COMPUTER SYSTEMS RESEARCH GROUP
UNIVERSITY OF TORONTO

VELUT ARBOR ÆVO

Design Issues in the Foundation of a Computer-Based Tool for Music Composition

by

William Buxton

Technical Report CSRG-97

October 1978

Structured Sound Synthesis Project
Computer Systems Research Group
University of Toronto
Toronto, Ontario
Canada
M5S 1A4

Abstract

This report presents a discussion of issues which arise in the design of computer-based tools for technologically-naive, but application-sophisticated users. The application area of music composition is used as a case study in the discussion. Rather than completed systems, the current work is concerned with the issues in getting started: the issues involved in arriving at a "base" or "foundation" structure on which an accessible, application-sophisticated tool can be built.

Following a discussion of the general problem, the necessity of arriving at a suitable taxonomy of the tasks undertaken by the eventual user of the tool is presented. This is followed by a historical perspective of computer-music. We then present arguments for particular design decisions; namely, an approach to establishing the basis for a tool for computer-assisted composition. The chosen approach is based on a highly interactive system which relies heavily on graphic techniques for the user interface and a digital sound synthesizer for generating sounds in real-time. Examples of such graphical techniques are given, as is the design for a special-purpose digital sound synthesizer which we have developed as part of the Structured Sound Synthesis Project (SSSP) of the University of Toronto.

Central concepts developed include considering composition in terms of four main sub-tasks: definition of a palette of timbres, definition of the pitch/time structure, orchestration, and performance. We stress the importance of allowing flexibility in the order in which tasks are undertaken, allowing the performance of only partially specified scores. Because scores are represented as hierachical structures, any "chunk" may be treated in the same way as a single note. Furthermore, the structure exploits the redundancies occuring in scores which consist of transformed instances of the same basic material. We conclude that -- while much work remains to be done in terms of methodology -- accessible user interfaces can be designed if the base structures are designed so as to closely parallel user cognitive structures.

Preface

The work described in this report has been undertaken as part of the research of the Structured Sound Synthesis Project (SSSP) of the University of Toronto. The SSSP is an interdisciplinary project whose aim is to conduct research into problems and benefits arising from the use of computers in music composition. This research can be considered in terms of two main areas: the investigation of new representations of musical data and processes, and the study of human-machine interacton as it relates to music.

This document is the second in a series of technical documents to be issued by the SSSP. The purpose of these documents is to facilitate access to information concerning the activities and interim results of the project. Because it presents interim results, comments, criticisms, or any other type of feedback is solicited.

Being part of a larger project, it is clear that the work described in this report owes much to the thought and efforts of fellow researchers. We are greatly indebted to the principal investigators of the SSSP: Ron Baecker, Gustav Ciamaga, Les Mezei, and K. C. Smith. The confidence and encouragement which they have shown has been the prime reason behind any success which this report may demonstrate. Since this report is based on my M. Sc. thesis, this is especially true for Professor Mezei (my first reader) and Professor Baecker (my second reader) who have been my main guides through the mysteries of computer science.

Given the technologically-naive state in which I arrived at the University of Toronto, it is clear that a great number of other people contributed to the work developed in this report. Bill Reeves, Greg Hill, Mike Tilson, and Robert Pike have made a great contribution in terms of software. This has been aided by other members of the Dynamic Graphics Project and the members of Professor Baecker's Graduate Seminar in Computer Graphics. In terms of hardware, a great deal is owing to Larry Sasaki and Guy Fedorkow who are largely responsible for any understanding of digital design which we might possess. For actually realizing in hardware my ideas for the digital synthesizer, I must thank Al Fogels and Guy Fedorkow for their designs, and Scott Mc Kenzie for the construction. To the Canadian Electronic Ensemble, Dennis Patrick and Norma Beecroft, we are indebted for their efforts in keeping musical considerations from becoming buried beneath the potentially dominating technology. To the secretarial and editorial skills of Marylee Coombs and proof-reading of David Sherman, I owe a great deal in terms of the readability of this document.

The research of the SSSP is funded by the Humanities and Social Sciences Research Council of Canada (grant S76-0099), while logistic support comes from the Computer Systems Research Group (CSRG) of the University of Toronto. This support is gratefully acknowledged. Finally, I would like to acknowledge the moral support of my immediate family, and the enlightened spirit of team-work and co-operation which has made this work both possible and enjoyable.

Table of Contents

Table of Figures

1. *Introduction*

The aim of this report is to investigate some of the issues which arise during the design of a computerized tool to aid in the composition of music. The intention is not to present a finished design for yet another "ideal" system; rather, we are concerned with more fundamental questions. For example, given the lack of theory regarding user-computer dialogue, given our lack of understanding of the compositional process, and given the changing state of technology, how does one nevertheless approach designing the "foundation" or "base" for a system which is accessible to the computer-naive but application-sophisticated user? When we know that the system specifications are going to change and evolve as our insight into "unknowns" increases, how do we arrive at a decision about where to start, such that the inevitable high level changes will play the least amount of havoc with the lower level (base) structures? Finally, how can we evolve a design which is not the specification for an implementation, but rather a strategy for evolving its evolution?

From the above, it is clear that the issues to be dealt with have implications which extend beyond the current case study dealing with music. Music, however, appears to be a particularly good test area for such a study. Musicians are typically technologically unsophisticated, while the tasks for which they would utilize such a computerized tool are suitably non-trivial. To put it bluntly, if you can provide a usable tool for the average music student, you can do so for anyone. It is the intention of this report to demonstrate a suitable methodology for laying the ground-work for such a system. It is to be hoped that such a demonstration both encourages and aids researchers working in other application areas.

2. *Basic Issues*

2.1. *Introduction*

Our first task is to establish some bounds on the scope of our discussion. Clearly, the key to doing so lies in the basis of our topic: the development of improved computer-based tools in various application areas [1]. From this, addressing the question "improved for whom?" highlights the fact that the prime focus of our discussion must be the eventual user of the system; especially in terms of his access to the potential benefits offered by the tool. Thus, of far more interest is the "handle", rather than functional aspects of any particular tool. The central problem, then, is one of human engineering. Obviously, this is not to imply that functional details of the application area can be ignored. Rather, it is our observation that the "front-end" engineering of systems is too often neglected, thereby causing only a small portion of the system's full benefit to be realized.

In the extreme case, the point can be expressed as a paraphrase of a cliche from metaphysics: "If one creates a computer-based tool (for music composition, for example) with an impeccable theoretical base, if it is not cognitively, physically and economically

[1] We use the term "tool" in the broad sense, meaning a congenial computer-based environment which serves as a useful aid in the undertaking of some complex task.

accessible to the intended beneficiary, to what extent does it exist in his practial reality?" Our intention, then, is to investigate the foundation or basis for such access. In this endeavour, our prime thesis is that the design of an effective tool is contingent upon a detailed analysis of the eventual user and the cognitive structures which he employs in undertaking the tasks for which the tool is designed as an aid. This is not a particularly novel thesis. We cite in particular Miller and Thomas (1977), Meister (1976), J. Martin (1973), and Meister and Rabideau (1965) as references on the topic. Nevertheless, the literature is rather sparse when it comes to documenting and evaluating particular applications taking this approach. Bennet (1972), Davis (1966), Licklider (1968), and T. Martin (1973) present general surveys of the literature. Of the case studies reported, most deal with the restricted domain of avaiation and military applications. Laske (1978) and Vercoe (1975) are two exceptions to this. Both deal with the application of most interest to us: music composition; however, both suffer from a lack of reference to previous work such as that cited above.

When it comes down to developing "real-world" systems, the prime defficiency of the literature is in providing a theoretical base and methodology for design. There are, of course, exceptions. Parsons (1972), for example, reports on several experiments and methodologies to establish such a theoretical base. Such work ranges from having roots in formal mathematical modelling (Sheridan and Ferrell, 1974) to behavioral psychology (Meister, 1976). In the work which follows, we adopt a highly pragmatic approach. We want to build a system today, and not have to wait for the theory to catch up. In addition, through this implementation, we want to contribute towards the evolution of a theory. Therefore -- to the extent possible in this context -- we hope to make some contribution to this seeming gap in the literature. In our attempt, we shall lean heavily on J. Martin (1973) and Miller and Thomas (1977) insofar as general consideration go. From this point of departure, we shall attempt to follow through on their implications in our specific task area.

2.2. *The User: Establishing a Profile*

Given a suitable application for a computer based-tool, our first course of action is to establish who is going to use it. Only then can we address the question as to how he should actually do so. Insofar as it is possible, the point is to establish a profile of the eventual user. In so doing, we must ask questions regarding:

1. his experience with technology
2. his experience in the application area
3. his intelligence, or capacity for learning
4. his motivation; both degree and source
5. the amount of time available for learning vs. production
6. the ability of the user to determine the suitability of the tool to match his expectations
7. the expected work pattern: daily, weekly, steady, bursts, etc.

To each of these questions we must have at least a semblance of an answer before we can continue. We can take the last one as a case in point. The constraints on the user interface are different for the user working with the system daily, than for the user who works in bursts, say every month or so. In the latter case -- which is not so uncommon (such as query systems for upper management, or music systems which we will discuss in detail) -- it is clear that the design must be such that the user does not have to go through an extensive "refresher course" following each layoff period.

For the purpose of this study, it is our intention to focus on one particular type of user. Therefore, we can already respond to the first two questions above. Namely, our interest is in promoting the use of computer techniques in areas outside of the information sciences. Therefore, the user around which we are basing our study is one which is computer-naive, but application-sophisticated. Our concern is to attempt to ensure that the only constraint on his ability to carry out his designated task is his own experience in the application area. Thus, our goal is to evolve an approach to design which will result in a tool which minimizes the burden imposed by the system in the user's planning of strategies and execution of tasks.

2.3. *The Nature of the User-Computer Dialogue*

Our specific concern is with the design of the underlying "base" structure for a particular computer-based tool. However, in order to achieve the flexibility requisite for such a foundation, we must take a look at the nature of the high level structures which it must support. Of central importance among these structures is the nature of the user-machine dialogue.

In investigating the nature of this dialogue, our key concern is the user. At this stage the pragmatics of implementation are of limited concern. Our analysis should be based on the "ideal" situation of unlimited resources. Otherwise, the potential for (planned) upward compatibility is limited. This point is especially true given the current rate of development in technology.

J. Martin (1973) isolates two particular approaches to user-machine dialogue: those which are computer initiated and those which are user initiated. The former, computer-initiated dialogue, is that in which the user is prompted by the computer as to his next step and/or the options currently open to him. So-called "conversational" systems and menu oriented systems fall into this category. On the other hand, in user-initiated dialogue, the user receives no prompting. He must be aware of what commands are available, and be able to structure their order of execution so as to enable him to carry out a particular *task* [2].

Each of these two types of dialogue have particular strengths and weaknesses. Computer-initiated dialogue, for example, poses a far smaller burden on the user's memory than that which is user-initiated. Furthermore, the conversational nature of computer-initiated dialogue implies an interactive form of communication. This in turn provides instantaneous feedback to the user about the response to his previous action. Therefore, besides prompting the next step, it enables the provision of positive or negative feedback to the previous one. Such attributes are clearly desirable in systems designed for the type of user profiled in the previous section.

---

[2] Our definition of "task" comes from Miller (1953) *via* Meister (1976; p. 96): "a group of discriminations, decisions, and effector activities related to each other by temporal proximity, immediate purpose and a common man-machine output."

There are, however, drawbacks. Two, in particular, stand out. First, user-initiated type dialogues are generally more efficient to implement, and -- once the user is familiar with the system -- more efficient to execute a particular task. Secondly, user-initiated type dialogues are usually more general, or flexible, in that the ordering of steps is not fixed. For users utilizing the system daily, a strong case can be made for user-initiated dialogue; however, for beginner or casual users, the opposite is true. Practically speaking, some combination of the two is generally most appropriate. The problem, then, is to arrive at a suitable balance.

2.4. *Strength vs. Generality*

The main issue raised in considering the alternative forms of dialogue can be reduced to the question of the tradeoff between strength and specificity *vs.* weakness and generality [3]. Essentially, we can view computer-initiated dialogue as strong, but limited by its specificity. On the other hand, user-initiated dialogues are less limited (i.e., more general), but weaker due to the added burden on the user. The problem can be viewed then as how to exploit this strength while not sacrificing important generality, or unneccessarily limiting the (musical) potential of the system. The result will be a system which is more accessible to the user, more reliable (since better understood), and easier to implement.

It is in attempting to distribute functions between the different forms of dialogue that we again see the necessity of breaking the user's problem space into a suitable taxonomy of tasks, and viewing the user's behaviour within this context. The benefits are, first, that we can obtain appropriate groupings of commands within computer-initiated structures. Second, where user-initiated dialogue is used, we can structure commands so as to parallel the user's cognitive structure; that is, commands will "fit" the user's normal way of thinking or working, and we can therefore minimize the added burden on the user's memory resulting from this type of dialogue. Finally, we can design our underlying structures so as to optimize and facilitate the implementation of the different types of dialogue.

There are dangers, however, in taking too rigid a view of the above. To be sure, before even starting, we should be aware that there is no "correct" analysis. Opinions will differ among users, or even from day-to-day (as experience develops) with any single user. Furthermore, we should be aware that any implementation will influence the user's results. This is true both from the point of view of the external representation used (for example, form of music notation), or the access of particular operators (e.g., even if all operators are there, the relative "ease" of expressing certain things will affect their use; i.e., the path of least resistance).

Dealing with these dangers reverts back to the necessity of establishing an adequate profile of the prospective user. Furthermore, we must not only consider his situation when starting with the system, but how demands and "profile" change with experience. This is a point which will be dealt with in a later section of this chapter.

[3] This is not a new issue and we intend only to touch on it here. Newell (1969) gives a general account, and Truax (in press) a detailed discussion relative to music application.

2.5. *In Defense of 'Dirty' Systems*

While we have stated that the performing of a task analysis is a prerequisite to the evolution of a "good" design, we must confess our inability to provide a rigorous method for its undertaking. The absence of such a theory is one of the points highlighted in Meadow (1970), for example. Laske (1977), however, proposes the adoption of an attitude which may provide a key to developing the basics for such a theory.

Accepting that at the outset we have a very limited understanding of the user's behaviour within the problem space, Laske (1977) proposes a rather straightforward strategy for gleaning at least some additional insight. He suggests that we adopt the attitude that the computer implementation of a tool to undertake a particular task is an explicit model or theory of our current understanding of the cognitive structures utilized in carrying out that task. As a methodology for developing a better model of the task (and hence a better tool), he advocates the observation and interrogation of "guinea pig" users of the system. Analysis of the results of observation are then fed back into an improved model (viz., implementation), and then tested for validity. The formulation as presented by Laske is rather complex for a rather simple notion: that successive iterations of a system -- with testing -- will result not only in a better system, but better insight into the problem space which it encompasses. The validity of this assertion has proven true in a number of instances in our research (Buxton and Fedorkow, 1978). Even in the simplest of cases, such as designing the layout of a particular graphics menu, it has taken several iterations to arrive at an "optimal" configuration. This is true even when following guidelines such as those provided in Van Cott and Kinkade (1972).

The problem is, of course, the time and expense involved in carrying out such an iterative approach. The point that we wish to make is that, nevertheless, it should play a part in systems design (Baecker, personal communication). Furthermore, we are convinced that with well planned data, hardware, and control structures (i.e., the foundation whose design is our main concern), the economics of such testing are feasible in many cases. Too often too much emphasis is placed on programming "correct" systems. Our argument is that we should design our underlying structures so as to facilitate the "quick-and-dirty" implementation of prototype systems (analogous to breadboarding in electronics) to perform preliminary testing of user response.

The explicit adaptation of this approach can go a long way towards overcoming our own knowledge gaps which exist at the start of a project, when -- nevertheless -- design decisions must be made. Furthermore, by adopting this approach in designing the underlying structures we see that certain progress is made in obtaining the flexibility to accommodate the changes which inevitably occur in the future.

2.6. *The Evolving Needs of the User*

2.6.1. *Introduction*

During our discussion of the strength *vs.* generality issue, it was pointed out that the form of dialogue which is most appropriate for the novice user is not necessarily the most efficient for those who are more experienced. In this section we intend to go a bit deeper into this question and attempt to establish certain principles concerning the user's changing needs as he becomes more experienced with the system.

To begin with, let us point out one attribute of our selected application area, music, which makes it particularly appropriate for such a study. Namely, unlike most applications -- such as banking -- utilization of the system on the part of the potential user community is completely voluntary. That is, assuming that the tool has the potential to serve some useful purpose, a broadly based user community is indicative of an accessible user interface. If the human engineering is poorly thought out, most composers will not make the effort. They simply do not have the time. We are, therefore, provided with a -- not so rigorous -- measure of success.

