEMX,ONE,PTIME);
                 TESTDAT.PMAT(PARAM).PTIME := PTIME;
                 ITEMREAD(PCLAB,KD,ITEMX);
                 END;
             IF COM.OPE  NIL THEN
                   DISMOUNT(COM);
             IF COM.S  NIL THEN
                   DISMANTLSTACK(COM.S);
             SETERM(COM,KD,PCLAB,ITEMX);
            END;
  ';'      : IF PARAM = 1 THEN TESTDAT.NODATT
                                                := TRUE;
                          ELSE BEGIN
                               KD := 1;
                               PCOMINIT(INLINE,PCLAB);
                               END;
        : WRITELN (' ERROR');
```

```
                END;
            END
    END;
```

```
PROCEDURE MASKSET (VAR ADRSS : CHARLINE; VAR I : INTEGER;
                   VAR MPASS : MASKNODE);
 VAR M,J,H,L : INTEGER;ITEMM  : ITEM;
        ZZ : SHORT;
     C,D,Z  : SHORT;

             (*MASKSET READS FOR NUMBERS IN THE CHARACTER
              LINE 'ADRSS' SENT BY THE 'SETERM' AND STORES
               THEM IN THE NEW MASKNODE STRUCTURE WITH
               THE SLOPE VALUE IT DERIVES FROM THEM. MASKSET
               LOOKS FOR TWO THREE OR FOUR NUMBERS.
               IT THEN LOOKS FOR THE 'PICK' MODE SYMBOL
               '&'. IF IT IS FOUND THE PICKNODE IS INITIALIZED
               THE NUMBER OF ELEMENTS IN THE FOLLOWING LIST
               OF ITEMS IS COUNTED AND AN AVERAGE RANGE IS
               CALCULATED.'ST' IS INITIALIZED.*)

BEGIN
   C := 0; D := 0;
   WITH MPASS DO
       BEGIN
         SR := NIL;
         T2 := TESTDAT.PMAT(PARAM).PTIME;
         ITEMREAD(ADRSS,I,ITEMM);
         READSTR(ITEMM,ONE,ZZ);
   A := ZZ;
   ITEMREAD (ADRSS,I,ITEMM);
   READSTR(ITEMM,ONE,ZZ);
   B := ZZ;
   ITEMREAD(ADRSS,I,ITEMM);
   IF ITEMM(1) IN NUMS THEN
        BEGIN
         READSTR(ITEMM,ONE,ZZ);
         C := ZZ;
         ITEMREAD(ADRSS,I,ITEMM);
         IF ITEMM(1) IN NUMS THEN
           BEGIN
            READSTR(ITEMM,ONE,ZZ);
            D := ZZ;
            ITEMREAD(ADRSS,I,ITEMM);
           END;
         ELSE D := C;
        END;
  ELSE BEGIN
       C := A;
       D := B;
        END;
  ASLOPE := (C-A)/T2;
  BSLOPE := (D-B)/T2;
 END;
```

```
IF ITEMM(1) = '&' THEN
   WITH MPASS DO
            BEGIN
               L := I;
               Z := 0;
               M := 2;
               NEW (SR);
               NEW (SR.ST);
               WITH SR DO
               BEGIN
                 INPROGRSS := FALSE;
                 REPEAT
                  ITEMREAD(ADRSS,L,ITEMM);
                  INCR(Z);
                 UNTIL ITEMM(1) IN STOPCHARS;
                 RANGE := 100/(Z-1);
                 WITH ST DO
                   BEGIN
                     LLABEL(1) := '&';
                     NUM  := 1;
                     NEXT := NIL;
                     L := I;
                     REPEAT
                     LLABEL(M) := ADRSS(L);
                     INCR(M);
```

```
                                INCR(L);
                                UNTIL LLABEL(M-1) IN STOPCHARS;
                                I := L - 1;
                             END;
                          END;
                      END;
IF ITEMM(1) IN STOPCHARS THEN I := I-1;
END;
```

```
PROCEDURE LISTPROC ( VAR INSTK : STKPNT;
                     VAR PVAL : SHORT);
VAR ITEMS : ITEM;VAL1 : SHORT; IND : INTEGER;
    INTERM : STKPNT;

        (*LISTPROC RECEIVES A POINTER TO A STACKNODE
          AND READS THE ITEM IN THE CHARLINE OF THE
          STACKNODE WITH THE POINTER KD. IF A NUMBER
          IS FOUND, THIS BECOMES THE FINAL VALUE.
          IF A LISTLABEL IS FOUND, THE POINTERIS
          SENT TO 'PUSHSTACK' TO REPLACE THE DATA AT
          THE TOP OF THE STACK CHAIN AND LISTPROC IS
          RECURSIVELY UNTIL A NUMBER IS FOUND. IF
          A RECYCLING CHARACTER IS FOUND THE POINTER
          IS RESET AT THE BEGINNING OF THE DATA AND LISTPROC
          IS CALLED RECURSIVELY. LISTPROC NEVER LOOKS
          FURTHER THAN THE FISRT NODE POINTED BY INSTK*)

BEGIN
INTERM := INSTK;
 WITH INTERM DO
  BEGIN
  IF LLABEL(1) = '&' THEN VAL1 := -999999;
   ELSE
     BEGIN
```

```
       ITEMREAD(LLABEL,NUM,ITEMS);
       IF ITEMS(1)  IN NUMS THEN
                          READSTR(ITEMS,ONE,VAL1);
       ELSE IF ITEMS(1)  = 'L' THEN
            BEGIN
             PUSHSTACK(INTERM,ITEMS);
             LISTPROC(INTERM,VAL1);
            END;
       ELSE IF ITEMS(1)  IN STOPCHARS THEN
                BEGIN
                  IF LLABEL(1)  = '|' THEN
                     BEGIN
                      NUM := RESETPT;
                      LISTPROC(INTERM,VAL1);
                     END;
                   ELSE
                      BEGIN
                       POPSTACK(INTERM,ITEMS);
                       LISTPROC(INTERM,VAL1);
                      END;
                END
     END
   END;
  PVAL := VAL1;
INSTK := INTERM;
```