2.6.2. *The First Encounter*

In general, the most critical period in "hooking" potential users is the first half hour encounter. If after this period they feel comfortable (i.e., not intimidated, nor overwhelmed) in expressing themselves, then an important success in the area of human engineering has been scored. While it may be argued that such rapid integration with a system is unrealistic, preliminary results (Buxton and Fedorkow, 1978) indicate just the opposite. Stated bluntly, a user should simply not require a two week course before being able to work independently with a system.

Significantly, the basis for realizing such lofty aims lies within the nature of computer-based tools. There are two main points to consider and both provide perhaps the main selling point in choosing computer technology over the more conventional analogue approach. The first point has already been alluded to. That is, through computer-initiated dialogue, the tool itself can guide the user's acquisition of the requisite operational skills. Again, we point out the implication of an on-line interactive system with quick and informative response to errors or queries for help or further elaborations.

The second point, related to the first, is the ability to control the sequence of presentation of the system's features. Rather than being confronted (and intimidated) by a maze of knobs and dials, or lists of options, the operations lying beyond the user's current needs are filtered out of his view. He knows there is more there, and is presented with a strategy for probing deeper.

Consider the analogy of an onion. Regardless of how many layers we see or know about, the overall structure or form is always visible. For the various options or decisions in undertaking a particular task, we do not have to rely on rote memory in determining the most appropriate "next step". Rather, since we always have an overall view of the conceptual framework (the entire onion), the next step can be deduced through logic, rather than memory [4]. The former is the old case of not being able to see the forest for

[4] This assumes that the logic of the system design fits the calculus (i.e., cognitive struc-

the trees. The latter reduces the burden on the memory of the user, thereby freeing this critical resource to concentrate on the already demanding application task. As always, the goal is to design the structures such that the user can work with the belief or confidence that he knows what he is doing, that he doesn't have to be paranoid about destroying files or damaging the equipment, and that if he needs help, it is readily available in a clear and concise manner.

A specific example of this type of approach cited by J. Martin is the programming language APL (Iverson, 1962). Without getting into a discussion of the relative merits of APL as a language, it is worthwhile recognizing that a novice can begin working confidently with a sub-set of APL with very little training.

2.6.3. *As Experience Develops*

It is clear that if a user is making relatively heavy use of the tool, he will desire to work in a less verbose, more efficient manner than that employed by the novice. Therefore, we must provide for the possibility of a smooth transition from computer-initiated to user-initiated dialogues, as user experience develops. We must pay particular attention to this point -- the evolving needs of the user -- since his being application-sophisticated implies that he is intelligent, highly motivated, and will probably learn quickly.

To begin with, it is important to consider that the data must be able to be accessed and manipulated using different types of command structures (such as alpha-numeric or graphic), reflecting the demands of the different types of dialogue. The same task will often be able to be undertaken using different routines, each using a different form of dialogue. In this case, the designer may simply choose to give each of these functionally similar commands different names, but this puts a burden on the user to remember a larger number of command names. The system should allow for functionally similar routines to be invoked using the same command name. Which version is executed should be determined by some easily controlled variable set by the system or the user. For example, the system should automatically respond with the graphics version of a command when working at a graphics terminal, whereas the same command typed at a conventional terminal should respond with an alpha-numeric version. The ability to provide such a feature is intimately tied to the operating system on which the commands are implemented. Our demands on the operating system go even further, however. In the first place, the experienced user should be able to branch from one program module to another without going through the intermediate modules (such as menus or monitors) which are often provided to guide the novice user. Secondly, we should take into account that as sophistication develops, the user may want to briefly branch from his current process in order to undertake some secondary task (such as find out the time of day). In this case, the user does not want to go through the trouble of saving his work, exiting the current process, executing "time", returning, and then restoring things to their previous state. Rather, it is clear that the underlying system should allow him to temporarily suspend his current process, execute the new one, and on its completion, resume automatically from where he left off. In applications -- such as music or computer aided design -- where non-linear thinking is commonplace, the

---

tures) of the user. Again, we see the role of the task analysis and its importance.

importance of having the ability to easily provide such features is essential to designing a system which can accommodate the user's preferred strategies. The general point to be made is that the operating system on which a computer-based tool is constructed is a prime determinant in its chances for success.

2.7. *Special Hardware: Godsend or Curse?*

The question of using special hardware in order to solve certain problems in the user-machine interface is one of the more contentious issues to consider (cf. J. Martin, 1973, p. 144). On the front end of the system, the obvious attraction is that one can develop a custom-built "handle" for the tool. Quite apart from the problem of expense, reliability, and exportability, the main difficulty is the fact that what we want from the "handle" after a certain amount of use, is usually very different from when we designed it. Again, we see a conflict between the flexibility to try different approaches, and the question of efficiency.

Considering the front end of systems there are certain approaches which can be taken to avoid many of the pitfalls mentioned. First, it is worth noting that the layout and operation of most instrumentation panels can be implemented in software through the use of well designed interactive graphics (Newman and Sproull, 1973). Through this one often gains most of the benefits of special hardware, while maintaining the requisite flexibility. Furthermore, while it is often claimed that graphic techniques are overly expensive, when balanced against the alternative of special hardware, they generally come out ahead. This can be expected to become even more true given current trends in technology.

The second point is that if special hardware is built, its design should be considered from the viewpoint of developing an efficient transducer whose function can change in different contexts. That is, build in as much generality as possible without compromising the original purpose for which the device was built. Tracker-balls, joy-sticks, and pressure-sensitive devices are examples of transducers which evolved from this approach and can no longer be considered "special-purpose".

Finally, we must admit that in discouraging the use of special hardware, we are in danger of contradicting other design criteria which we have specified. Prime among these is that of demanding a high degree of interaction. In certain applications -- and music, our case study, is one of them -- the complexity of the operation carried out by the tool is such that interaction can generally only be obtained through the use of special hardware (e.g., the digital synthesizer). In this case, a certain amount of compromise must be made and an appropriate decision can only made by careful analysis of both user and application. We shall present an example of such an analysis in Chapter Five.

2.8. *The Physical Environment*

A final point to be made concerns the physical environment in which the user will be working. Too often too little attention is paid to the effect of this parameter on the effectiveness of the tool. To begin with, the work area should be as free from distracting noise and activity as possible. This includes noise generated by air conditioning or machinery normally found in computer rooms. Furthermore, any special requirements of the application area should be taken into account. For example, in a music system it is important that the composer be able to audition his material in an acoustically favourable environment. He should not be distracted by the noise of others, nor others by the noise (hopefully musical) generated by him. Finally, even the hours of access should ideally be made to fit the user's projected work habits. If the user normally works evenings or weekends, the facility should be available.

Each of these issues may seem to be of only marginal importance. It would be interesting to see, however, how the productivity of a system is affected simply by the colour of the walls and the furnishing of the room. More than we ever thought, in all probability.

2.9. *Summary*

Our view of the basis of evolving the underlying (data, control, and hardware) structures for a computer-based tool have been given. Following pointers to relevant literature, certain points were highlighted. In particular, the necessity of establishing a profile of prospective users was stressed. As a case study, we have taken the example of a computer-naive but application-sophisticated user. In terms of data structures, the need for flexibility to accommodate different high level structures was mentioned. Similarly, certain requirements of the system's operating system were identified, which could result in a control structure which was accommodating to the user. Finally, problems and advantages of special hardware were briefly discussed.

Throughout, the discussion focussed on non-task-dependent issues, with a view to increasing the accessibility of computer-based tools to the user. Before continuing with more task-specific considerations, we present a review of the literature in our chosen application area: music composition.

3. *Historical Perspective*

3.1. *Introduction*

This chapter intends to provide a general introduction to computers as they relate to the production of music [5]. The approach taken is that of a general overview. Our goal is to present the conceptual and theoretical background which would enable the reader to evaluate and compare the various systems extant, and provide a context for the

[5] Buxton (1977a) presents an earlier version of this chapter. Buxton (1977b) provides a directory to those active in the field.

discussion in the chapters which follow.

3.2. *Music Systems in General*

The multiplicity of approaches to "computer music" are such that the composer-user is frequently overwhelmed by the diversity. Thus, in order to impose some order on our presentation, we shall commence by establishing certain criteria whereby various systems can be compared. To begin with, our discussion will present the material in terms of two main application areas, the use of the computer in the compositional process, and the generation of acoustical signals. The reader should be aware, however, of the bias implicit in this separation of abstract musical structures on the one hand, and sound on the other. This is a bias which is neither reflected in all of the systems to be discussed, nor is entirely justifiable in terms of music theory. Keeping these misgivings in mind, we use this approach for ease of presentation.

In addition to the above mentioned separation of topics, three other considerations should be introduced in order to facilitate our discussion. These are,

1. What is the theoretical basis or "model" on which the system is founded (implicitly or explicitly), and what are the resulting musical assumptions or restrictions imposed on the user?

2. What is the hardware configuration on which the system is implemented; that is, what equipment is used and how is it set up?

3. What is the mode of man - machine communication; that is, how do the composer and the system interact?

While these criteria are neither mutually exclusive nor all encompassing, they do provide a basis for comparison among systems of interest. We now proceed to discuss these systems according to the two application areas mentioned above, computers and the compositional process, and computer aided sound generation.

3.3. *Computers and Composition*

Historically, there have been two main trends in the use of computers in the compositional process. These can be characterized as those programs which on being initialized, would generate "musical" structures without further intervention by the composer (composing programs), and those which serve as "aids" to the composer in carrying out lower level compositional tasks (computer aided composition). Since each approach gives rise to interesting peculiarities, we will deal with them separately.

3.3.1. *Composing Programs*

Much of the initial use of computers for musical purposes was in the writing of programs which, after being initialized with the appropriate data, would generate a completed musical structure. Early examples of such usage which are of historical importance include, Hiller and Isaacson's work at Illinois (Hiller and Isaacson, 1958, 1959 and Hiller, 1959) which resulted in the *ILLIAC Suite for String Quartet* (1957); the ST programs of Xenakis (Xenakis, 1971), from which the composition *Atrees* (1962) was produced; and Koenig's PROJECTs 1 & 2 which resulted in, for example, *Uebung feur Klavier* (1969).

In each of these systems, it is the embodied model of the compositional process which is of prime importance. In order to be implemented, each demanded that the author first formalize, and then program the "rules" of a particular theory of composition. This is true even if the author is unaware of it. *Every* program for composition embodies a specific set of such rules. Thus, it is the nature of this "theory" and its implications for the user which we shall investigate. While these are the only three systems we will deal with under this heading, it must be realized that many other models have been proposed or implemented for the generation of musical structures. These include models based on linguistics, cybernetics, systems theory and so on. See for example (Clough, 1969), MUSICBOX (Wiggen, 1972), Moorer (1972), and MUSCOMP (Rader, 1977).

The goal in the early experiments of Hiller and Isaacson was to have the computer undertake the composition of quasi-traditional counterpoint. The results of their first four experiments constitute the movements of the *ILLIAC Suite for String Quartet* (1957). The first of these experiments involved the generation of simple diatonic melodies as well as of two and four part polyphony. In the second experiment, four part first species counterpoint was produced. A more modern idiom was chosen for the third experiment. Here, chromatic music based on tone rows was produced. Finally, the fourth experiment involved the production of "Markovian" music, that is, music where the notes are generated randomly, but where the probability of any particular note being chosen is dependent on the last note(s) selected.

Throughout these experiments, the basic technique or "model" used was a generate-and-test, or "Monte-Carlo" (McCracken, 1955), technique. This can be described in terms of three basic steps: initialization, generation and testing. To begin with, the user of this technique must set up a table of "rules" or "conditions" which define which combinations of notes are considered "legal". This constitutes the initialization. Thus, in experiment two, the rules for voice leading, etc. for first species counterpoint were specified. Once these "rule tables" have been initialized, a composition can be begun. The process is as follows: a note is generated at random (the generate step). This note is then tested for acceptability against the "rules" which were specified in the initialization phase (the test step). If it is accepted, it is appended onto the score. Otherwise, a new attempt is made to generate an acceptable note. Thus, via repeated iterations through the generate and test procedures, a composition is gradually built up.

In dealing with the "generate and test" technique, there are certain significant points to be considered. First, it is important to note that the nature of the rules, which is of prime interest from a musical point of view, is completely arbitrary from a technical viewpoint. Thus, different stylistic traits, for example, can be generated simply by having the composer define his own set of rules. This is not, however, as useful a property as might be at first imagined. To begin with, this technique is "left to right". That is, a composition is "through-composed" from start to finish. As a result, there are severe stylistic limitations on the types of musical structures which can be generated. In addition, changing the "rule table" is a non-trivial endeavour, which significantly limits the composer's freedom.

Xenakis refers to his computer generated compositions as "stochastic" music. In realizing these works he makes heavy use of the science of probability and statistics. Generally described, stochastic music implies simply that random variables, selected according to certain probabilities, are utilized in the calculation of a musical structure. In order to get a better feeling for how such calculations function in the ST programs, it would be worthwhile to investigate briefly Xenakis's ideas on the perception of musical structures. These ideas center on the concept that what is of highest musical importance in such structures are the composite "groups" of sounds, rather than the individual sound events. Thus, each "group" of sounds which is perceived as a structural entity can be thought of as a sound "cloud". The speed, colour, density, and shape of this "cloud" then give a means of characterizing the group as a whole. This is preferable to having to describe the cloud note-by-note via its constituent elements.

In adopting a theoretical basis in which the isolated event is secondary to the group, Xenakis then strove to evolve a meta-language for music which reflected this approach. Given his ideas about "clouds" of sound, it is not surprising that he turned to statistics and probability, which are well suited for such description. The essence of the ST program, therefore, is that it enables the user to describe the characteristics of the clouds of sound in a musical structure, using the terminology of statistics and probability. The program then uses "stochastic" procedures to calculate the elements of these clouds according to the user's specifications. The musical implications for the prospective user, therefore, are that he must accept Xenakis' formalization concerning "clouds" of sound, and be prepared to specify his ideas in terms "understandable" by the program. While there are definite limitations imposed on the composer by Xenakis' system, it is one of the few which has resulted in tangible musical results.

The work of Koenig, as illustrated by the programs PROJECT 1 and PROJECT 2, is based on an extension of serial technique which was prevalent in the 1950s. PROJECT 1 (1964) leaves little room for influence by the user. Basically, the same process generates each piece, with only random variations. The program outputs information for manual transcription concerning the following parameters: timbre, rhythm, pitch class, octave register, and dynamics. Each composition thus generated consists of seven "form-sections." The central idea behind the program is a variation between the "periodicity" and "aperiodicity" over each parameter. In terms of PROJECT 1, periodicity implies a sequence of similar values while aperiodicity means dissimilar values. For each parameter there is a scale of seven levels of periodicity. Thus for any particular parameter, each "form-section" has a different degree of periodicity (i.e., one scale degree for each form-section). The sequence in which the degrees of each parameter's scale appear in the "form-section" is random and may be different for each parameter. Thus, PROJECT 1 can be seen as a program which generates compositions according to a very narrowly defined compositional model. Since it was written for Koenig's personal use, this is not a drawback as long as the idea works musically, which it does, in this author's opinion. The biggest problem in this approach, however, is the investment required to produce a program with such limitations.

Based on his experience with PROJECT 1, Koenig attempted to write a compositional program which would be of general application. The result was PROJECT 2. Basically, the attempt in PROJECT 2 is to enable the user to specify the compositional rules whereby each of the various parameter values are selected throughout the piece. Principles such as "aleatoric" (uniformly random), "series", "ratio" (weighted aleatoric) and "tendency" are available, for example. As a result of this increased flexibility, however, the user is confronted with the somewhat formidable task of understanding the framework of the program in which his input data functions. As well, he must format this data in the appropriate manner. Once this is done, however, the program is able to produce compositions of quite diverse nature. Currently an interactive version of the program is being developed. With it, PROJECT 2 shall not only become more accessible to composers, but will probably fulfil its promise as a tool for research into problems in computer composition.

While some compositions of interest and musical merit have been produced by composing programs such as those discussed, certain questions do arise. The prime one is this, given that decision making is undertaken by the program, to what extent do we possess sufficient knowledge of the musical processes involved to program the knowledge base on which these decisions are made? Each of the projects mentioned represents an attempt to deal with this problem. These efforts have brought to light several previous misconceptions concerning music, just as attempts at automated speech translation did in linguistics. The central issue was the inadequacy of traditional music theory to deal with the "musical process." That is to say, we are severely limited in our current ability to establish a knowledge base for computerized musical decision making. Consequently, those systems which have had musical success, such as those of Xenakis and Koenig, have of necessity been highly specialized in that aspect of music with which they dealt, and therefore have been highly personalized. The writing of a "generalized" system for the composition of music would presume a complete understanding of a "grammar" for music; however, it is doubtful that such an understanding can exist. Thus, while programs such as PROJECT 2 are valuable in exploring a particular theory, at this stage they cannot, by their very nature, be of general application.

In terms of hardware, each of the above mentioned systems was initially implemented on a large-scale computer (the Xenakis program, for example, on an IBM 7090). User-machine interaction involved the preparation of the initial input data, and collection of the final results; there being no composer intervention during the actual realization of a composition due to the automated nature of the programs. In these early systems, the completed composition was output by the computer in the form of alphanumeric symbols. This encoded version would then be manually transcribed into common musical notation (CMN) for performance by traditional musical instruments. It is clear, however, that given appropriate facilities, the musical data could have been output directly in the form of CMN. This could be done without affecting the compositional aspects of the program, while significantly improving the user-machine interface. An example of such a program has in fact been written (Byrd, 1974), which automatically transcribes data produced by Xenakis' program. In addition, it is clear that user-machine communication would be further enhanced if the output of compositional programs could be in the form of an acoustic realization of the completed work. This is supported and demonstrated in the following discussion of computer aided composition and sound synthesis techniques.

3.3.2. *Computer Aided Composition*

The above discussion has brought to light two main problems concerning composing programs. First, it was illustrated that the more knowledge and power that is built into a program, the less general is its musical application (see also Truax, in press). Second, attention was drawn to the limitations in our ability to formalize a basis for musical decision making. As a result of these limitations, in many systems an alternative to composing programs has been taken. This we will call "computer aided composition." We acknowledge that in the general sense, this term could cover the use of any computer system (from sound synthesis to composing programs) in the creation of music; however, for the purpose of this survey a more limited scope is intended.

The key feature distinguishing computer aided composition from composing programs is the degree of interaction between the composer and the program during the realization of a composition. In brief, computer aided composition implies only limited decision making on the part of the computer, which is subject to the composer's intervention and control. Such intervention and control takes the form of a dialogue between the composer and the program, and its nature is extremely important in the evaluation of such systems.

One approach to computer aided composition is illustrated by the SCORE program developed at Stanford University (Smith, 1972). SCORE is primarily a program which enables a user to input, in music oriented terms, the pitch and rhythmic data to a sound synthesis program. The effect, therefore, is to render the technology more accessible to the musician. The user may not only create motives, but easily transpose or otherwise transform them. As well, he may introduce various degrees of randomness over note sequences. All this is accomplished using an easily learned (for musicians) alpha-numeric command language. One of the drawbacks with SCORE, however, is that while the specification of the data to the program is interactive, its acoustic realization is not necessarily so. As was stated above, SCORE is a program to input data to a sound synthesis program; however, the type of synthesis program generally used with SCORE is of the MUSIC V type (see discussion of digital synthesis, below). Unfortunately, programs such as MUSIC V do not easily lend themselves to interactive sound synthesis.

In terms of compositional power, the active role taken by the SCORE program is quite minimal. It is primarily a tool of convenience which has proven its value in actual practice. Its compositional power can be augmented, however, as can that of most sound synthesis programs of the MUSIC V type. This is accomplished by combining the basic program with special compositional subroutines. The use of such programs to generate parts of a composition has been described by Howe (1975a). Typically, the composer would write such subroutines himself, to generate certain parts of the musical data for a composition. The problem is that the composer is then generally obligated to learn computer programming -- a not altogether musical endeavour. Furthermore, in taking this approach, the composer must confront many of the problems discussed in our presentation of composing programs. Nevertheless, if composers were provided with well designed languages (such as Smalltalk: Ingalls, 1977), the potential for exploring this approach to composition would be greatly expanded [6]. Such an

[6] This is particularly true in cases where the music system is implemented in the same

approach to computer composition has proven useful to many composers, such as Howe, and deserved more attention. Nevertheless, for the composer just beginning to utilize computers, it is important to realize that alternatives do exist.

As we saw, one of the key drawbacks of the SCORE system was that it did not necessarily enable the user to interactively audition all or part of the composition in progress. In recent years many researchers have been developing systems which, to varying degrees, overcome this problem. For the most part, rather than the large computer (PDP-10) used by SCORE, these systems have been implemented on less expensive mini-computers. By using such machines, it becomes economically feasible to have the entire system dedicated to serving a single musician-user. Such a dedicated machine thus enables the prompt response (acoustic or otherwise) to the composer's commands. Thus, the tools are provided, throughout the entire compositional process, for the "intervention" and "control" associated with computer assisted composition. Examples of such systems are the NRC system in Ottawa (Pulfer, 1970 and Tanner, 1972), the GROOVE system (Mathews and Moore, 1970), POD (Truax, 1973 and Buxton, 1975), and that of the Experimental Music Studio of M.I.T. (Vercoe, 1975).

Each of the systems mentioned enables the composer to mould his materials in a way somewhat analogous to a sculptor. With the NRC and M.I.T. systems, the composer expresses himself in terms of common musical notation. The GROOVE system, on the other hand, utilizes a convenient form of graphical notation to represent scores as functions over time. To a greater or lesser extent, each system enables the user to deal with groups of sounds at a time, thereby going beyond the note-by-note approach of most sound synthesis programs. In many cases, especially in the POD program, the system augments the simple transformations possible with the NRC system. This program has the ability to generate groups of sound according to criteria similar to those seen in the ST program of Xenakis. Here, groups or structures can then be easily played back, augmented, and modified, thus defining the gradual evolution of a composition.

We see then, that the role of the computer aided system extends beyond that of an, albeit powerful, musical scratch pad, to what could be considered a composer's "assistant". All of this does not come without certain drawbacks, however. As was stated earlier, the main design criteria of such systems is to optimize, on a musical level, the communication between such an assistant and the composer. In so doing, certain sacrifices as regards sound quality or diversity must usually be made. Given a system appropriate to his needs, however, the composer is usually well compensated for such drawbacks, most of which are being overcome by current advances in technology.

In summary, the main attraction of computer aided composition systems is the potential for the user to assimilate the technology so as to serve his musical ends. That is, in working with a program such as POD, the user is freed to concentrate on problems of composition, the design of well formed musical structures, rather than computer programming. Finally, it could be stated that it will most likely be through the experience gained in working with such systems that composers will come to better understand the compositional process, and thereby enable the development of better technological

(congenial) language as provided the user. This is a point made by Baecker (1969), with regard to systems for animation.

tools for their craft (Laske, 1975).

3.4. *Sound Production with the Aid of Computers*

With sound synthesis, one must keep in mind the main task. This is the creation of an electrical signal, which is the analogue of the acoustic pressure function defining the sound to be produced. Simply stated, the goal is to produce a voltage comparable to that output by the stylus of a record player. Once produced, the electrical signal can be fed into an amplification system and converted into sound. In attempting to generate such a signal, however, one runs into several problems. To begin with, the pressure function associated with most sounds of musical interest is extremely complex (Risset and Mathews, 1969, Grey, 1975, and Benade, 1976). Thus, it is necessary to find a less complicated representation of the sound, before such a signal can be generated. Describing and synthesizing sound via its formant structure (as in much speech synthesis), or component sine waves (as in Fourier synthesis) are two examples of such representations or "acoustic models" of sonic phenomena [7].

Assuming the existence of such a model, it then remains to be asked, "how is the model seen by the user?". That is, given that a user wishes to define the characteristics of a sound to be synthesized using that model, what is the "description language" that he must use to do so? Thus, it is important to distinguish between the mathematical model being employed (the "internal representation") and how it is seen from the outside (the "external representation"). In some systems, such as those using Fourier synthesis, there is little difference between the two; however, in cases where the model is very complex, it is clear that some sort of language more suitable to the musician is desirable. The purpose of such a language is to render the acoustic model "transparent" to the user. Through such a language, the composer is able to specify information in terms oriented to his application (music), letting the system translate this data into a form more appropriate to the acoustic model in use. While not directly related to computers, the "Solfege" developed by Pierre Schaeffer (1966) represents an important effort in the development of such a music oriented description language for sound. More recent research, such as that of Kaegi (1973,74; Kaegi and Tempelaars, 1978), is now oriented towards the implementation of such description languages in interactive computer-music systems.

Computers have many advantages over conventional modes of sound synthesis, such as the traditional electronic music studio. Generally stated, the computer is well suited to deal with the complexities involved. In this regard, both its memory and calculating power play an important role. With a suitable computer, one can efficiently simulate and test various acoustic models. The development of digital F.M. by John Chowning of Stanford University (Chowning, 1973), which has had such an impact on electroacoustic music, is a case in point. Furthermore, it is precisely with the computer that we have the flexibility to develop description languages that make the resources of such acoustic models more accessible to the composer.

[7] Regarding such models, the interested reader is referred to the excellent detailed survey in (Moorer, 1977).

Some of the systems discussed below offer extreme flexibility, but often at the expense of increased cost and complexity. Others are easy to work with, but limited in sonic repertoire and quality. Trade-offs must be made, and these are largely user/application dependent. To date, there have been three main approaches to using computers in the sound generating process. These are, digital synthesis, hybrid systems, and mixed digital systems. Each of these approaches is presented below, with appropriate examples.

3.4.1. *Digital Synthesis*

This is the "classical" technique of sound synthesis first developed by Max Mathews of Bell Laboratories. It is the technique used in the MUSIC IV & V programs (Mathews, 1969), and their derivatives, including MUSIC 4B & 4BF by Howe and Winham (Howe, 1975b), and MUSIC 360 (Vercoe, 1971,73). As well, it is used in the system of the C.E.M.A.Mu. (Xenakis), the IRMA system (Clough, 1971), and POD (Truax, 1973). While a complete discussion of digital synthesis is beyond the scope of this paper, the basic concepts are outlined below. For a more detailed treatment, the reader is referred to (Mathews, 1969).

Sound is perceived due to variations in the atmospheric pressure, as sensed by the ear. Each different sound is characterized by a unique pattern of pressure variation. Assuming that for a given sound we knew what this pattern was, we could then generate a sequence of numbers whose magnitude fluctuated in a way analogous to the pressure pattern under consideration. If the variation in the numbers' values is an adequate representation of the desired acoustic pressure variation (what is considered adequate will be discussed below), we can output the "samples" of the number sequence from the computer, through a digital-to-analogue (D-to-A) converter (Kritz, 1975; Freeman, 1977). It is clear, therefore, that the voltage output by the D-to-A converter will then be analogous to the variations of the pressure pattern, just as is the sequence of numbers given as input. This fluctuating voltage can then be fed to an amplification system in the manner already discussed, thereby producing the desired sound.

Inherent in digital synthesis is an important trade-off. Information theory tells us that in order to adequately represent the bandwidth of audio (circa 16 kHz), the minimum number of numerical samples needed to represent one second of sound is 32,000 (Mathews, 1969). Even with the most powerful computers, this factor renders the calculation of all but the shortest and simplest compositions extremely expensive. This is where the nature of the acoustic model used is very important. The MUSIC V class of programs (which dominate the field) utilizes a model which digitally simulates the workings of apparatus found in an electronic music studio. While offering generality and complexity (one can simulate any "idealized" studio set-up with this system), one must pay in terms of long turnaround (i.e., typically a day between the time that data is submitted and the time when acoustic output is returned); furthermore, the complexity of the calculations involved dictate the use of a large general purpose computer, such as the larger models of the I.B.M. 360 series. This implies expense, sharing the system with other users, and generally working in a "batch" (card readers, etc.) environment, none of which is conducive to creative work. On the other hand, the general availability of such computers, the portability of the software, and the generality offered, makes such systems attractive to many users. Furthermore, the replacement of the "batch" approach by timesharing has improved the user-machine interface of some systems, such as that at Stanford University.

One can, however, take an alternate approach. Due to recent technological developments, small low cost (under $20,000.) "mini-computers" are a viable alternative for music systems. While these machines have neither the calculative power nor the memory capacity of their larger brothers, they do make it economically feasible for an entire system to be dedicated to a single musician-user. This makes it possible for the first time to have computer music systems tailor made to meet the composer's needs. In digital synthesis, the price one pays for these advantages is a loss in generality and sound quality; a mini-computer can simply not do as much in as short a time as an I.B.M. 360/195, for example. However, by choosing an acoustic model which is computationally efficient, these drawbacks can be largely overcome, with the added benefit that sounds can be auditioned immediately, in "real-time." Two examples of such systems are those of Truax (1973) and that at the Xerox Research Labs at Palo Alto, California (Saunders, 1977; Kaehler, 1975). Each of these systems is highly interactive, and capable of producing complex sounds with time-varying spectra. The results of such interaction are systems in which the potential for learning is very great.

While the XEROX and POD systems are in some ways limiting, such limitations are largely technical, and are rapidly being overcome by current technology. One should examine carefully their advantages from a musical viewpoint. Both offer a wide palette of timbres through their use of the F.M. technique, and both are easy to learn. Furthermore, one must reconsider the terms in which we mean "loss of generality" for such systems. While the POD system is far less flexible than MUSIC V in its capacity for sound generation, its implementation enables the synthesis portion to be combined with an interactive compositional system, thus offering the composer a complete interactive music package.

In summary, the trade-off with digital synthesis is generality and sound quality *vs.* interaction and cost. The choice between the two is largely dependent on whether the system is composition or research oriented.

3.4.2. *Hybrid Systems*

In hybrid systems, sound production is carried out by analogue generators (oscillators, synthesizers, etc.), rather than by a computer. The computer in this case is used as a device to control the operation of the peripherals. Examples of such systems are PIPER (Gabura and Ciamaga, 1968), GROOVE (Mathews, 1970; Mathews and Moore, 1970), the Yale synthesizer (Friend, 1971), MUSYS (Grogono, 1973) and EMS (Wiggen, 1972).

In using peripheral sound generators, the computational demands are greatly reduced, as compared with digital synthesis. Whereas digital synthesis requires a minimum of 32,000 samples per second, hybrid systems only need approximately 100 samples for each device being controlled. As a result, smaller, and therefore less expensive computers can be used (for example, MUSYS utilizes a PDP 8, and EMS Stockholm a PDP 15). Furthermore, since sound quality is dependent on the quality of the devices being controlled, interactive systems can be implemented without the resultant loss of quality seen in digital synthesis.

By the very nature of hybrid systems, the acoustic model (and description language) typically used is a reflection of the apparatus being controlled. Such is the case with both the EMS and MUSYS systems; for example, in the EMS1 language, one specifies a sound in terms of the connections, settings and timings for the actual apparatus of the analogue studio. The computer then "plays back" the events as specified, to be accepted or modified by the user. The GROOVE system has expanded on this concept by introducing a graphics oriented control language which enables the user to modify the values of previously defined parameters, during playback. The user's role during playback, thereby becomes analogous to the conductor's in orchestral music.

Smaller, portable hybrid systems are now appearing, which are suitable for performance in concert situations. Examples of such systems are the HYBRID IV system of Kobrin (1975; Smith and Kobrin, 1977), and the systems commercially available from Donald Buchla Associates. Whether used in the studio or in concert, the main appeal of hybrid systems is the ability to perform (in real-time) compositions made up of complex control and timing functions, and patching sequences, thereby bypassing the previous dependence on audio tape in auditioning the complete composition; furthermore, the utility of many systems extends beyond this use of the computer as an expanded sequencer, in that the user is able to invoke previously defined compositional procedures during actual performance. Two interesting approaches to this type of system are presented in (Rosenboom, 1975) and (Pinzarrone, 1977).

There are however, several drawbacks to hybrid systems. While the quality of the sound output by an analogue device may be quite high, the stability and accuracy can in no way match that of the digital device. Furthermore, whereas with a system such as MUSIC V, one can hypothetically simulate any number of analogue devices in any configuration, with hybrid systems one is restricted by the number and type of actual devices available.

3.4.3. *Mixed Digital Systems*

Mixed digital systems are those systems in which a computer is used as a control device for a digital sound generator, such as a digital oscillator. Examples of this approach to sound synthesis are: the Dartmouth synthesizer (Alonso, Appleton and Jones, 1975), the system designed for IRCAM (Alles and di Giugno, 1978), that developed at Bell Labs (Alles, 1978), the Samson Box at Stanford (Moorer, 1977), and the SSSP synthesizer (Buxton and Fedorkow, 1978). In addition, there are: the Moore synthesizer at Stanford (Moorer, 1977), the VOCOM system (Zinovieff, 1972), VOSIM (Tempelaars, 1976), the University of Illinois synthesizer (Beauchamp, Pohlman, and Chapman, 1975), the EGG synthesizer (Manthey, 1978), and that described by Chamberlain (1976). This type of system is perhaps the most promising in terms of the future of interactive computer music systems.