```
END;
```

```
 PROCEDURE LISTPICK (VAR RPICK : SHORT;
        VAR VALI : SHORT; VAR SELECT : INTEGER);
  VAR PLACE : SHORT; K :INTEGER;
       DONE : BOOLEAN;

                (*LISTPICK USES THE RANGE 'RPICK' AND THE NUMBER
                  'VALI' TO RETURN AN INTEGER 'SELECT' THAT GIVES
                  THE PLACE IN THE LIST THAT IS WITHIN RANGE OF
                  THE RANDOM NUMBER RECEIVED*)

BEGIN
   K := 1;
   PLACE := RPICK;
    DONE := FALSE;
   WHILE NOT DONE DO
     BEGIN
       IF VALI >= PLACE THEN
        BEGIN
          PLACE := PLACE + RPICK;
          INCR (K);
        END
        ELSE DONE := TRUE;
     END;
   SELECT := K;
  IF SELECT
```

```
END;
```

```
PROCEDURE MASKPROC (VAR MAK : MASKNODE; VAR VALR : SHORT;) ;
 VAR T1 : SHORT;VI,X : INTEGER;
     V,W,VARI :REAL;
         VA : SHORT;
      TERMSET : TERM;
       LAB : ITEM;
       MAK1 : MASKNODE;


                        (*MASKPROC RECEIVES A NODE OF  MASK
                          COORDINATES, CHECKS IF IT IS CURRENTLY
                          OUTPUTING A LIST; IF SO IT GOES DIRECTLY
                          TO 'LISTPROC'. OTHERWISE NEW COORDINATES
                          ARE CALCULATED BY REFERENCE TO THE NEW
                           PARAMETER  PTIME  VALUE  AND  THE  NUMBERS
OBTAINED
                          ARE SENT TO THE RANDOM NUMBER GENERATOR AS
                          MINIMUM AND MAXIMUM. IF THE MASK IS DEDICATED
                          TO SELECTING AN ITEM IN A LIST, THE RANDOM
                          NUMBER IS USED TO SELECT A NEW ITEM THROUGH
                          THE 'LISTPICK' PROCEDURE. OTHERWISE THE
                          NUMBER IS OUTPUT AS THE FINAL VALUE.*)
```

```
BEGIN
 WITH MAK DO
  BEGIN
   IF SR = NIL OR SR.INPROGRSS = FALSE THEN
    BEGIN
     T1 := T2 - (TESTDAT.PMAT(PARAM).PTIME);
     V  := A + (ASLOPE * T1);
     W  := B + (BSLOPE * T1);
     VARI := G05DAF(V,W);
     VA :=  ROUNDTOSHORT(VARI);
    END;

   IF SR  NIL AND SR.INPROGRSS = FALSE THEN
      WITH SR DO
       BEGIN
         LISTPICK(SR.RANGE,VA,VI);
         SR.ST.NUM := 2;
         FOR X := 1 TO VI DO
            ITEMREAD(ST.LLABEL,ST.NUM,LAB);
         IF LAB (1) = 'L' THEN
            BEGIN
            SR.INPROGRSS := TRUE;
            PUSHSTACK(SR.ST,LAB);
            LISTPROC(SR.ST,VA);
```

```
                        END;
                 ELSE
                   IF LAB(1) IN 'NUMS' THEN
                        READSTR(LAB,ONE,VA);
              END;



          ELSE IF SR  NIL AND SR.INPROGRSS = TRUE THEN
                    BEGIN
                    LISTPROC(SR.ST,VA);
                    IF VA = -999999 THEN
                       BEGIN
                       SR.INPROGRSS := FALSE;
                       MAK1 := MAK;
                       MASKPROC(MAK1,VA);
                       MAK := MAK1;
                       VALR := VA;
                        END;
                      END;
            END
VALR := VA;
    END;
```

```
 PROCEDURE GETERMVAL(VAR INTERM : TERMPTR; VAR VALT : SHORT);
 VAR VAL : SHORT; NO : INTEGER;

                       (*GETERMVAL PROCESSES ONE TERM ONLY AND
                         RETURN THE CURRENT VALUE OF THAT TERM TO
                         'TERMSCAN'. DEPENDING ON THE ACTUAL
                          MODE IT CALLS 'LISTPROC'OR 'MASKPROC'*)
BEGIN
  WITH INTERM DO
   CASE ACTUAL OF
    'P'   : VALT := TESTDAT.PSET(PI);
    'C'   : VALT := CONSTA;
    'S'   : LISTPROC(INTERM.S,VALT);
    'R'   : MASKPROC(R,VALT);
      END;
   END;
 FUNCTION TERMSCAN ( TRANSTERM : TERMPTR;) : SHORT;
VAR  VAL1,A1,B1,VAL2,VAL3 : SHORT;
     OPER, CONCP : CHAR;
          CURTERM : TERMPTR;
          DC    : BOOLEAN;

                    (*TERMSCAN PROCESSES A LINKED CHAIN OF
                      TERMS RECURSIVELY. EACH TERM IN THE
                      CHAIN IS CONTRACTED TO 'GETERMVAL' WHILE
                      THE OPERATOR LINK DETERMINES THE OPERATION
                      TO BE PERFORMED BETWEEN THE TERMS. TERMSCAN
                      RETURNS A FINAL PARAMETER VALUE TO THE
                      'GIVSET' PROCEDURE.*)

BEGIN
CURTERM := TRANSTERM;
  GETERMVAL(CURTERM,VAL1);
  IF CURTERM.OPE = NIL THEN TERMSCAN := VAL1;
   ELSE
    BEGIN
       CASE CURTERM.OPE.RATOR OF
          '*' : VAL1 := VAL1 * TERMSCAN(CURTERM.OPE.NEXTERM);
          '+' : VAL1 := VAL1 + TERMSCAN(CURTERM.OPE.NEXTERM);
          '/' : VAL1 := VAL1 / TERMSCAN(CURTERM.OPE.NEXTERM);
          '-' : VAL1 := VAL1 - TERMSCAN(CURTERM.OPE.NEXTERM);
```

```
            END;
    END;
TERMSCAN := VAL1;
END;
```

```
PROCEDURE EVACUATE(VAR OCLAU :OCLOUTPUT;VAR KG : INTEGER;);
VAR IND,A : INTEGER;

                (*AT THIS TIME THE ONLY PURPOSE OF THIS ROUTINE IS
                  TO PRINT THE PARAMETER VALUES OF THE OUTBUFFER
                  ON THE TERMINAL SCREEN. THE OUTBUFFER STRUCTURE
                  IS A RECORD CONTAINING THE TRACK NUMBER OF EACH
                  EVENT. THAT IS PRINTED FOR EACH EVENT. COMMENT :
                  THE 'EVACUATE' ROUTINE WILL EVENTUALLY BE RESPON
                  SIBLE TO FEED A SYNTHESIS PROCEDURE IN SOFTWARE
                  OR HARDWARE. *)
BEGIN
 FOR IND := 1 TO KG DO
  BEGIN
    WRITELN(' EVENT NUMBER: ', EVENT);
    WRITELN(' TRACK NUMBER: ', OCLAU(IND).TRACKNUM);
    FOR A := 1 TO 8 DO
     WRITE(OCLAU(IND).BUFSET(A));
    WRITELN;
    FOR A := 9 TO 15 DO
     WRITE(OCLAU(IND).BUFSET(A));
    WRITELN;
    INCR(EVENT);
   END;
END;
```

```
PROCEDURE OUTPUT(VAR OCLA : OCL);
VAR G,GG,TRACK : INTEGER;

                 (*PROCEDURE  TO  PROPCESS  EVERY  ACTIVE TRACK AND
CHECK
                  THE OLDP1 TO SEE IF IT IS READY TO OUTPUT. IF IT
                  IS, THE PSET OF THAT TRACK WILL BE OUTPUT AND
                  REFILLED BY GIVSET UNTIL THE OLDP1 IS  NOT  EQUAL
TO
                  0. ALL VALUES ENTERED IN THE OUTBUFFER ARE GIVEN
                  A VALUE OF 0. ALL TRACKS ARE PROCESSED TO ENSURE
                  THAT ALL INSTRUMENTS HAVE A NON-ZERO WAITING
                  TIME IN OLDP1. ONCE THIS IS DONE, FINDSMALL IS
                  CALLED TO SELECT THE INSTRUMENT WITH THE SMALLEST
                  WAITING PERIOD AND THE LAST EVENT ENTERED IN THE
                  OUTBUFFER IS GIVEN A WAITING TIME EQUAL TO THAT.
                   THE  'EVACUATE'  ROUTINE IS THEN CALLED TO EMPTY
THE
                  OUTBUFFER. NOTE THAT ONLY THE LAST EVENT  IN  THE
BUF
                  FER HAS A NON-ZERO WAITING TIME(PSET(1)).*)

BEGIN
  GG := 1;
  FOR TRACK := 1 TO OCLA.NOOFTRACK DO
```

```
BEGIN
 IF OCLA.TRARRAY(TRACK)   NIL THEN
  WITH OCLA.TRARRAY(TRACK) DO
    IF TDP   NIL THEN
      BEGIN
       WITH TDP DO
        WHILE OLDP1 = 0 DO
          BEGIN
          OLDP1 := PSET(1);
          OCLA.OUTBUFFER(GG).BUFSET(1) := 0;
          FOR G := 2 TO NOOFPARAM DO
               OCLA.OUTBUFFER(GG).BUFSET(G) :=(PSET(G));
          OCLA.OUTBUFFER(GG).TRACKNUM := TRACK;
          GIVSET(TDP);
          INCR (GG);


          END;
      END;
 END;
 GG := GG - 1;
 LLAST := GG;
 FINDSMALL(OCLA);
 OCLA.OUTBUFFER(GG).BUFSET(1) := OCLA.SMALLP;
 TOTIME := TOTIME + OCLA.SMALLP;
 EVACUATE(OCLA.OUTBUFFER,GG);
```

```
END;
```

```
PROCEDURE COMTIMINIT(VAR TRANSFER : TRACKNODE;
                         VAR INTAKE : CHARLINE;);

VAR ITEMT : ITEM;
     TIMER : SHORT;
      J : INTEGER;

            (*THE COMTIMINIT PROCEDURE READS INTO ONE
              LINE OF THE OCLSOURCE TRACK DATA TRANSFERED
              FROM 'TIMEUPDATE' OR 'SETOCL'. IT USES THE
              KK POINTER IN THE TRACKNODE TO GET FIRST A
              TIMER SYMBOL, THEN A COMMAND. THE COMMAND IS
              EITHER A 'STOP' SYMBOL IN WHICH CASE THE DATA
              POINTER 'TDP'IS DISPOSED, OR AN INSTRUMENT
              NAME. IN THAT CASE A NEW TDP IS ASKED AND
              VARIABLES ARE INITIALIZED, AND THE PSET IS
              FILLED BY A CALL TO GIVSET.*)

BEGIN
WITH TRANSFER DO
  BEGIN
  FOR J := 1 TO 12 DO ITEMT(J) := ' ';
  ITEMREAD(INTAKE,KK,ITEMT);
  IF ITEMT(1) = ';' THEN COM := ';';
  ELSE
```