The concept of a mixed digital system can be gained by considering that any procedure that can be realized as a program (software), could theoretically also be realized by appropriate apparatus (hardware). That is, one could build a special processor to execute any programmable task. This processor is in turn controlled by the CPU of the main computer. While expensive, in realizing a complex procedure in hardware rather than software, one gains in one area in particular, that of time. Thus, if one developed an especially good -- but time consuming -- paradigm for sound generation using MUSIC

V, for example, the paradigm could then be realized in hardware, enabling it to function in real time. In many cases, the benefits that accrue out-weigh the drawbacks of special purpose hardware (as pointed out in chapter two). Consequently, much research is now being carried out (by Kaegi, for example), to develop models which lend themselves to such implementation.

Mixed digital systems take the best from both worlds. One has the speed (and associated convenience) of an analogue hybrid system, combined with the accuracy and stability of digital synthesis. This is emphasized by the sampling rates of about 50 kHz possible with this technique (Buxton and Fedorkow, 1978), or the ability of the Alles and di Giugno synthesizer to output 32 voices of high quality F.M., in real time. One shortcoming, as with the analogue hybrid systems, is the limitation set by the hardware configuration of the "synthesizer". This factor merely emphasizes the need to evolve an adequate acoustic model before realizing it in hardware. While in many cases the use of mixed-digital systems is the best design choice, the problems of time and expense in developing special-purpose hardware should be kept in mind when making design decisions.

3.5. *Summary*

In summary, we have seen that there exist several different approaches to computer music. The various degrees to which a digital computer can participate in the compositional process has been discussed, together with the various types of such participation. Both composing programs and computer "aided" composition were examined. In addition, we have seen that there exist several approaches to generating sound for musical purposes using a computer. Techniques considered included digital, hybrid, and mixed digital synthesis. It has been shown that the various systems extant can be compared in terms of how acoustic phenomena are represented to the machine and to the user, and furthermore, in terms of the method of obtaining sounds from these representations.

From the above survey, it can be seen that the current trend in computer music is towards systems which are more "accessible" to the composer in the physical, economic, and music-theoretical sense. This is seen, by this author, as a tendency towards small interactive systems. It is felt that with mini-computer based mixed digital systems, coupled with well designed modes of communication (graphic languages, etc.), the full potential of computers will be felt in both the composition and performance of music.

4. *Systems Analysis and Design Decisions*

4.1. *Introduction*

We have presented the basis of our approach (Chapter Two) and established the general context within which we are working (Chapter Three). In this chapter we shall develop the implications of the former on the latter.

In Chapter Two it was argued that the key to the sought-after tool lay in good ergonometrics: tailoring the environment to fit the user's psycho/motor structures. Towards this end, several implications in terms of hardware and software structures were discussed. The central point made, however, was that since we do not possess *a priori* the requisite knowledge to achieve good ergonometrics, we must adopt a strategy for its attainment. The basis proposed for such a strategy was classical problem reduction combined with an iterative approach to understanding the components of the problem space. The process, then, becomes one of analyse, implement, test and observe, re-analyse, re-implement, etc. In terms of the base structure -- our main concern -- it is essential to provide the flexibility to support the successive stages of this process.

At this point, the designer is faced with a dilemma: he needs the base structures in order to gain the desired insights into the user's problem space, yet seemingly needs this knowledge before the base structures can be defined. Aspects of this question have been partially broached in Chapter Two. There it was shown that certain decisions can be made, once the basic design attitude is adopted. The questions of accessibility, interaction, special-purpose hardware, and command structure are cases in point. It is clear, however, that to get beyond a certain point we must take into account problem-area dependent considerations. Given our introduction to the application area in Chapter Three, we will devote the rest of this chapter to the consideration -- and the methodology of consideration -- of such task-dependent issues.

4.2. *Composition as Design*

The basis for the decisions made by the systems designer is an understanding of the user's goals, or tasks, and the strategies which he employs in their attainment. At the outset, such understanding will be limited, but it will -- hopefully -- increase through successive iterations of the system. In terms of music composition, we view these goals and strategies in the context of a design process: the design of "well-formed" sonic structures which one construes as "music". In terms of the tool which we are attempting to develop, the context then becomes one of computer-aided design (CAD).

While not wanting to become submerged in the problems of CAD, there is one issue which we would like to address. This is the question of the balance between accommodating old strategies in the new tool *vs.* the new tool introducing new strategies towards new or old goals. The question posed by the former is, "If we maintain old strategies, why do we need the new tool?", while the alternative question is, "If we adopt new strategies (methods of working, conceptualization, notation, etc.), how do we make them accessible in this new, already foreign, environment?" Not trying to be superficial, we feel that the issue is somewhat of a "red herring" when viewed objectively. Our response fits neatly in with our comments regarding the naive and experienced user in Chapter Two. Briefly stated, we feel that a key design philosophy should be to make the novice user as comfortable and secure as possible right from the start. The key to this is providing him with as much as possible which relates to his previous experience. Within this familiar context, he will quickly feel at home, and consequently open to the introduction of more "idiomatic" uses of the tool. The two approaches are only in conflict when a dogmatic, rigid attitude is taken. The approach should be, "use the familiar to introduce the new". Furthermore, the notion of familiar should extend beyond the domain of music. As will be shown in an example in Chapter

Eight, the use of analogy, or metaphor (based on previous experience), is one of the most powerful pedagogical tools available to us.

4.3. *Music Composition: a basic Task Taxonomy*

There are essentially two levels from which one can approach defining a taxonomy of tasks. Each has its own advantages. The first is general in application, such as that of Berliner et al (1964) shown in Figure 4.1 [8]. The second is application specific, and serves as the main focus of our attention [9].

For our purposes, let us consider "musical design" (viz., composition) in terms of four main sub-tasks. These are:

1. Definition of the palette of timbres to be available. This we call *object definition*, which is analogous to choosing the instruments which are to comprise the composer's orchestra. The main expansion on the analogy is that the composer also has the *option* to "invent" his own instruments.

2. Definition of the pitch-time structure of a composition, a process which we can call *score definition*. In conventional music, this task would be roughly analogous to composing a piano version of a score.

3. The *orchestration* of the "score" (defined in step 2) using the repertoire of instruments, or *objects* (defined in step 1).

4. The *performance* of the material developed thus far, whether an entire (orchestrated or unorchestrated) score, or simply a single note (to audition a particular object, for example) [10].

The simple breakdown of tasks is rather straight-forward and may, therefore, provoke a "so what" response. It is our intention, however, to show that it is precisely in this simplicity that the strength of the analysis lies. This simple breakdown of tasks is the absolute "heart", or "basis", of our approach. In it we have the basis for deriving the foundation of a very flexible, powerful system -- without pushing beyond our current

[8] This we cite second-hand from Meister (1976; pp. 104-106).

[9] It should be noted, however, that tasks isolated by our application-specific analysis -- e.g., "perform" -- can be explicated in terms of those of the general level -- such as "motor processes". We include the taxonomy of Berliner et al. in order to point out that different approaches to task classification exist, as well as for its contribution in reducing the "blurr" that often exists between fundamentally different processes.

[10] We include performance as part of the compositional process based on the opinion that a piece of music is not completed until it is heard. While some theorists would dispute its need, we would argue that composers of conventional music have always had such aural feed-back -- in the mind's ear -- as enabled by a familiarity with the long tradition of western music; a tradition which does not exist for the composer of contemporary music.

Classification of Behaviors

Processes	Activities	Behaviors
Perceptual processes	Searching for and receiving information	Detects
		Inspects
		Observes
		Reads
		Receives
		Scans
		Surveys
	Identifying objects, actions, events	Discriminates
		Identifies
		Locates
Mediational processes	Information processing	Categorizes
		Calculates
		Codes
		Computes
		Interpolates
		Itemizes
		Tabulates
		Translates
	Problem solving and Decision making	Analyzes
		Calculates
		Chooses
		Compares
		Computes
		Estimates
		Plans
Communication processes		Advises
		Answers
		Communicates
		Directs
		Indicates
		Informs
		Instructs
		Requests
		Transmits
Motor processes	Simple/Discrete	Activates
		Closes
		Connects
		Disconnects
		Joins
		Moves
		Presses
		Sets
	Complex/Continuous	Adjusts
		Aligns
		Regulates
		Synchronizes
		Tracks

Figure 4.1: A general taxonomy of tasks as developed by Berliner et al.
(Figure from Meister, 1976; p 104)

(basic) understanding of the problem area. We shall see the consequences of this analysis through the rest of this chapter; first in the broad, and then in the specific sense.

4.4. *Control Structures and User Strategies*

There are two key properties of the breakdown of tasks outlined above. First, each task has an analogy in conventional music, which facilitates conceptualization. Second, while the general tasks/goals are presented, no restrictions have yet been placed on the strategies -- both mode and sequence of operation -- to carry them out. Let us pursue this second point.

Even with our limited understanding of the compositional process, we can state one thing with certainty: different composers work different ways (as does the same composer at different times). Some are primarily concerned with timbre, others with pitch or time structures. Some work "top-down", others "bottom-up". It is clear, therefore, that our base structures must support these different approaches to whatever degree possible. Furthermore -- keeping in mind the "onion" concept from Chapter Two -- if the composer wants to concentrate on one particular aspect of his "design" (for example rhythmic structure), we should be able to free him from worrying about other details secondary to this main concern.

In effect, what the above implies is (a) that there should be no order imposed on the sequence in which the composer undertakes his various tasks, and (b) that there should be alternative methods of undertaking any particular task. The latter is largely a question of supporting different representation for the same data, and allowing for various processes to input into a particular data structure (such as a score). These issues are fairly straight-forward, have been alluded to in Chapter Two, and are demonstrated in Chapter Eight. The question of order of operation, however, requires more investigation.

Allowing the composer to choose the sequence in which he undertakes the tasks outlined goes a long way in meeting the demands of various compositional approaches. For example, allowing all the timbres to be defined before a single note is written, or, the entire score written before any thought is given to timbre; a score orchestrated before the "instruments" (i.e., objects) even exist, and at any time enabling the interim results to be auditioned (the performance task). Furthermore -- in keeping with our notion of "composer as non-linear thinker" -- the above implicitly implies the ability to jump back-and-forth from task to task, even before its completion.

4.5. *Defaults*

The clear implication of the above demands, in terms of the base structures, is that we must be able to handle incompletely specified data. The obvious approach to doing so is to incorporate a method of defaults [11] into the system design. Not so obvious is the enormous power that this affords us above and beyond the capabilities described in the

[11] The substitution -- by the system -- of data not specified by the user.

previous section.

The prime benefit of defaults is that they allow us to ignore details which are of only secondary concern at the moment, as well as provide a simple mechanism for error recovery. This is pedagogically useful in that we are able to "hide" from the novice user details which are beyond his current level of competence. We feel that in too many previous systems the necessity of specifying every last detail (correctly), before being able to assess the musical results is an impediment to composers' access. This situation, which has been accepted as a property of computer music in general (as opposed to a deficiency in particular approaches) is well illustrated in the following quote from Howe (1975b; p. 170):

> The note concept and the unit-generator concept require the user to be absolutely precise with respect to all of the details of the sound qualities in his music. While this is not so unusual for composers who have had some experience with electronic music, it is generally more difficult for composers whose prior experience was restricted to instrumental composition. Many details must be specified that these composers have come to take for granted in instrumental composition, and the terms in which the sounds themselves are described involve new concepts. This is one of the sacrifices made in order to achieve the advantage of complete control over all aspects of the music. Computer sound synthesis is a medium for obtaining what you specify; you get all and only that. At least, one is always in the position of knowing what one has, whether or not that was what one wanted.

While there are some legitimate arguments against the use of defaults, if the system (in particular, through the use of "profile" files) is flexible enough to allow the more experienced user to easily "personalize" the defaults when he is working, these objections are largely overcome [12].

4.6. *Scores*

4.6.1. *Introduction*

Having isolated four basic sub-tasks in the compositional process, we will now examine the nature of the task-specific structures by which they are supported. Our discussion is divided into three sections dealing with scores, objects (timbre), and performance. We shall begin in this section with scores.

While music presents some problems not encountered in other areas of CAD, it also has some attractive features which are equally unique. In particular, the temporal nature of music provides a key to data organization not found, for example, in a CAD system for circuit design. Thus, it is "natural" that our representation of a score follow the form of an ordered sequence of musical events. To this point, there is nothing different

[12] A "profile" file is a user-defined file whose contents are used to set system defaults and initialize various variables in the computing environment.

conceptually from any of the systems described in Chapter Three. Where our approach differs, however, is in what constitutes a music event. This we shall develop in the next two sections of this chapter.

4.6.2. *Hierarchical Representation of Scores*

In Chapter Three it was pointed out that most systems gravitated towards one of two extremes: those which dealt with the score from a note-by-note approach (e.g., Vercoe, 1975b), and those which dealt with the score as a single entity (e.g., Xenakis, 1971). It is obvious, however, that structures falling somewhere between the "note" and "score" level play an important musical role. Therefore, systems which lean towards the "note" and/or "score" level are seen as largely inadequate in dealing with these middle level structures. Truax (1973) recognized this and his POD system was an attempt to deal with the problem. His approach, however, was based on the use of stochastic processes, and therefore assumes other problems of compositional programs detailed in Chapter Three. The problem of dealing with the different structural levels of a composition -- from note to score -- remain largely unresolved.

Two observations concerning the above provide the basis of our approach to the problem. First, what has hitherto been considered two extremes are seen as two instances of the same thing. Both deal with the composition "chunk-by-chunk". The only real difference is the size of the chunk: a note or an entire score. If we could provide a structure through which the composer could cause an operator (e.g., "play", "transpose", etc.) to affect any "chunk" of the composition -- from note to score -- we will have gone a long way in overcoming the problems of previous systems.

The key to allowing this "chunk-by-chunk" addressing lies in our second observation: that the discussion of structural "levels" immediately suggests a hierarchical internal representation for scores. Such a structuring of the data goes a long way in enabling the specification of scope (definition of "chunks") of operators. A "play" command, for example, can affect a terminal node (single note) or some non-terminal (thus causing the sub-tree or "sub-score" below that node to be played). The important point to note is that such a structuring of the data allows any "chunk" of a score to be treated *in exactly the same manner as a single note;* with the same ease and clarity, regardless of "chunk" size!

In suggesting the use of a hierarchical representation, we are immediately in danger of having the reader assume the existence of some particular model which functions as the basis of our approach. One might assume a Shenkerian bias (eg., Smoliar, 1976) while another might feel that we are attempting to use grammars and techniques from linguistics (eg., Winograd, 1968) in our approach to music. It is important to point out that neither of these assumptions is well founded. In the next few sections, we shall present the basis for our structuring of the data, and show that it does not imply the composer having to specify grammars, etc., in order to work efficiently.

4.6.3. *The Musical Event*

In the introduction to our discussion of scores we defined a score as "an ordered sequence of musical events". Central to an understanding of our hierarchic representation of scores is the notion of "musical event" as used in this definition.

What is meant by a "musical event" is quite simple. It is an event which occurs during the course of a composition which has a start-time and an end. Thus, the entire composition constitutes a musical event (the highest level), as does a single note (the lowest level). Similarly, chords, motives, movements, etc., are all musical events. In fact, any of the "chunks" -- as described in the previous section -- can constitute a musical event [13]. Thus, any musical event (e.g., a motif) can be made up of composite musical events (e.g., chords and notes); hence the basis for our hierarchy.

In considering the concept of a musical event, it is important to realize that the starting time of the next event is completely independent of the duration of the current one. Therefore, as we see in Figure 4.2, for example, the same two events (G4 and C5) can occur in sequence (Bar one) or parallel (Bar two), or in some combination of the two (Bar three). Similarly, we see in Figure 4.3, for example, that each of the four parts in a string quartet can be considered as a separate musical event (each made up of events of a lower level).

With the musical events, there are two autonomous notions of time: duration and entry-delay. The first is self-explanatory, and the second is the delay before the onset of the next event in the sequence. In melodic figures the two are equal. In a chord, the entry delay is equal to zero. The important thing to note is that in performance, for example, they can be modified independently or together. Changing both will vary tempo while adjusting the articulation proportionatly. Adjusting duration indepenently of entry-delay will result in a change in the articulation of notes, for example. Thus, there is a great deal of potential for the "conducting" of a score built into the underlying structure.