```
IF ITEMT(1) = '|' THEN
  BEGIN
    IF TDP  NIL THEN
     FOR J := 1 TO 15 DO
      BEGIN
      DISMANTLSTACK(TDP.PMAT(J).COM.S);
      IF TDP.PMAT(J).COM.OPE  NIL THEN
        DISMOUNT(TDP.PMAT(J).COM);
      END;
    DISPOSE(TDP);
    FOR J := 1 TO 12 DO ITEMT(J) := ' ';
    ITEMREAD(INTAKE,KK,ITEMT);
    IF ITEMT(1) IN NUMS THEN
     BEGIN
      READSTR(ITEMT,ONE,TIMER);
      PTIME := TIMER;
      FOR J := 1 TO 12 DO ITEMT(J) := ' ';
      ITEMREAD(INTAKE,KK,ITEMT);
     END;
    ELSE PTIME := 9999999;
    NODAT := FALSE;
    IF ITEMT = 'STOP' THEN
       BEGIN
         COM := ITEMT;
         CLDP1 := PTIME;
         END;
      ELSE
           BEGIN
             NEW(TDP) ;
             CLDP1 := 0;
             COM := ITEMT;
             WITH TDP DO
               BEGIN
                 NODATT := FALSE;
                 NNAME := ITEMT;
                 FOR J := 1 TO 15 DO
                  BEGIN
                   PMAT(J).PTIME := 0;
                   PMAT(J).CCM.S := NIL;
                   PMAT(J).COM.OPE := NIL;
                   PMAT(J).KD := 1;
                  END;
                 GIVSET (TDP);
               END;
           END;
    END;
  ELSE WRITELN (' ',' TIMER EXPECTED. ERROR')
 END;
END;
```

```
PROCEDURE TIMEUPDATE(VAR OCLA : OCL);
VAR X,TRACK : INTEGER;
  YES, YESYES : BOOLEAN;

                (*THE TIMEUPDATE PROCEDURE IS RESPONSIBLE
                  FOR KEEPING TRACK OF ALL TIMING PARAMETERS
                  IN THE EXECUTION OF A ORCHESTRATION. ALL
                  TIMING PARAMETERS ARE SUBTRACTED BY SMALLP
                  IF AN OCL TIMER IS FOUND TO BE 0, THE INITIA
                  LIZATION OF A NEW COMMAND FOR THAT TRACK
                  IS CONTRACTED OUT TO 'COMTIMINIT'. IF THE
                  NEW COMMAND IS FOUND TO BE THE END OF THE TRACK
                  THE TRACK POINTER IS DISPOSED. THE PROCEDURE
                  PERFORMS A TEST TO DETERMINE WHETHER SOME
                  TRACKS ARE STILL PLAYING AND IF NOT SENDS
                  A POSITIVE 'FINISHED' FLAG.
                  THE OLDP1 OF ALL TRACKS IS ALSO SUBTRACTED
                  BY SMALLP, THEREFOR GUARANTEEING THERE WILL
                  ALWAYS AN EVENT WITH AN OLDP1 WITH VALUE=0.
                  THE ONLY EXCEPTION TO THIS IS WHEN THE TRACK
                  SET FOR NEXT OUTPUT IS DISPOSE BY REFERENCE
                  TO THE 'COMTININIT' PROCEDURE. THIS EXCEPTION
                  IS COVERED BY A CHECK OF OLDP1 BEFORE THE
                  NEW COMMAND IS INITIATED. IF IT IS EQUAL
                  TO 0, A NEW REFERENCE TO 'FINDSMALL' IS
```

```
              MADE AND THE LAST TIME DELAY OF THE OUTPUT
              BUFFER IS CORRECTED. IF A TRACK HAS NOT
              BEEN INITIALIZED, ALL TIMING VALUES IN THE
              INSTRUMENT'S 'PMAT' STRUCTURE ARE SUBTRACTED
              BY SMALLP.*)
BEGIN
  YES := TRUE;
  YESYES := FALSE;
  FOR TRACK := 1 TO OCLA.NOOFTRACK DO
   IF OCLA.TRARRAY(TRACK)  NIL THEN
    WITH OCLA.TRARRAY(TRACK) DO
     BEGIN
      PTIME := PTIME - OCLA.SMALLP;
      OLDP1 := OLDP1 - OCLA.SMALLP;
      IF TDP  NIL THEN NODAT := TDP.NODATT;
      IF PTIME
       BEGIN
        IF OLDP1 = 0 THEN
           YESYES := TRUE;
        COMTIMINIT(OCLA.TRARRAY(TRACK),
                   OCLSOURCE.OCLDAT(TRACK));
       END;
      IF COM = ';' THEN
           DISPOSE(OCLA.TRARRAY(TRACK));
      IF COM  ';' AND TDP  NIL THEN
```

```
           WITH TDP DO
             BEGIN
               FOR X := 1 TO 15 DO
                   WITH PMAT(X) DO
                    PTIME := PTIME - OCLA.SMALLP;
             END;
     END;
FOR TRACK := 1 TO OCLA.NOOFTRACK DO
  IF OCLA.TRARRAY(TRACK)  NIL THEN YES := FALSE;
IF YES THEN FINISHED := TRUE;
IF YESYES AND NOT FINISHED THEN
 BEGIN
 FINDSMALL(OCLA);
                OCLA.OUTBUFFER(LLAST).BUFSET(1)                              :=
OCLA.OUTBUFFER(LLAST).BUFSET(1)
   + OCLA.SMALLP;
 TIMEUPDATE(OCLA);
 END;
END;
```

```
PROCEDURE FINDSMALL(VAR OCLA : OCL);
VAR TRACKK : INTEGER;
 FOUND  : BOOLEAN;
                     (*THE FINDSMALL WILL LOOK AT ALL THETRACKS
                       AND FIND THE SMALLEST 'OLDP1' VALUE AND
                       WILL MAKE SMALLP EQUAL TO THAT. THIS
                       MARKS THE AMOUNT TO BE SUBTRACTED FROM
                       ALL TIMING PARAMTERS IN THE TIMEUPDATE
                       PROCEDURE. IT ALSO GIVES THE TIME DELAY
                       VALUE OF THE LAST EVENT ENTERED IN THE
                       OUTPUT BUFFER.*)
BEGIN
 TRACKK := 1;
 WITH OCLA DO
     BEGIN
      WHILE TRARRAY(TRACKK) = NIL DO INCR(TRACKK);
      SMALLP := TRARRAY(TRACKK).OLDP1;
      FOR TRACKK := TRACKK+1 TO NOOFTRACK DO
        IF TRARRAY(TRACKK)  NIL AND TRARRAY(TRACKK).OLDP1


        SMALLP := TRARRAY(TRACKK).OLDP1;
     END;
END;
```

```
PROCEDURE GIVSET(VAR INDAT : DATAPOINTER);
 VAR PVAL : SHORT;
     RETURNP : PCLA;

                        (*THE GIVSET PROCEDURE RECEIVES A POINTER
                          TO A PDAT STRUCTURE. THE TASK OF GIVSET
                          IS TO FILL THE PSET ARRAY WITH THE
                          PARAMETER VALUES OF THE NEXT EVENT
                          FROM THIS INSTUMENT BY USING THE 'PMAT'.
                          UPON ENTERING THE POINTED STRUCTURE IS
                          COPIED TO THE GLOBAL PDAT 'TESTDAT' SO
                          THAT ITS CONTENTS CAN BE MODIFIED BY THE
                          ROUTINES THAT WILL USE THE 'PMAT'DATA FROM
                          GIVSET.
                          EACH POSITION IN THE 'PMAT' ARRAY IS
                          PROCESSED INDIVIDUALY AFTER A TIMING
                          CHECK IS PERFORMED. IF THE TIMER IS
                          SET FOR RENEWAL, 'PCOMINIT' IS CALLED
                          WITH THE TERM  COMMAND  AND  THE  PARAMETER
DATA
                          OF THE INSTRUMENT PREVIOUSLY FETCHED IN THE
                          LIBRARY.'PCOMINIT' IS RESPONSIBLE
                          TO INITIATE THE NEXT COMMAND OR SEND
                          A 'NODAT' FLAG IF THE INSTRUMENT HAS
                          NO MORE NOTES TO PLAY.*)
```

```
BEGIN
 TESTDAT := INDAT;
 SEARCHPCLTREE(INDAT.NNAME,RETURNP);
 IF RETURNP.PCNAME = 'NCT FOUND' THEN
  WRITELN(INDAT.NNAME,' WAS NOT FOUND. IGNORED.')
 ELSE
 WITH TESTDAT DO
 FOR PARAM := 1 TO NOOFPARAM DO
  WITH PMAT(PARAM) DO
   BEGIN
      IF PTIME
           PCCMINIT(PMAT(PARAM),RETURNP.PCL(PARAM));
      IF NODATT = FALSE THEN
       BEGIN
          PASSPOINTER := COM;(*TERMSCAN ACCEPTS A POINTER*)
          PVAL := TERMSCAN(PASSPOINTER);(*VALUE ARRIVES*)
          COM := PASSPOINTER; (*THECOMMAND IS UPDATED*)
          PSET(PARAM) := PVAL;(*VALUE IS STORED*)
       END;
    END;
 INDAT := TESTDAT;
END;
```

```
PROCEDURE SETHEADER;
VAR Y : INTEGER;

                        (*THE SETHEADER PROCEDURE READS THE FIRST
                          GROUP OF LINES IN THE INPUT FILE AND
                          STORES THEM IN THE HEADER LIBRARY. IT
                          WILL LOOP UNTIL THE 'END' SYMBOL IS
                          ENCOUNTERED. ONE LINE CONTAINING THE
                          LIST LABEL IS ALWAYS FOLLOWED BY THE
                          DATA OF THAT LIST. *)
BEGIN
 Y := 0;
 REPEAT
     INCR(Y);
     READLN(DATAFILE,HEADLIST(Y).LLNAME);
     IF HEADLIST(Y).LLNAME  'END'  THEN
      BEGIN
      READLN(DATAFILE,HEADLIST(Y).LDAT);
      WRITELN (' ',HEADLIST(Y).LLNAME,' ',HEADLIST(Y).LDAT);
      END;
 UNTIL HEADLIST(Y).LLNAME = 'END' OR HEADLIST(Y).LLNAME(1)  'L';
 IF Y = 1 THEN HEADLIST(1).LLNAME := 'NOLIST';
END;
```

```
PROCEDURE SETPCLTREE;
  VAR FOUND,DUPC,DONE : BOOLEAN;
    SE,SS : INTEGER;
      INDEX :INTEGER;
      ITEMR : ITEM;
      DUPLIC : CHARLINE;
      PCLRET : PCLA;
      I,SD,  KL   : INTEGER;

                          (*THE SETPCLTREE ROUTINE FILLS THE
                            PCL LIBRARY WITH THE INSTRUMENT
                            DEFINITION DATA FROM THE INPUT FILE.
                            IT LOOKS FOR  CONTROL  WORDS  FOR  INSTRU
MENT DUPLICATION SYMBOLS AND SUBSTITUTES
                            PARAMETER COMMAND STRINGS WHEN GIVEN.
                            ALL LINES FROM THE INPUT FILE ARE READ
                            INTOTHE CHARACTER ARRAY 'DUPLIC'.*)

BEGIN
 DONE := FALSE;
 FOUND := FALSE;
  DUPC := FALSE;
 INDEX := NUMOFINST;