We can express the notion of musical event as a simple grammar (where Mevent is an abbreviation for musical event): [14]

```
Composition ::= Mevent;
Mevent      ::= Mevent* | Score | note;
Score       ::= Mevent;
note        ::= terminal (i.e., some musical note);
```

Besides the ability to isolate different components of the composition, this structure has the benefit that the tree structure actually represents a "recipe" of how the composition was put together. Thus the additional features of being able to backtrack or

[13] This notion of an event being either a simple sound or a more complex structure is somewhat similar to the use of *sound pattern* (simple) and *gemishes* (complex) in the system of the Institute of Musicology, Arhus, Denmark (Hansen, 1977; Manthey, 1978).

[14] In the grammar, non-terminals begin with an upper-case character.

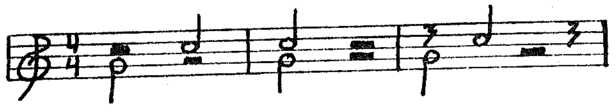


Figure 4.2: Temporal relationships between
simple musical events.
Bar 1: sequence (melodic)
Bar 2: parallel (chord)
Bar 3: mixed

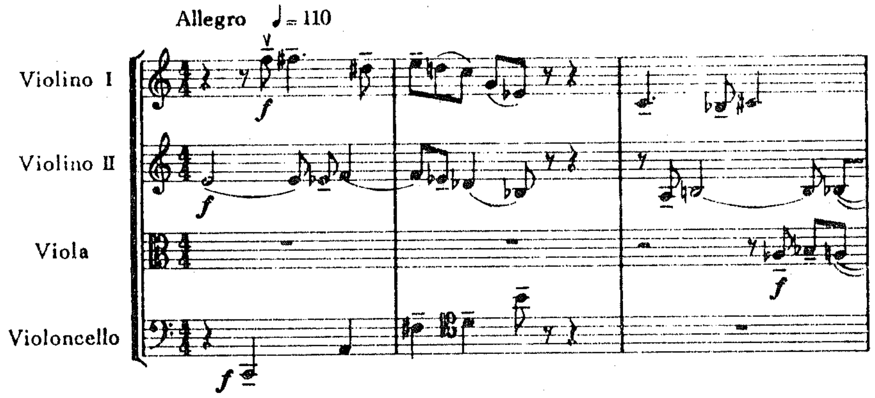


Figure 4.3: High level musical events - an
example. Each line of the quartet
can be considered a single musical
event. The events overlap in a re-
lationship similar to Bars 2 & 3 in
Figure 4.2.
(From Bartok, <u>String Quartet No. 4</u>, First Movement)

reassemble scores are provided. Throughout it should be kept in mind that the common simple list structure used to represent scores is convered by the model: a tree of level one. Therefore, the user has a choice as to his score representation. Complexity is not forced upon him.

4.6.4. *Instantiation*

Our choice of a hierarchic score representation makes possible additional features not yet discussed. Consider, for example, the common case where a composition is made up of certain base material which is then repeated, developed, transposed, etc. In this case, the score could contain several instances of a particular musical event, but where each instance may be transformed in some way. One need only consider one of the examples in the literature of the "theme and variations" form to find a good illustration of this point. In terms of a tree structure, we see that this case could be described as there being more than one instance of a particular sub-tree. Where we can derive power from this observation is in stating that consequently, there should only be one *master-copy* of that sub-tree, and at each *instance* we store only the sub-tree identifier and the transformations to be effected for the particular instance [15].

There are a number of benefits to this approach. First, it is easy to isolate all instances of "motif A", for example. Second, the size of the score is reduced considerably, since only one copy of the motif is saved [16]. Third, it is clear that our file system and data structures must be able to treat any musical event as a free-standing self-contained structures; a sub-score. Therefore, any sub-score can be played, edited, etc., on its own. Most important, any change to the master copy of any sub-score in a composition will be reflected in *every* instance of that structure. Thus, if a re-occurring figure in our composition is an octave jump up, followed by a semi-tone fall, by simply changing the master copy of this figure to a major triad, all instances would be similarly affected *by this one action!*

4.6.5. *Summary*

In the preceeding discussion, an argument has been made for the adoption of a hierarchically based internal representation of scores. Through this approach we can provide the basis for the composer's ability to address himself (and his commands) to the "chunks" of the score with which he is concerned. Furthermore, through the use of instantiation we are able to exploit the redundancies inherent in musical structures and gain savings both in space and ease of operation. Further discussion of score-related issues -- with the required details -- is presented in Chapter Seven. In addition, illustrative examples, particularly of the use of instantiation, are found in Chapter

[15] This notion of instance was developed and used extensively by Sutherland (1965) in his SKETCHPAD system.

[16] This is admittedly at the expense of speed. However, consider that if we do have to do an expansion before the score can be performed, we are still no worse off than the linked list representation of MUSIC V, for example. Furthermore, we still have the hierarchic representation intact, as a master "recipe" enabling backup, transformation, etc.

Eight.

4.7. *Objects & Timbre Definition*

4.7.1. *Introduction*

If we are going to synthesize sounds, we have an obvious interest in being able to control "timbre"; however, the nature of "timbre" for musical purposes is rather elusive. For example, the American Standards Association (1960) states "Timbre is that attribute of auditory sensation in terms of which a listener can judge two sounds similarly presented and having the same loudness and pitch are dissimilar." Traditional explanations (e.g. Helmholtz, 1954) have restricted their description to the physical (viz. acoustical) properties of sounds. Two things are clear, however: that ideally, timbre should be described in the perceptual, rather than acoustical domain; second, timbre is a multi-dimensional attribute of sound, such that the number of dimensions inhibits the understanding and control of the perceived phenomenon. Thus, our prime objective is to establish the underlying structures which will: (a) facilitate the implementation of different high-level external representations of our repertoire of timbres, and (b) support an effective editor for exploring the properties of the multi-dimensional attributes of this repertoire. Throughout, the intention is that initial work at the lower acoustical level will provide insights enabling us to develop a control mechanism functioning at the higher perceptual level. As our insights into representations of timbre improve through experience and experimentation, we are able to refine our external representations accordingly. Note how this is an instance of the iterative appoach proposed in Chapter Two. Here we have a clear case where the requisite knowledge for implementing the "preferred" system does not exist, so we adopt a methodology for its attainment.

In our approach the analogy to the timbre of a musical instrument is an *object* (after Schaeffer, 1966). By our definition, an object is: "a named set of attributes which will result in sounds having different pitches, durations, and amplitudes to be perceived as having the same timbre". In our definition, it is significant that we have stated nothing about the nature of those attributes constituting an object. The notion of an object simply provides a conceptual framework in which the composer can view his activities. All objects have a name and all instances of a particularly named object sound "the same" [17]. Conceptually, this is all that the composer need understand, plus the fact that there is an editor which will aid him in (a) controlling his palette of timbres -- by defining and modifying his own set of objects, and (b) "orchestrating" the notes in a score from this set of objects.

[17] Note that we use the notion of instance here in exactly the same manner as during our discussion of scores. That is, there is only one *master-copy* of any particular object. Any change to that master-copy is therefore reflected in every instance of the object. This provides an effiecient mechanism for refining the definition of a trumpet timbre, for example, or changing all "trumpets" to "flutes".

To the extent discussed thus far, objects have much in common with *instruments* in the MUSIC IV class of programs (Mathews, 1969). Differences will become evident as we progress, but perhaps the most important is the method in which they are defined. MUSIC IV instruments are defined in terms of components called *unit generators*. In order to define an instrument, the user must understand how to combine these unit generators into a meaningful configuration. Although the difficulty of this task has been somewhat reduced by having the unit generators correspond to modules found in the electronic music studio, as well as the provision of a catalogue of basic instruments (Risset, 1969), it is still a significant obstacle in the musician gaining a fluency with the system.

Our approach to the problem is to take a few well-proven configurations of unit generators and "package" them so as to optimize on the ability of the composer to explore their full potential. Clearly this decision relates back to the strength *vs.* generality issue raised in Chapter Two. Our choice to take the more limiting but strong approach is based on our belief that the "real" problems in computer music are not in the area of sound synthesis. We are confident that the research of people such as Moorer (1977) and Le Brun (1977) will help bring an ever-expanding repertoire of computer-based sounds to the repertoire of composers. We are less confident, however, that enough attention is being placed on the development of tools to aid the composer in controlling these sounds in a musical context. In examinining what we consider as being of prime musical importance, we see that it is in relationships developed among sounds, rather than in the intrinsic value of sounds themselves. Therefore, providing the composer with access to "any sound imaginable" is not a prerequisite for a musically useful tool. A reasonably broad palette of musically interesting sound classes -- sounds having time-varying timbre, etc. -- is adequate.

Having adopted this approach, the problem is to select those instruments or *acoustic models* which we will support. In this decision, the prime considerations are: the range of the timbral palette, suitability to efficient implementation, ease of control protocol, and perhaps most of all, how well the model lends itself to the implementation of a user-congenial interface. Moorer (1977) gives a good survey of the alternatives open to us. One thing which becomes clear in examining the different models available is that the high computational bandwidth required for sound synthesis is in conflict with our desire to reserve as much computational overhead as possible for higher level musical processes (such as enabling the user to "conduct" a performance). Therefore -- in spite of our reservations about special-purpose hardware -- it was seen as expedient to go to a mixed-digital system. The alternative hybrid approach was judged inappropriate in light of both our experience with analogue systems (Vink and Buxton, 1974), and the benefits of the mixed-digital approach outlined in Chapter Three: speed, stability, and precission. As a consequence, the constraints on what acoustic models chosen are even more pronounced due to the expense of implementation and the resulting fact that we will have to live with our decision for a long time. The result of our search is the choice of five different acoustic models: *fixed waveform*, *frequency modulation*, *additive synthesis*, *waveshaping*, and *VOSIM*. A description of each of these models, as well as a discussion of why they were chosen, follows below. Additional details regarding their hardware implementation and supporting data structures appear in Chapters Six and Seven, respectively.

4.7.2. *Fixed Waveform*

The fixed waveform mode of timbre specification is the simplest model of sound synthesis. It consists simply of an oscillator whose frequency, amplitude and waveform can be specified by the user. By itself, the technique is rather limited from a musical point of view. While any waveform can be output, the waveform -- and hence timbre -- does not change during output [18]. Nevertheless, the technique serves as the basis for others to be discussed below, and therefore serves a useful pedagogical function.

4.7.3. *Frequency Modulation*

Frequency modulation is a synthesis technique developed by John Chowning (1973) at Stanford University. Digitally, the technique has had a large impact in computer music because it provides a computationaly efficient method of defining and synthesizing sounds having time-varying spectra. The basis of the model is that in modulating the frequency of one oscillator by another, we can exercise control over the frequency and amplitude of the generated sidebands. This control can be exercised through the specification of about six parameters. Furthermore, many of these parameters can vary over time, thereby enabling the control of time-varying spectra. While the frequency modulation model means that one object requires two oscillators, its ability to generate time-varying spectra makes it far more useful musically than the fixed waveform mode.

4.7.4. *Waveshaping*

Like frequency modulation, waveshaping is a technique which enables the synthesis of sounds having complex time-varying spectra, while utilizing a minimum of resources (both in hardware and control). Furthermore, it enriches the timbral palette which can be provided to the user by virtue of the breadth of sounds which it can produce. The basic concepts of waveshaping were developed by Schaefer (1970) and further refined for digital sound synthesis by Le Brun (1977). Essentially, waveshaping synthesizes complex spectra through the control of non-linear distortion. The technique involves taking the output of one oscillator and -- rather than using it as a signal -- using it as an address into a table. The contents of the table entry acessed is treated as a waveform sample, scaled in amplitude, and output as an audio signal. Audio spectrum is then controlled by a combination of: the waveform and amplitude of the original oscillator's output, and the function stored in the table.

4.7.5. *Additive Synthesis*

Additive synthesis (Risset and Mathews, 1969) is one of the best understood methods of defining the acoustical attributes of a sound event. One of the prime advantages of the technique is the existence of working systems to analyze sounds in terms of the model. This gives great flexibility in acoustic and psycho-acoustic research -- as well as music -- through the technique of "synthesis by analysis" (Grey, 1975). This is the process of

[18] Strictly speaking, this is not quite true if one counts changes in amplitude.

analysing sound, then re-synthesizing it with or without effecting transformations on its various attributes (as isolated by the model of analysis). As long as our vocabulary to describe sounds is so limited, the benefit of being able to use sounds from nature as a reference is obvious. The main drawback of the model, however, is that -- compared to frequency modulation, for example -- it requires a rather large amount of information and several oscillators (a critical resource).

Additive synthesis can be understood in terms of the technique of Fourier Analysis. In a simplified description, any complex periodic waveform can be described in terms of a set of constituent partials, or simplified waveforms. In the Fourier series, these partials are all integral multiples of the fundamental. They are sine functions whose frequency, amplitude and phase must be specified. In sounds found in nature, the frequency and amplitude of each of these partials may vary in time, thereby giving the sound it's time-varying timbre. It is exactly this dynamic phenomenon which characterizes natural music sounds and makes their timbre interesting. We must only ensure that our hardware and software structures will support efficient control of these parameters.

4.7.6. *VOSIM*

We have stated above that a prime goal in object definition is to enable the specification of timbre according to a perceptual rather than acoustical model. Towards this end, one model of sound synthesis is particularly appealing. This is the *VOSIM* technique developed by Kaegi and Tempelaars (1978) The attraction of this model is that it enables us to synthesize quasi-speech sounds, and thereby exploit certain findings in psycho-linguistics. Below, we shall outline these properties and then give a brief description of the model which makes their exploitation possible.

The aspect of psycho-linguistics which we wish to exploit is called the "cardinal vowel quadrangle", defined by the linguist Daniel Jones (1956, 1972). Essentially, the cardinal vowel quadrangle -- shown in Figure 4.4 -- is a two-dimensional space, bounded on its four corners by the four prime cardinal vowels: [i] - as in "heat"; [a] - as in "had"; [ɑ] - as in "father"; and [u] - as in "cool". Each point on the surface of the space defined by this quadrangle correlates to a unique vowel sound; furthermore, all vowels in Indo-European languages are contained within the bounds of this surface. The most striking property of the quadrangle, however, is that physical *proximity* in the physical domain equates with subjective *similarity* in the perceptual! Thus, the closer two points are, the more similar are their timbre and vice versa (since vowel quality equates to timbre in musical terms).

To this stage, we have referred to vowels as steady entities defined by points. Clearly, however, a line between points has meaning, and in phonetics is termed a "dipthong". The difference between a point and line is seen in the English words "a" and "eye", respectively. Most musical sounds are "dipthong" in nature; that is, their sound quality varies in time. Thus, not only is the ability to specify timbres by a line encompassed in the model, but also the ability to specify the type of motion along that line (linear, exponential, random, etc). The use of physical gestures -- such as drawing -- is one obvious approach. Finally, so far all discussion has revolved around vowel (viz., quasi-periodic, or pitched) sounds. However, both music and speech include non-periodic sounds, or sounds with noise components. In speech these are the "consonants", and in

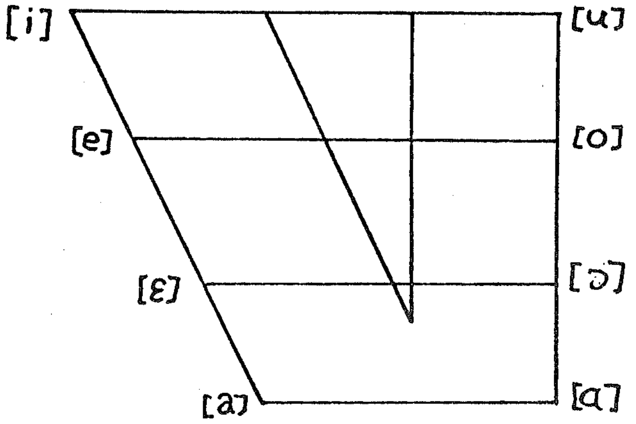


Figure 4.4: The primary cardinal vowels
(After Jones, 1956)

music we have, for example, percussive sounds. Again, this feature is encompassed in the VOSIM model where we have the ability to impose varying degrees of non-periodicity on the vowels.

The basis of the VOSIM model is presented by Kaegi (1973, 74), and Kaegi and Tempelaars (1978). The method involves outputing a stored function (such as a sine squared pulse), as a *one shot*. That is, only one cycle of the respective function is output, the period of which is determined by a user controlled parameter. Another parameter is then provided in order to control the delay before a subsequent pulse is output. Using this model -- shown in Figure 4.5 -- it can be seen that the following points are true. First: the fundamental frequency (and therefore pitch) of the output is determined by the sum (t_t) of the periods of the pulse (t_1) and delay (t_2). Second: the model causes a strong formant to appear at the frequency $1/t_1$ (Kaegi and Tempelaars, 1978). Third: random variation of t_2 will result in sounds having a noisy spectrum. Fourth: all of the variables can be time varying thus enabling one to generate glissandi (t_t), dipthongs (t_1 -- while keeping t_t constant), and degree of noise (deviation of t_2).

By a combination of two oscillators working in parallel in this mode, we are able to utilize Kaegi's method to synthesize most vowel-type sounds. By adding the ability to introduce a degree of random variation (the degree of which is controllable over time), we are also able to synthesize consonant sounds.

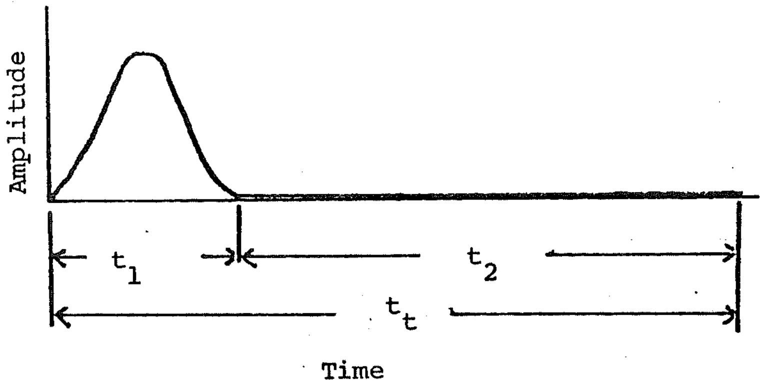


Figure 4.5: The basic VOSIM function
(After Kaegi and Tempelaars, 1978)

4.8. *Performance Factors*

4.8.1. *Introduction*

Our discussion of scores and objects has focussed mainly on the organization of the information structures. In the performance stage we see not only the importance of these structures, but also the hardware transducers used for control.

The first step in our discussion is to clarify what we are referring to when we speak about performance. Mathews has described the use of computers in music performance as falling somewhere between two extremes [19]. These extremes are described in terms of the following two modes:

1. Piano Mode
2. Tape-Recorder Mode

The former of these -- piano mode -- refers to the instrumentalist situation where *all* control information is coming from a human performer. The second mode -- tape-recorder mode -- refers to the situation where all performance information has been predetermined. This mode could just as easily be called "player-piano" mode.

[19] Mathews' comments were made during a talk given while visiting the University of Toronto, June 1978.

The problem now becomes one of determining where one wishes to fit in between these two extremes. In terms of piano mode, it is difficult to imagine how to develop a performance practice and transducers which were not either novelties or throwbacks to previous approaches (such as the synthesizer or electronic organ). On the other hand, experience with tape music has shown the deficiencies of the "tape-recorder" mode, such as the lack of a visual component and inability to vary performances.

Clearly what is needed is an approach which gives the possibility of real-time interpretation (the ability to follow another instrumentalist, or dancer, for example) without having to go to the full extreme of "piano mode".

4.8.2. *Conducting*

In considering the various alternatives, the model of performance which seems most appropriate to our needs is that employed by the GROOVE System (Mathews and Moore, 1970). This is an approach in which the performer's role is analogous to that of a *conductor* rather than that of an *instrumentalist*. In this case the computer does the actual performance -- playing a pre-defined score on the digital sound synthesizer -- while the composer/performer is able to affect aspects of nuance such as articulation, dynamics, balance, tempo, etc. In terms of realizing this objective, we must look at its implications in terms of the base structures.

To begin with, let us consider the score organization as already described. One common function of the conductor is to address himself to one section of the orchestra, such as winds or strings, and make some gesture regarding balance or articulation, for example. Here is a clear example of where our use of a hierarchical representation of scores is advantageous. In this case, the composer need only have the "strings" as a particular sub-score, and he can then -- assuming adequate computational power -- address his commands to that sub-score independent of whatever else is occuring.

In "conducting", we are able to make use of our general-purpose transducers (e.g., the graphics tablet, sliders, etc., described in Chapter Five). No special hardware is needed except for the digital synthesizer (as mentioned previously in this chapter, and described in Chapter Six). It is clear, however, that all "conducting" should not need to be undertaken during one "master" performance. Similarly (and consequently) a particular performance should be able to be saved, played back, and subsequently edited. This places a burden on the real-time capabilities of the computer (saving data) and the data structures. In this regard, we see that heavy use can be made of *stored functions* (described in detail in Chapter Seven). These are simply functions over time which we allow to control any "conductable" parameter in a score. If no function is specified, a default (normally steady state) function is inserted. These functions can be previously defined, or their values can come directly from the input transducers (such as the *sliders)*. In the latter case, the data may be saved at the composer's discretion. Examples of parameters which can be thus controlled include tempo, note duration independently from tempo (hence articulation), dynamics, and localization. Again we emphasize the fact that these parameters can be affected *independently* for each sub-score in a composition, thereby building up a very complex, sophisticated structure.

It is clear that the features desired and demands made up to now make very strong demands on our computing environment and information structures. In the next three chapters we shall present details of a system which meets these demands.

5. *The Computing Environment*

5.1. *Introduction*

In Chapter Two we defined "tool" as "a congenial computer-based environment which serves as a useful aid in the undertaking of some complex task". It is the purpose of this chapter to examine a bit more deeply the basis of this environment. In particular, we will discuss the choice and details of the computing environment in which we have chosen to implement our system. This we shall do in terms of the two main components: hardware and software structures.

5.2. *Hardware*

5.2.1. *Introduction*

A general overview of the physical environment is presented in Figure 5.1. Here the main functional components are shown. These can be subdivided into four main categories: the input/output (I/O) transducers, the host computer, the slave processor, and the digital synthesizer. Due to its special role in the system, the digital synthesizer will be discussed on its own in Chapter Six. The components of the other three categories are discussed below.

5.2.2. *I/O Transducers*

5.2.2.1. *General*

In selecting the I/O transducers to be used by the system, it was consciously attempted to avoid special purpose hardware, according to the ideas expressed in Chapter Two. The transducers used have been chosen so as to minimize as much as possible the physical gestures that the user must make in executing a particular task. In this regard, we have attempted to assemble a configuration in which typing could be kept to a minimum. As a result, the transducers are oriented very heavily towards interactive computer graphics. Again, this is in keeping with -- and a result of -- the importance which we place on the flexibility to utilize different representations of data and processes. Graphics are well suited for the task. We shall now discuss each of the key transducers independently.

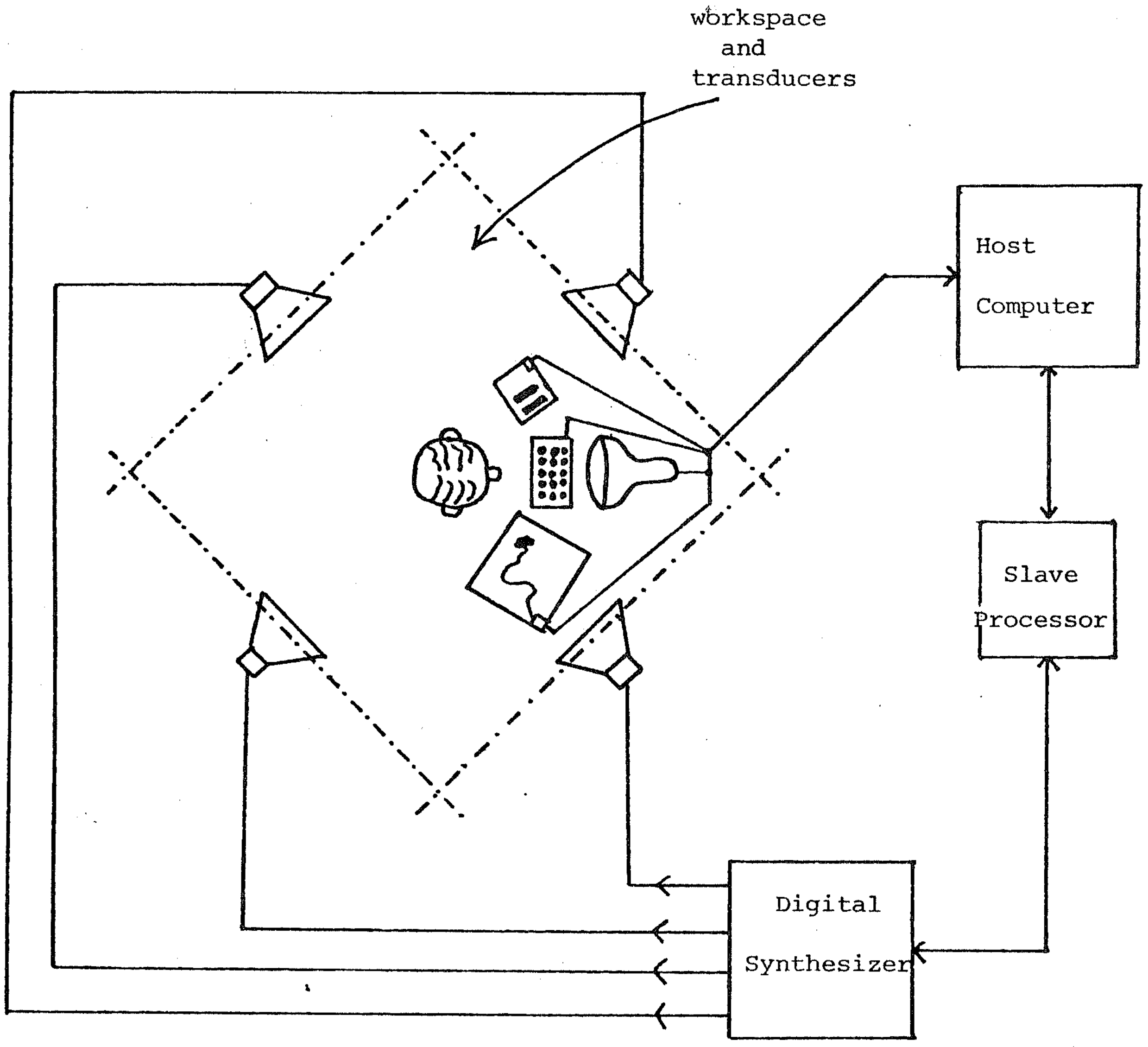


Figure 5.1: Functional block diagram
of physical environment

5.2.2.2. *Graphics Display*

Perhaps one of the most important components in the system is the graphics display. The display used is a fast refresh vector-drawing display produced by the Three Rivers Computer Corporation of Pittsburgh. Use of refresh over the alternative storage or raster-scan (video) techniques was chosen primarily due to the potential for dynamic graphics. One can, for example, animate complex processes, have graphic scores scroll across the screen, or selectively erase, move, or edit individual picture components. With the Three Rivers display and display processor, all of this is possible free of flicker (even with the most complex images) in real-time.

5.2.2.3. *Digitizing Tablet*

Equally important to the display of information graphically is to enable the use of graphical techniques for data entry. Various alternatives are available. These include light-pens, "joy-stick" controllers, tracker-balls, touch sensitive panels, digitizing tablets, and mice (tracker balls mounted in a small housing enabling them to be used much like a digitizing tablet). For our purposes, we have chosen to make use of a digitizing tablet. This device facilitates sketching and pointing, and provides for very high resolution when required.

The tablet consists of a flat panel over which one can move a small device called the "cursor". Most important, the device can be used in combination with the graphics display in such a way that placing the cursor at a particular position on the tablet will cause a graphic ikon called the "tracking cursor" to be positioned at the corresponding position on the surface of the graphics display. Thus, all of the facilities of a light-pen are provided (for example, drawing or pointing), without the associated arm fatigue and visual problems.

Besides the uses already mentioned, the tablet has certain other attractive features. For example, mounted on top of the cursor are four buttons which can be used to initiate different events or indicate responses to questions. One of the values in this is that the hand need not leave the cursor for the typewriter-type keyboard in order to reply to quries requiring a key-stroke response. (Note that of the graphic input devices listed above, only the tablet and the mouse conveniently lend themselves to the incorporation of such buttons.) Finally, coupled with appropriate software, the tablet can function as the input device to a pattern-recognizer which the user can "train" to learn and recognize a set of user-defined shorthand graphics symbols (such as for eight-note, rest, etc.).

5.2.2.4. *Sliders*

The "slider box" is another input device useful in interactive computer graphics. Designed by Fedorkow (see Fedorkow, 1978 for details), the device consists of a box containing three general purpose switches and two infinite sliders. Each slider (shown in Figure 5.2) consists of a touch sensitive continuous plastic belt in combination with a motion detector [20]. The user may touch the exposed part of the slider (circa 13 c.m.)

and move the belt up or down. All of these interactions can be monitored by the host computer and used as control information. For example, in "conducting" a score, one slider can be used for tempo and the other for dynamics. Controlling the sliders with one hand leaves the other free to use the tablet. Thus very high bandwidth is possible with no typing and very little physical effort. Furthermore, even though there are only two physical sliders, they can be used to control several different "virtual" potentiometers which may be displayed on the CRT. In this case, one can rapidly change their context (i.e., reassign a slider to a different potentiometer) simply by pointing at the potentiometer with the cursor and touching one of the sliders. In so doing we see one of the most attractive features of the slider: it is *motion* rather than *position* sensitive. Therefore, unlike a normal potentiometer, there is no need to "null" it, or reset its position when switching it to control graphic fader A from controlling graphic fader B (which may be in a different position).

5.2.2.5. *Typewriter-type Keyboard*

In addition to the above, a normal typewriter-type keyboard is provided. This enables the graphics display to be used as a conventional terminal. When working graphically, however, a system can be designed such that about the only time the user need use the keyboard is in initially calling up a program and when assigning names to files, such as scores.

5.2.2.6. *Loudspeakers*

Obviously, it is important that the user be able to audition his material under the most ideal circumstances possible. In this case, the work station should be in a room designed to the standards of a recording studio control room. While this is seldom possible, every effort should be made to provide good monitoring facilities away from the noise of the computer room, and away from other workers. In many cases, even this is impossible. In such a situation, perhaps one of the best investments to be made is in high quality headphones. In choosing headphones, however, besides the normal criteria of sound quality, comfort, and durability, the ability to exclude external noise should be considered.

5.2.3. *Host Computer*

The key demand on the host computer is the ability to support the demands of the software and peripheral hardware described. In particular, attention must be paid to the ability to meet the real-time demands of the system. Finally, in keeping with our axiom of accessibility, the system should be the smallest, least expensive machine capable of meeting these demands. For the initial implementation, we have chosen to utilize a Digital Equipment Corporation (DEC) PDP-11/45 machine. This processor is complemented by three disk drives, a magnetic tape drive, hard copy output (both

[20] While all electrical components were developed by Fedorkow, we are indebted to Allison Research, Inc., Nashville, Tennessee, for their co-operation in letting us utilize the slider mechanism developed by them.

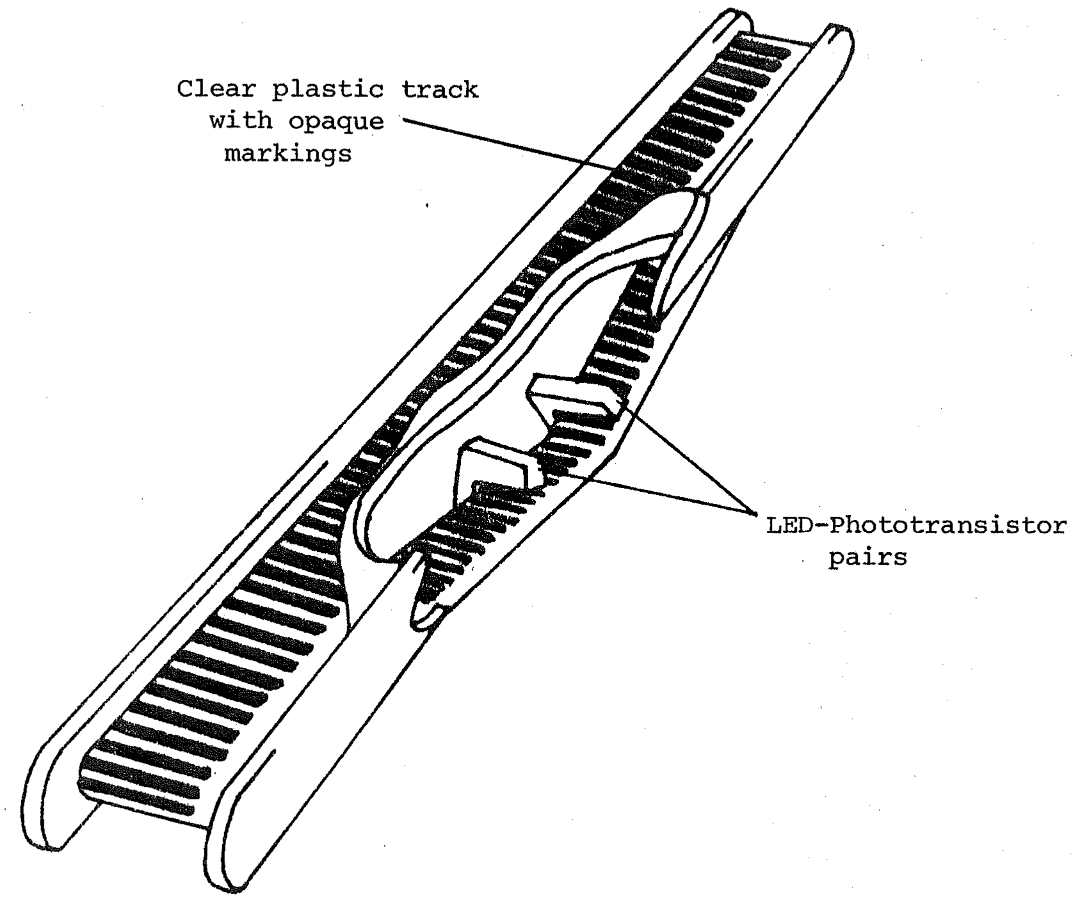


Figure 5.2: The Slider

graphic and alpha-numeric), cache memory (to increase speed of program execution), and full memory complement.