 WHILE NOT DONE AND INDEX
```

```
BEGIN
 WRITELN;
 KL := 1;
 FOR I := 1 TO 100 DO DUPLIC(I) := ' ';
 READLN(DATAFILE,DUPLIC);
 IF DUPLIC = 'FIN;' OR INDEX >= 10  THEN DONE := TRUE
 ELSE
                   (*THE NAME OF THE INSTRUMENT IS READ.
                     THE NEXT LINE IS READ AND PARSED TO LOOK
                     FOR THE 'COPY' WORD. IF IT IS FOUND THE
                     NEXT WORD IS PARSED TO GT THE NAME OF THE
                     INSTRUMENT TO COPY. THE COPY IS MADE FOR
                     EACH PARAMETER COMMAND STRING. THE DUPC
                     FLAG IS TRUE.
                     IF THE COPY SYMBOL IS NOT FOUND, THE
                     LINE BEING PARSED IS STORED AS THE FIRST
                     PARAMETER COMMAND STRING. *)


     BEGIN
      INCR(INDEX);
      WITH PCLLIB(INDEX) DO
        BEGIN
         ITEMREAD(DUPLIC,KL,PCNAME);
         WRITELN('INSTRUMENT : ',PCNAME);
         H := 1;
```

```
    FOR I := 1 TO 100 DO DUPLIC(I) := ' ';
    READLN (DATAFILE,DUPLIC);
    KL := 1 ;
    ITEMREAD(DUPLIC,KL,ITEMR);
    IF ITEMR = 'COPY' THEN
       BEGIN
        WRITE(' COPY OF INSTRUMENT ');
        ITEMREAD(DUPLIC,KL,ITEMR);
        SEARCHPCLTREE(ITEMR,PCLRET);
        IF PCLRET.PCNAME =  'NOT FOUND' THEN
          WRITELN (' INSTRUMENT UNKNOWN');
        ELSE
           BEGIN
             FOR H := 1 TO NOOFPARAM DO
             PCL(H) := PCLRET.PCL(H);
             WRITE(PCLRET.PCNAME);
             WRITELN;
           END;
        DUPC := TRUE;
        FOR I := 1 TO 100 DO DUPLIC(I) := ' ';
        READLN(DATAFILE,DUPLIC);
       END;
ELSE
 DUPC := FALSE;
```

```
            (*THIS LOOP PROCESSES THE NEXT LINE
              UNTIL THE 'END' CONTROL WORD IS MET.
              IF THE DUPC FLAG IS TRUE THEN THE
              LINE IS PARSED TO SEE WHICH PARAMETER IS
              TO BE REPLACED. THE SUBSTITUTION IS MADE.
              IF THE DUPC FLAG IS FALSE, THE INPUT
              LINE IS STORED IN ORDER.*)

H := 1;
WHILE DUPLIC  'END;' AND H
 BEGIN
   IF DUPC = TRUE THEN
     BEGIN
        SS := 1;SE := 1;
        ITEMR := DUPLIC(2);
        WRITE(DUPLIC(1),DUPLIC(2),' REPLACED BY ');
        READSTR(ITEMR,ONE,SD);
        WHILE DUPLIC(SE)  '|' DO INCR(SE);
        FOR I := 1 TO 100 DO PCL(SD,I) := ' ';
        WHILE DUPLIC(SE)  ';' DO
           BEGIN
             PCL(SD,SS) := DUPLIC(SE);
             INCR(SS); INCR(SE);
           END;
```

```
                PCL(SD,SS) := ';';
                WRITELN (PCL(SD));
               END;



           ELSE
               BEGIN
                PCL(H) := DUPLIC;
                WRITELN(PCL(H));
                INCR(H);
               END;
           FOR I := 1 TO 100 DO DUPLIC(I) := ' ';
           READLN(DATAFILE,DUPLIC);
         END;
     END;
    END;
      END;
   NUMOFINST := INDEX;
  END;
```

```
PROCEDURE EXEC;
VAR J,L : INTEGER;

                    (*THE PROCEDURE EXEC WILL LOOP UNTIL THE
                      GLOBAL 'FINISHED' FLAG BECOMES TRUE.
                      THE TWO MAIN STAGES OF A MUSCIL EXECUTION
                      ARE CALLED. OUTPUT WILL SHIP OUT THE
                      PARAMETRIC VALUES OF ALL EVENTS CURRENTLY
                      MARKED FOR OUTPUT AND WILL REFILL FOR THE
                      NEXT ONE, REGARDLESS OF TIMING.
                      TIMEUPDATE WILL TAKE THE TIME DELAY OF THE
                      LAST EVENT SHIPPED OUT AND SUBTRACT IT
                      FROM ALL TIMING PARAMETERS IN ALL
                      TRACKS AND INSTRUMENTS PLAYING. THE VALUE
                      OF 'FINISHED' IS CONTROLLED BY THE
                      'TIMEUPDATE' ROUTINE. IT IS SET TO TRUE
                      WHEN ALL TRACKS ARE EMPTY OF FURTHER
                      DATA.*)

 BEGIN
  FINISHED := FALSE;
  WHILE NOT FINISHED  DO
   BEGIN
    OUTPUT(OCLA);
    TIMEUPDATE(OCLA);
   END;
END;
```

```
PROCEDURE SETOCLTREE;
VAR INDEX,A,B,C,D : INTEGER;
    DUPLI : CHARLINE;


                    (*THE SETOCLTREE ROUTINE READS DATA FROM
                      THE GIVEN INTPUT FILE OR FROM THE
                      TERMINAL. ALL DATA IS READ INTO THE
                       CHARACTER  ARRAY  'DUPLI'  WHICH  IS  EXA
MINED FOR CONTROL WORDS. WHEN A NEW
                      ORCHESTRATION IS ENCOUNTERED, A NEW
                      POINTER IS INITIALIZED.
                      CONTROL WORDS ARE 'END' INDICATING THE
                      THE END OF AN ORCHESTRATION SCORE,'FIN'
                      INDICATING THE LAST ORCHESTRATION HAS JUST
                      BEEN INPUT. *)