There are various good reasons for choosing this machine, apart from the fact that it was available. First, it supports the UNIX operating system, the attractions of which are discussed below. Second, it provides a powerful tool during the research and development phase, and yet is downward compatible (hardware and software) to a smaller, less expensive, more portable machine, the DEC LSI-11/2. Thus, once the initial research and development phase is completed and a more stable system evolved, accessibility can be greatly increased.

5.2.4. *Slave Processor*

The main function of the slave processor is to enable redundant computation and "mindless crunching" to be farmed out by the host computer. This frees valuable computational overhead for higher level processing. That is, the host machine serves a supervisory role while the slave does the leg work. An example of the type of computation carried out by the slave is the expansion and interpolation of stored functions and the direct servicing of the real-time demands of the synthesizer.

For the function of slave processor, a minimal configuration of a DEC LSI-11/2 was chosen. The processor includes 16K RAM, floating-point option, and interfaces. It is mounted in the chassis of the synthesizer and utilizes the synthesizer's power supplies. While this processor is perhaps more expensive than some micro-processors, it has the attraction that it is able to execute code written in the high level language of the host processor. Code can be written, tested, and debugged on the host and then down-line loaded to the slave. The 16-bit machine is also significantly faster in carrying out arithmetic operations than most of its 8-bit counterparts. Finally, it will retain compatibility even when the host is scaled down to the LSI-11/2.

5.3. *Software*

5.3.1. *Operating System*

One of the main attractions of the PDP-11/45 is that it supports the UNIX operating system (Thompson and Ritchie, 1974). Simply stated, in our opinion, UNIX is the most state-of-the-art operating system available commercially today for use on mini-computers. UNIX is a time-sharing system for which there exist single-user subsets for smaller machines (eg., Lycklama, 1978). Therefore, the same basic system can be used on both the large PDP-11/45 and the future LSI-11/2 host processor. In addition, there exists a UNIX based system for the slave processor (Lycklama & Christensen, 1978), which provides for overall system unity. With the time-sharing version, the attraction is that several users can utilize the music software simultaneously, thereby increasing accessibility. Furthermore, both research and development and application work can be undertaken simultaneously. An impression of the power of UNIX in combination with the PDP-11/45 can be gleaned by considering that the system can support up to eight other users -- doing text editing, graphics, etc. -- and still perform a sixteen part composition in real-time *without resorting to use of the slave processor!* Again, remember that all this is on a time-sharing system running on a mini-computer.

In terms of its other features, UNIX has a very convenient hierarchically structured file system which facilitates the handling of the numerous files used in a composition. Through the feature known as the "shell", often-used sequences of commands can be abbreviated by the user into a single command. In addition, it is the shell which enables us to achieve two of the demands made in Chapter Two. First, to be able to use a control condition (such as type of terminal) to determine which version of a particular command is to be executed. Second, to enable the current process to be suspended while the user temporarily branches off to another. The system not only supports, but is written in one of the most sophisticated high-level languages available on a mini-computer: the language "C", which is discussed below. Partially due to UNIX being written in C, the system can be modified to suit the needs of a particular installation. UNIX is documented (Thompson, 1978; Lions, 1977) and has a good text editor. In summary, it is felt that our development would be nowhere near its current state were it not for the tools provided by this system.

5.3.2. *High Level Language*

As stated above, the language chosen for implementing the base structures is the language "C" (Kernighan and Ritchie, 1978). The attraction of this language is that it supports complex data structures (such as aggregates of data types) coupled with dynamic storage allocation. "C" is a structured programming language with a clear control structure. Since it was written for the PDP-11, it generates very efficient code, which is important, given our real-time constraint. Furthermore, it is well documented (Kernighan and Ritchie, 1978). Again, a great deal of our success has been due to the availability of such tools.

5.3.3. *Graphics Support*

One of the main software packages on our system which serves to illustrate the power of UNIX and the PDP-11/45 in combination with well-chosen peripherals is the graphics support package, GPAC (Reeves, 1976). GPAC is a comprehensive package for supporting interactive computer graphics, including animation. The package includes many high level routines (such as scale, rotate, etc.), the composite effect of which is to free the programmer from low level details in order to concentrate on more important issues. The package is device independent. That is, the same routines will function whether the output device is a CRT (raster-scan, storage, or vector-drawing), or a hard-copy plotter. Second, the facility for interaction is augmented in that GPAC is what is known as "event-driven". That is, programs (with the resulting graphical display of information) can be structured so as to have the flow of control determined by the type and value of various user generated "events". Examples of such events would be: pushing one of the buttons on the cursor, moving the cursor or fader, lifting the cursor from the tablet, the absence of user activity for a given period of time, etc. Finally, GPAC is well documented (Reeves, 1977). In summary, GPAC makes it very easy for the programmer to provide immediate visual feedback to the user following a particular action. That this feedback can be in graphical form greatly increases our bandwidth of communication and improves the potential for our achieving a good user interface.

6. *Digital Synthesizer*

6.1. *Introduction*

The synthesizer is one of the most important components in the system. Generally described, it consists of one "real" oscillator which is time-division-multiplexed into sixteen "virtual" oscillators. The design owes much to the Dartmouth synthesizer (Alonso et al., 1976) as well as the VOSIM oscillator (Tempelaars, in press; Kaegi and Tempelaars, 1978).

The device is essentially a fixed sampling rate, accumulator-type digital oscillator. The sampling rate of each oscillator is 50 kHz, giving a bandwidth of 25 kHz. Dynamic range is well over 60 dB (and should improve with further adjustment) while frequency resolution is approximately .7 Hz (linear scale) over the entire bandwidth. This will shortly be improved to enable resolution of less than 1 "cent" at even extremely low frequencies [21]. The output signal of each oscillator can be fed to one of four analogue output busses which may then either be fed directly to an amplifier, or to a *channel distributor* (Fedorkow, 1978; Fedorkow, Buxton and Smith, 1977). The waveform output by each generator may not only be user defined -- up to 8 waveforms available at one time -- but one may switch waveforms in mid-cycle. This is possible since the 16 oscillators share a 16k buffer of 12-bit words to store waveforms. This 16k of RAM is partitioned into 8 2k blocks, one for each of the 8 possible waveforms defined by the user or system. Apart from this memory configuration, this synthesizer is particularly interesting because the oscillators may be used to generate sounds according to the synthesis modes described in Chapter Four (i.e. fixed waveform, frequency modulation, VOSIM, additive synthesis, and waveshaping modes) [22]. This goes a long way towards a "universal module" -- that is, all modules of a uniform type (with the resulting ease of conceptualization and communication). While this is in direct contrast with analogue synthesizers, a very wide repertoire of sounds is possible (including all phonemes in Indo-European languages, for example). We shall now present in greater detail the actual design of this device.

6.2. *Technical Details*

In this section, we shall give the design details of the digital synthesizer. Details will not, however, be taken to the logic level. Rather, the purpose is to illustrate and discuss the design approach to a level to enable the reader to evaluate the appropriateness of this design as compared to the alternatives. We shall begin by presenting an overview of the general architecture. This is followed by a discussion of the method of

[21] One "cent" is an interval equal to 1/100 of a tempered semi-tone. One cent is slightly below the limit of human pitch discrimination and therefore represents the preferred pitch resolution of the oscillators. See, for example Backus (1969).

[22] An important point to note is that the use of the various modes is not mutually exclusive. That is, it is perfectly possible to be synthesizing an 8 partial tone using additive synthesis, while we have VOSIM, FM, and fixed waveform modes being utilized at the same time. The flexibility of the arrangement is obvious, as is its benefit.

frequency control. Finally, a presentation of the various acoustic models embodied in the design is made.

6.2.1. *General Architecture*

The general layout of the device is shown in Figure 6.1. Here it is seen that the synthesizer itself is made up of four main modules: the controller, memory, oscillator, and digital-to-analogue converter modules, respectively. Communication among the modules is via a single high-speed data bus which is under the supervision by the controller module. Communication with the host computer is achieved *via* a parallel bus connecting the controller with an address decoder residing on the host computer's input/output bus.

Before proceeding to present details of the synthesizer's functional organization, a few points should be made concerning the method of interfacing. By isolating the synthesizer from the host, we are able to power down, test, repair, and/or modify the synthesizer *without* having to also power down the host. Furthermore, the host is protected from damage due to faults or damage in the synthesizer. These were important considerations given that the synthesizer was hosted by an expensive time-sharing system. Other users could not be allowed to suffer due to work being undertaken on the synthesizer. All of this was even more important since we take the same iterative approach to implementing hardware as we do for software. Namely, the device was up and running in the simple fixed waveform mode well before -- one by one -- the other modes were added.

6.2.2. *Frequency Control*

The basis of the digital oscillator is the sampling of a stored function. In this case, the function stored is one cycle of a selected waveform. The waveform is stored as 2K 12-bit samples in a random access memory (RAM) internal to the synthesizer. Sound is generated by outputing (scaled) samples from this table through a a digital-to-analogue converter (DAC) which is connected to some transducer such as a loudspeaker.

Since the waveform buffer (WFB) contains only one cycle of the waveform, frequency (cycles per second) is controlled by the number of times we cycle through the samples of the buffer each second. This can be controlled two ways. One is to sequentially output each sample of the buffer every cycle. Since the number of samples output per cycle is constant -- as is the WFB size -- the *rate* at which samples are output must vary for each frequency. The second alternative is to keep the sampling rate constant and vary the number of samples output each cycle; for example, if at frequency f every sample is output, then to output the frequency 2f we would output every second sample in the buffer. (Outputing half as many samples -- equally spaced throughout the buffer -- at the same rate doubles the frequency.) In the first case, frequency is specified in terms of the sampling rate; in the second, in terms of the offset between subsequent samples in the buffer.

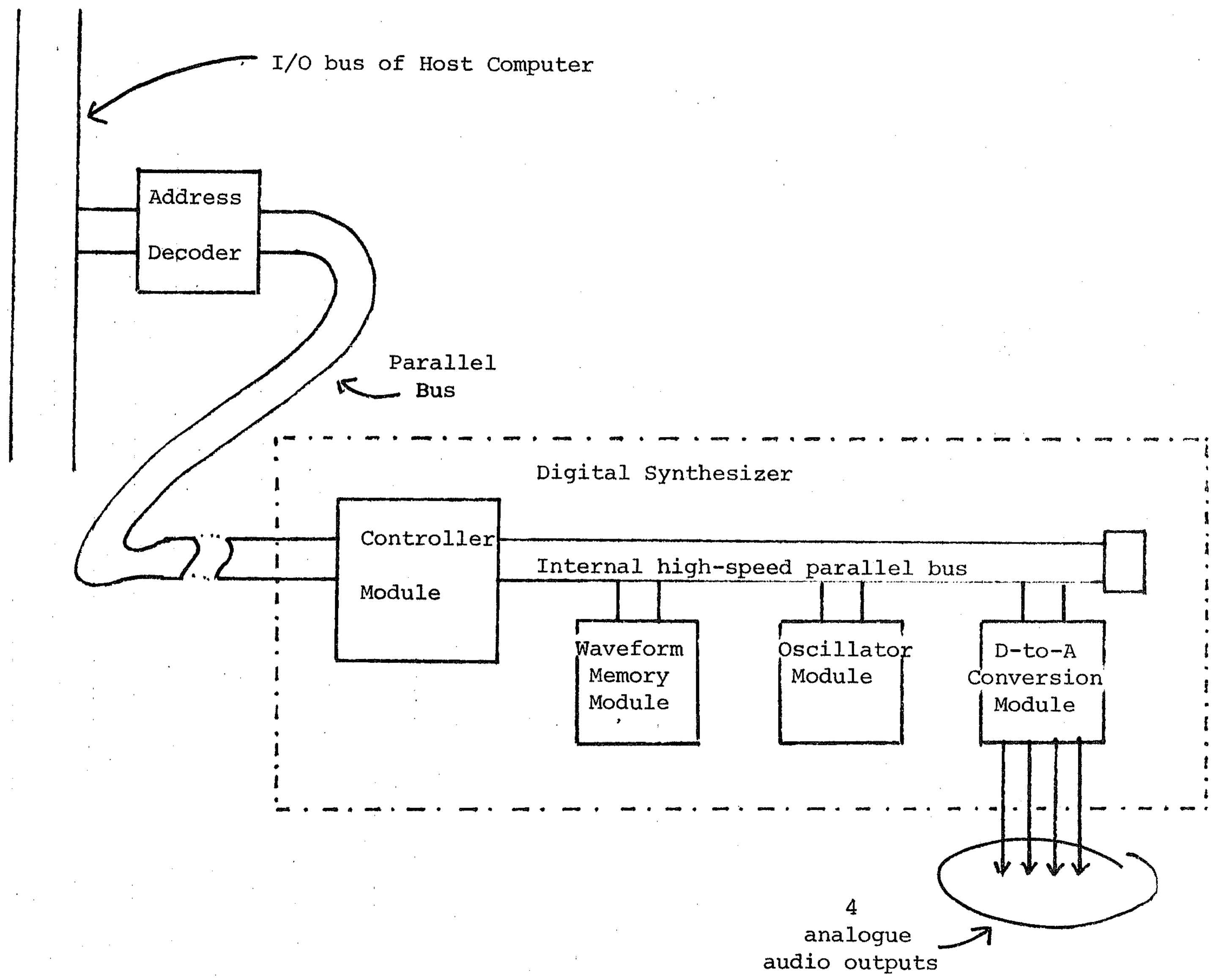


Figure 6.1: Block diagram of architecture of the digital synthesizer

Both approaches have been used with success. The variable sampling rate oscillator is used, for example, in the Dartmouth Synthesizer (Alonso et al., 1976) and the VOSIM oscillator (Tempelaars, in press; Kaegi and Tempelaars, 1978). However, two main problems must be overcome in taking this approach. First, when a bank of such oscillators is used -- each at a different frequency -- their samples are output asynchronously, making it difficult to mix or process them digitally. Hence, for example, each oscillator in the Dartmouth synthesizer has its own DAC. The second main problem is that at low frequencies the sampling rate may enter into the audio band (below about 16 kHz). This requires special consideration in terms of the filters following the DACs. Consequently, the cut-off frequency of the low-pass filters may be required to vary with the sampling rate.

In our system, we have chosen to take the more straightforward fixed sampling rate approach. The technique is well understood (it is the basis the MUSIC V software oscillators), the sampling of different oscillators is synchronous (making the time-division multiplexing of several oscillators rather straightforward), and the final stage filters have a fixed cut-off frequency. As has been stated above, frequency in this type of oscillator is controlled by specifying the offset in the WFB between subsequent samples. This offset -- or increment -- we call F_INC ("frequency increment"). Given the address of any sample n (A_n), then:

$$A_{n+1} = (A_n + \text{F\_INC}) \text{ modulo WFB size}$$

A generalized view of this mechanism is shown in the simple ramp generator shown in Figure 6.2. Here it is seen that the sum of the addition (and hence the address of the current sample) is accumulated in the register ACC -- to be used in calculating the address of the next sample. The modulo arithmetic is accomplished by simply ignoring the carry bits. Converting frequency from Hz to F_INCs is straightforward, given the sampling rate and WFB size. This is effected using the following formula:

$$\text{F\_INC} = \text{Hz} * \text{WFBS}/\text{SR}$$

where Hz is the frequency in Herz, WFBS is the waveform buffer size, and SR is the sampling rate.