BEGIN
INDEX := 0;
A := 1;
FOR D := 1 TO 100 DO DUPLI(D) := ' ';
READLN (DATAFILE,DUPLI);
WHILE DUPLI  'FIN;' AND INDEX
 BEGIN
   INCR(INDEX);
   NEW(OCLLIB(INDEX));
   C := 1;A := 1;
   ITEMREAD(DUPLI,C,OCLLIB(INDEX).OCLNAME);
   WRITELN(' POLY-SCORE : ',DUPLI);
   FOR D := 1 TO 100 DO DUPLI(D) := ' ';
   READ(DATAFILE,DUPLI);
   WHILE DUPLI  'END;' DO
     BEGIN
      OCLLIB(INDEX).OCLDAT(A) := DUPLI;
      WRITELN(' ',DUPLI);
      FOR D := 1 TO 100 DO DUPLI(D) := ' ';
      READLN(DATAFILE,DUPLI);
      INCR(A);
     END;
  OCLLIB(INDEX).OCLDAT(A) := 'END;';
  FOR D := 1 TO 100 DO DUPLI := ' ';
  READLN(DATAFILE,DUPLI);
  WRITELN;
 END;
END;
```

```
PROCEDURE SETOCL(VAR OCLABEL : ITEM);
VAR LL,TRACK,J,I,INDEX,XX :INTEGER;
 FOUND : BOOLEAN;
 ITEMTR : ITEM;

                    (*THE SETOCL ROUTINE SEARCHES THE
                     OCL LIBRARY FOR THE OCL LABEL RECEIVED
                     FROM THE MAIN PROGRAM. IF THE LABEL IS
                     FOUND, THE TRACK DATA IS READ INTO THE
                     PERMANENT 'OCLSOURCE' STRUCTURE WHERE
                     THE EXECUTIVE WILL COME TO GET DATA FOR
                     THE EXECUTION OF THE SCORE.
                     IF IT IS NOT FOUND, INPUT IS ASKED FROM
                     THE KEYBOARD.
                     AFTER INPUT, THE SETOCL ROUTINE COUNTS THE
                      NUMBER  OF  TRACKS  TO  BE  PLAYED AND INITIA
LIZES THE CONTROL DATA OF EACH TRACK.*)


BEGIN
 INDEX := 1;
 LL := 1;
 FOUND := FALSE;
 WRITELN(' SEARCHING FOR FILE ',OCLABEL);
```

```
WHILE  NOT FOUND AND OCLLIB(INDEX)   NIL DO
  BEGIN
   IF OCLLIB(INDEX).OCLNAME = OCLABEL THEN
    BEGIN
     OCLSOURCE.OCLNAME := OCLABEL;
     WHILE OCLLIB(INDEX).OCLDAT(LL)   'END;'DO
       BEGIN
       OCLSOURCE.OCLDAT(LL) := OCLLIB(INDEX).OCLDAT(LL);
       INCR(LL);
       END;
     FOUND := TRUE;
     OCLSOURCE.OCLDAT(LL) := 'END;';
    END;
   ELSE INCR(INDEX);
  END;

IF NOT FOUND THEN
    BEGIN
    WRITELN (' PLEASE INPUT AN ORCHESTRATION ');
      REPEAT
       READ(INPUT,OCLSOURCE.OCLDAT(LL));
       INCR(LL);
      UNTIL OCLSOURCE.OCLDAT(LL-1) = 'END;'
    END;
```

```
 OCIA.NCOFTRACK := LL - 1;
 CCLA.SMALLP := 0;

               (*OCLA DATA STRUCTURE IS INITIALIZED HERE.
                 FOR EACH TRACK A TRACKNODE IS CREATED,
                 A POINTER INTO THE TRACK IS SET TO 1,
                 INITIALIZATION IS CONTRACTED OUT TO THE
                 'COMTIMINIT' ROUTINE.*)

 FOR XX := 1 TO OCLA.NOOFTRACK DO
     BEGIN
      NEW (OCLA.TRARRAY(XX));
      OCLA.TRARRAY(XX).KK := 1;
      FOR LL := 1 TO OCLA.NOOFTRACK DO
          OCLA.TRARRAY(XX).TDP := NIL;
      COMTIMINIT(OCLA.TRARRAY(XX),OCLSOURCE.OCLDAT(XX));
     END;
 EVENT := 1;
END;
```

```
BEGIN (* MAIN*)
  FOR  H  := 1 TO 46 DO  (*THREE STORES FOR POINTERS ARE INITIAL
IZED TO 'NIL'. *)
  BEGIN
   STORESTACK(H) := NIL;
   STOREOPPTR(H) := NIL;
   STORETERMPTR(H) := NIL;
  END;
                             (*A GLOBAL POINTER IS INITIALIZED*)
 NEW(PASSPOINTER);
                            (*A FILE NAME IS ASKED FOR.IF NO FILE
                      IS GIVEN, DATA WILL BE READ FROM INPUT*)
 WRITELN (' TYPE A "MTS" FILENAME OR "CR" FOR KBD ');
 READ (FILENAME);
 IF LINELENGTH (INPUT)
 FILENAME := '*MSOURCE*   ';
 RESET (DATAFILE,FILENAME);
 IF FILENAME(1) = '*' THEN
  WRITELN(' INPUT LINE');
                            (*THE DATA IS READ INTO THE THREE
                              DATA LIBRARIES WITH CALLS TO THREE
                              LIBRARY MANAGEMENT ROUTINES FOR THE
                              THE DIFFERENT MUSCIL FILE SECTIONS.*)
 SETHEADER;
 NUMOFINST := 0;
 SETPCLTREE;
 SETOCLTREE;
                       (*LOOP WILL CYCLE UNTIL THE STOP COMMAND
                         IS RECEIVED. EACH LOOP EXECUTES ONE
                         ORCHESTRATION COMMAND LIST. ONLY ONE
                         OF THESE CAN BE PLAYED AND THE DATA
                          FOR  ITS  EXECUTION  WILL ALWAYS BE CON
TAINED IN "OCLA".*)
 WHILE COMMAND  'STOP' DO
  BEGIN
   WRITELN('EXECUTE SCORE ? ("STOP" TO QUIT)');
   READ(INPUT,COMMAND);
   IF COMMAND = 'STOP' THEN WRITELN (' GOODBYE');
   ELSE
      BEGIN
       TOTIME := 0;   (*SETS TOTAL TIME TO 0*)
       SETOCL(COMMAND);(*INITIALIZES THE OCLA DATA STRUCTURE*)
       EXEC;           (*EXECUTES THE DATA STRUCTURE*)
       WRITELN('EXECUTION  FINISHED.  TOTAL  TIME  =  ',TOTIME,'
SEC.');
      END;
  END;
 END.
```

BIBLIOGRAPHY

ABBOTT C. (1980) 4CED User's Manual.
IRCAM. Paris.

ARFIB D. (1978) Digital Synthesis of Complex Spectra by Means of Multiplication of Non-Linear Distorted Sine-Waves.
Proceedings : 1978 International Computer Music Conference. Evanston Illinois.

BATTIER M. (1979) A Composing Program for a Portable Sound Synthesis System.
The Computer Music Journal, 3-3.

BAYER D. (1977) Real-Time Software for a Digital Music Synthesizer.
The Computer Music Journal, 1-4.

BERG P. (1979) PILE - A Language for Sound Synthesis.
The Computer Music Journal, 3-1.

BERG P., ROWE R. & THERIAULT D. (1980). SSP and Sound Description.
The Computer Music Journal, 4-1.

BUXTON W. (1978). Design Issues in the Foundation of a Computer-Based Tool for Music Composition.
Technical Report CSRG-97. Toronto, University of Toronto.

BUXTON W. & FEDORKOW G. (1977). The Structured Sound Synthesis Project (SSSP): An Introduction. Technical Report CSRG-92, Toronto.

BUXTON W., REEVES W., BEACKER R. & MEZEI L. (1978). The Use of Hierarchy and Instance in a Data Structure for Computer Music.
The Computer Music Journal, 2-4.

BUXTON W., SNIDERMAN R., REEVES W., PATEL S. & BEACKER R. (1979).
The Evolution of the SSSP Score Editing Tools.
The Computer Music Journal, 3-4.

BUXTON W. et al. (1980) A Microcomputer-based Conducting System.
The Computer Music Journal, 4-1.

CHARLES D. (1976) Pour les Oiseaux. Entretiens avec John Cage. Belfond, Paris

CHADABE J., (1977) Some Reflections on the Nature of the Landscape within which Computer Music Systems Are Designed.
Computer Music Journal, 1-3.

CHABADE J., MEYERS R. (1978) An Introduction to the PLAY Program.
The Computer Music Journal, 2-1.

CHOWNING J. (1973) The Synthesis of Complex Audio Spectra by Means of Frequency-Modulation.
JAES, 21-7.

GREY J. (1975) An Exploration of Musical Timbre.
CCRMA, Stanford University, Calif.

HILLER L. & ISAACSON L. (1958). Experimental Music.
McGraw-Hill, New York.

HOWE H.S. (1975) Electronic Music Synthesis : Concepts, Facilities and Techniques.
W.W. Norton, New-York.

KOENIG G.M. (1975) Computer Composition.
Institute for Sonology. Utrecht.

LORRAIN D. (1980) A Panoply of Stochastic "Canons".
The Computer Music Journal, 4-1.

LeBRUN M. (1977) Waveshaping Synthesis.
CCRMA, Stanford University, Calif.

MATHEWS M. (1969) The Technology of Computer Music.
Cambridge, MIT Press.

MATHEWS M. & MOORE F.D. (1970) GROOVE - A Program to Compose, Store and Edit Functions of Time.
Comm. ACM 13-12.

McADAMS S. & BREGMAN A., (1979). Hearing Musical Streams.
The Computer Music Journal, 3-4.

MYHILL J. (1979) Controlled Indeterminacy : A First Step Towards a Semi-Stochastic Music Language.
The Computer Music Journal, 3-3.

REEVES W., BUXTON W., PIKE R., BEACKER R.(1978) Ludwig : An Exampl of Interactive Graphics in Score Editor.
Proceedings: 1978 International Computer Music Conference, Evanston.

RISSET J.C.(1981) Musique, Calcul Secret ?.
Unpublished Manuscript, CNRS, Marseilles.

ROADS C.(1979) Grammars as Representations of Music.
The Computer Music Journal, 3-1.

SCHEAFFER P.(1966) Traite des Objets Musicaux.
Editions du Seuil. Paris.

SMITH L. (1972) SCORE : A Musician's Approach to Computer Music.
Journal of the Audio Engineering Society, 20-1.

--------.(1978) SCORE User's Manual.
CCRMA, Stanford University, Calif.

--------.(1978) MUS10 User's Manual.
CCRMA, Stanford University.

TRUAX B.,(1977) The POD System of Interactive Composition Programs.
The Computer Music Journal, 1-3.

--------. (1976) A Communicational Approach to Computer Sound Programs.
Journal of Music Theory, 20-2.

--------. (1978) The Inverse Relationship Between Generality and Strength in Computer Music Programs.
Proceedings: First International Conference of Computer Music. Cambridge, MIT Press.

WESSEL D.(1979) Timbre Space as a Musical Control Structure.
The Computer Music Journal, 2-2.

XENAKIS Y.(1963) Musiques Formelles.
La Revue Musicale, Paris.

----------.(1971) Formalized Music.
Bloomington. Indiana University Press.