6.2.3. *Fixed Waveform*

We obtain the fixed waveform mode of operation through a slight extension of Figure 6.2. A simplified presentation of this mode is made in Figure 6.3. Here, the principal components added concern the WFB table lookup and the DAC mechanism. In addition to F_INC, four new addressable registers appear. These are: WF_SEL, OP_SEL, ENV, and AMP.

As was stated in the Introduction to this chapter, there are eight buffers in which waveforms can be stored. Therefore, besides calculating the address within any particular WFB (the process illustrated in Figure 6.2), one must also specify from which of the eight WFBs the samples are to be taken. This is the purpose of the register WF_SEL ("waveform select"), which is simply a 3-bit value catenated onto the address calculated by the ramp-generator. One benefit of being able to easily change waveforms is in the potential for minimizing distortion due to aliasing, or fold-over. This can be

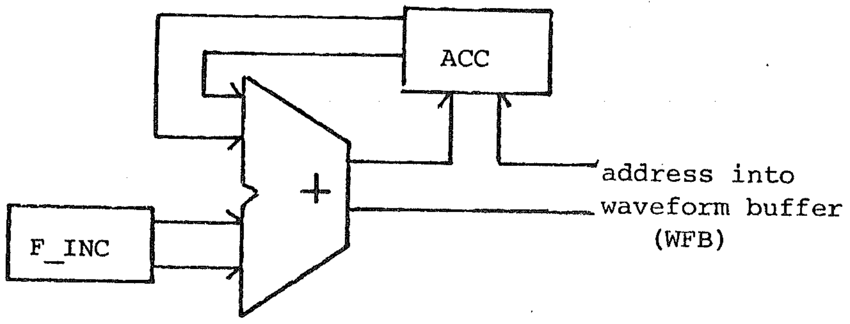


Figure 6.2: Simple ramp generator

accomplished simply by substituting simple sine tones for complex signals whose fundamental frequency is above a certain threshold. Since the harmonics of such signals would fall outside of the bandwith of human pitch perception, the substitution would be not be perceived and the generation of partials above the Nyquist frequency avoided.

There are four audio output channels in the synthesizer. The output of each of the sixteen oscillators may be routed to any one (and only one) of these four channels. The purpose of the register OP_SEL is simply to specify to which of these output channels the oscillator's output is to be routed.

Once a sample is obtained from the WFB, it is generally scaled in amplitude so as to be able to produce sounds of different loudness. One technique of doing so is to digitally multiply the sample by a scaling factor and then output the product through a normal DAC. This is the technique used by Alles and di Giugno (1978), for example. At the time of design, however, it appeared more economical and better from a control point of view to take an approach similar to that of the Dartmouth synthesizer. Here we carry out the scaling through the use of multiplying DACs, which were less expensive and complex than digital scaling. The waveform sample is placed in a 12-bit multiplying DAC and the output is scaled according to a reference voltage input. Thus, even at low amplitudes there are 12-bits of resolution of the waveform, resulting in better dynamic range than would be possible with an ordinary "fixed-point" 12-bit converter.

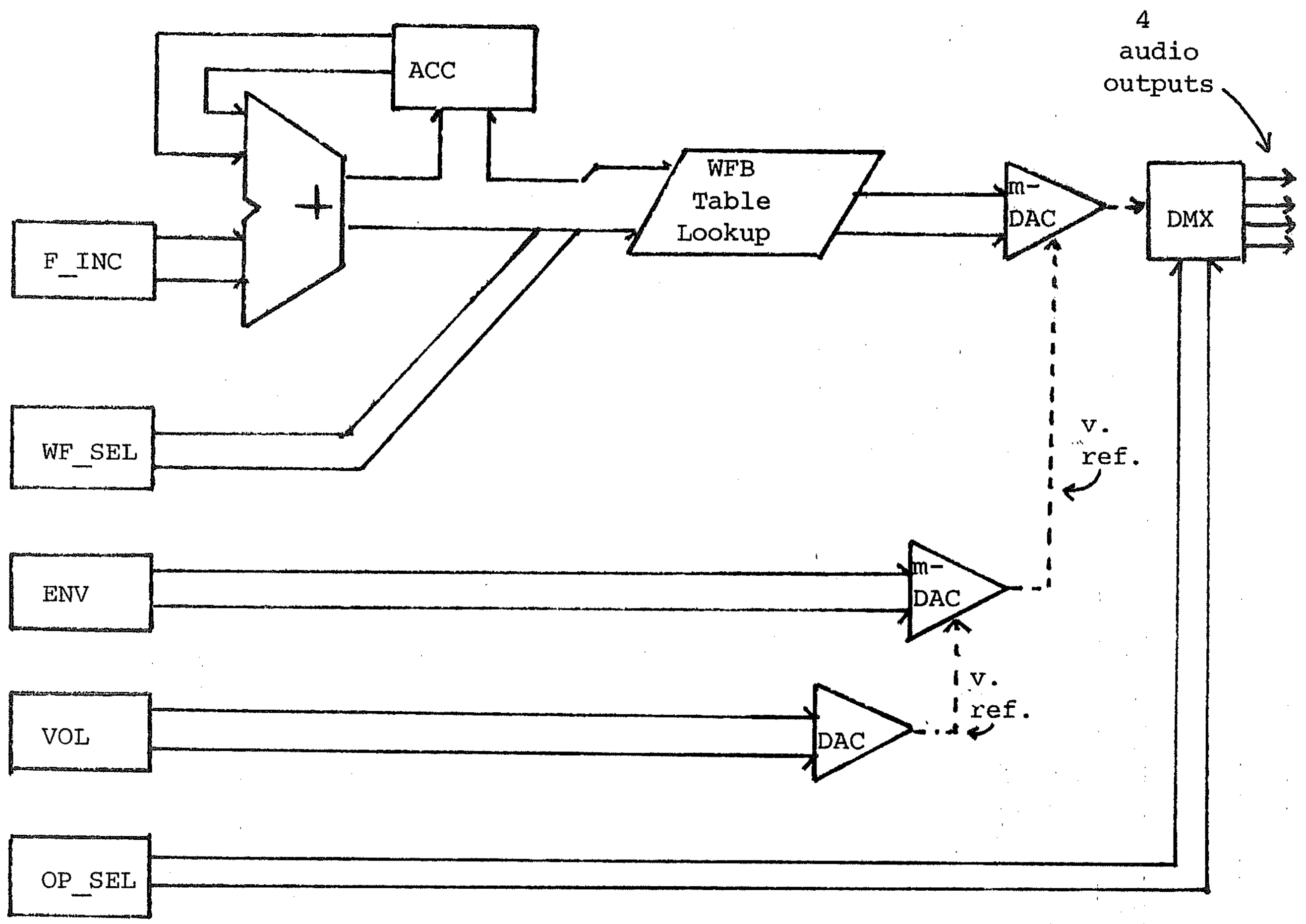


Figure 6.3: General view of simple fixed-waveform digital oscillator

One interesting idea of Alonso's which is incorporated into our design deals with the derivation of this reference, or "scaling" voltage. Rather than coming from a single amplitude value, separate notions of "envelope" and "volume" are carried over into hardware. That is, the envelope of a sound is scaled in hardware rather than software, thereby saving valuable CPU time. Amplitude scaling is, therefore, accomplished by three DACs in series. The first is the volume DAC. The second and third -- both multiplying DACs -- are the envelope and waveform DACs, respectively. The output of the volume DAC (as determined by the contents of register VOL) is used to scale the output of the envelope DAC (whose unscaled output is determined by the contents of register ENV). The scaled output of the envelope DAC is then used as the scaling voltage for the waveform DAC. One point worth noting in the current implementation concerns the volume DAC. Since loudness varies logarithmically with amplitude, a logarithmic -- rather than linear output -- DAC is used. Thus we have an example where psychoacoustic research has affected hardware design.

6.2.4. *Additive Synthesis: Bank Mode*

In the preceding section we saw how we can generate a sound having a particular fixed waveform and a varying amplitude contour. Given our knowledge of additive synthesis (discussed in Chapter Four), we see how a group of fixed waveform oscillators can be used to generate complex sounds having time-varying spectra. In this case, we have one oscillator corresponding to each partial to be synthesized. Since the technique simply involves the use of a group of fixed waveform oscillators, we refer to it as *bank mode.* For the same reason, we see that the technique requires no special-purpose hardware beyond that already described.

6.2.5. *VOSIM Mode*

The basis of VOSIM is the ability to output one cycle of a particular function (such as a sine pulse) followed by a controlled delay before the next cycle is output. Let us assume that the function is stored in one of the WFBs. Our method of implementation, then, incorporates a mechanism which outputs one cycle of the function stored in the WFB (the period controlled by F_INC), and then steps into a time-out mode for a specified delay period. A simplified illustration of our implementation of this mechanism is shown in Figure 6.4. Here it is seen that the period of delay is controlled by the contents of the register DEL. The oscillator functions in two modes: cycle and timeout. Cycle mode ends and timeout is triggered when there is a carry-bit out of register ACC (i.e., at the peak of the ramp, or when the WFB addressing "wraps around"). At this time -- timeout -- the contents of DEL are loaded into the count/compare (CNT/COMP) register which is decremented every 50/1000th sec. When the contents of this register equals zero (0), cycle mode is re-triggered and the CNT/COMP register disabled. We see, then, that when DEL equals zero we are continually in cycle mode and therefore effectively in fixed waveform mode of operation.

The VOSIM mechanism as described thus far is an over-simplification to facilitate the presentation of material. What the description omits is the method for controlling random deviation, or noise in the sound. The mechanism employed is illustrated in Figure 6.5. Again, the register CNT/COMP is loaded at the start of each timeout cycle. Similarly, CNT/COMP containing the value zero still triggers cycle mode, while an overflow

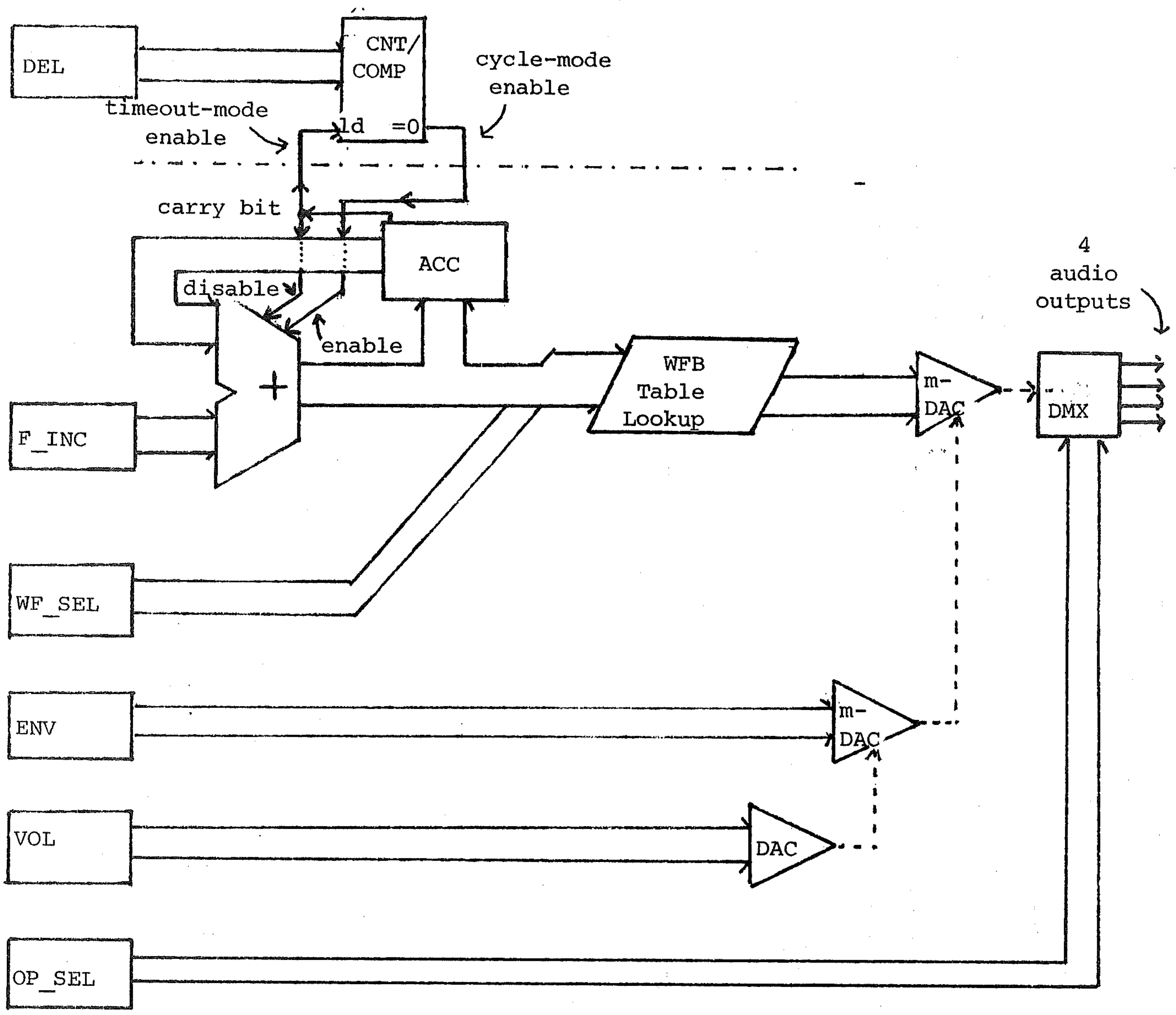


Figure 6.4: Simplified version of VOSIM oscillator

from ACC still triggers timeout. The difference, however, is in the value which is loaded into CNT/COMP. Instead of simply loading the value contained in register DEL, as diagrammed in Figure 6.3, the value loaded is the value contained in DEL *plus* a random value. The range of this random value is plus or minus some specified percentage of the contents of DEL. This percentage value is determined by the contents of the register DEV (% deviation). The actual origin of the random value is the random number generator labelled RNG.

It is clear from Figure 6.5 how the actual delay -- the value loaded into the register CNT/COMP -- is arrived at. A few points are worth noting, however. First, when the contents of DEV equals zero, we have effectively the situation diagrammed in Figure 6.4. Second, when the contents of DEL equal zero, we still effectively have fixed-waveform mode. Finally, the effective (average) fundamental frequency in VOSIM mode is determined by a combination of the contents of *both* the F_INC and DEL registers.

6.2.6. *Frequency Modulation*

The synthesizer has sixteen digital oscillators. Frequency modulation (FM) is implemented such as to allow any oscillator "n" to frequency modulate oscillator n+1 (modulo 16). While this format does not allow the use of multiple modulators of a single carrier wave (such as described in Schottstoedt, 1978), this deficiency is largely made up for by our ability to use modes other than FM. At the time the device was designed, the tradeoff was weighted towards economy and accessibility rather than generality.

In implementing FM certain extensions had to be made upon the basic oscillator as described thus far. These fall into two categories: those which enable the oscillator to be modulated, and those which enable it to modulate. The method of implementation is shown in Figure 6.6. Here a pair of oscillators are shown. For simplicity's sake the VOSIM components have been omitted (as in fixed waveform mode, register DEL would be set to zero). Similarly, since the DACs of the modulating oscillator are not used (i.e., are set to zero), they are not shown. Finally the diagram is made such that the first oscillator shows only the modulating mechanism, while the second shows only the additions to allow it to be modulated. It should be remembered, however, that both oscillators are in fact identical. This is seen in Figure 6.8, which shows a single oscillator which includes the mechanisms for all oscillator modes.

Returning to Figure 6.6, we see several points of interest. First, the maximum deviation of the frequency of the carrier oscillator is determined by the product obtained by multiplying the contents of the registers F_INC and MOD_INDEX of the modulating oscillator. Second, the actual instantaneous amount of deviation (MODULATION) is derived by multiplying the maximum deviation by the current sample taken from the WFB (again, of the modulating oscillator). Finally, the actual modulation is effected by adding the MODULATION to the contents of the register F_INC of the carrier oscillator. The sum of this addition is then used as input into the ramp generator.

There are a few additional points to note in the above. First, remember that both modulator and carrier may address any one of the eight WFBs through the use of their WF_SEL registers. Thus, we are not restricted to FM with sine waves only. Second -- and not so obvious -- every oscillator is *always* modulating its neighbour. However, in fixed-waveform mode, for example, these MOD_INDEX registers are set to zero --

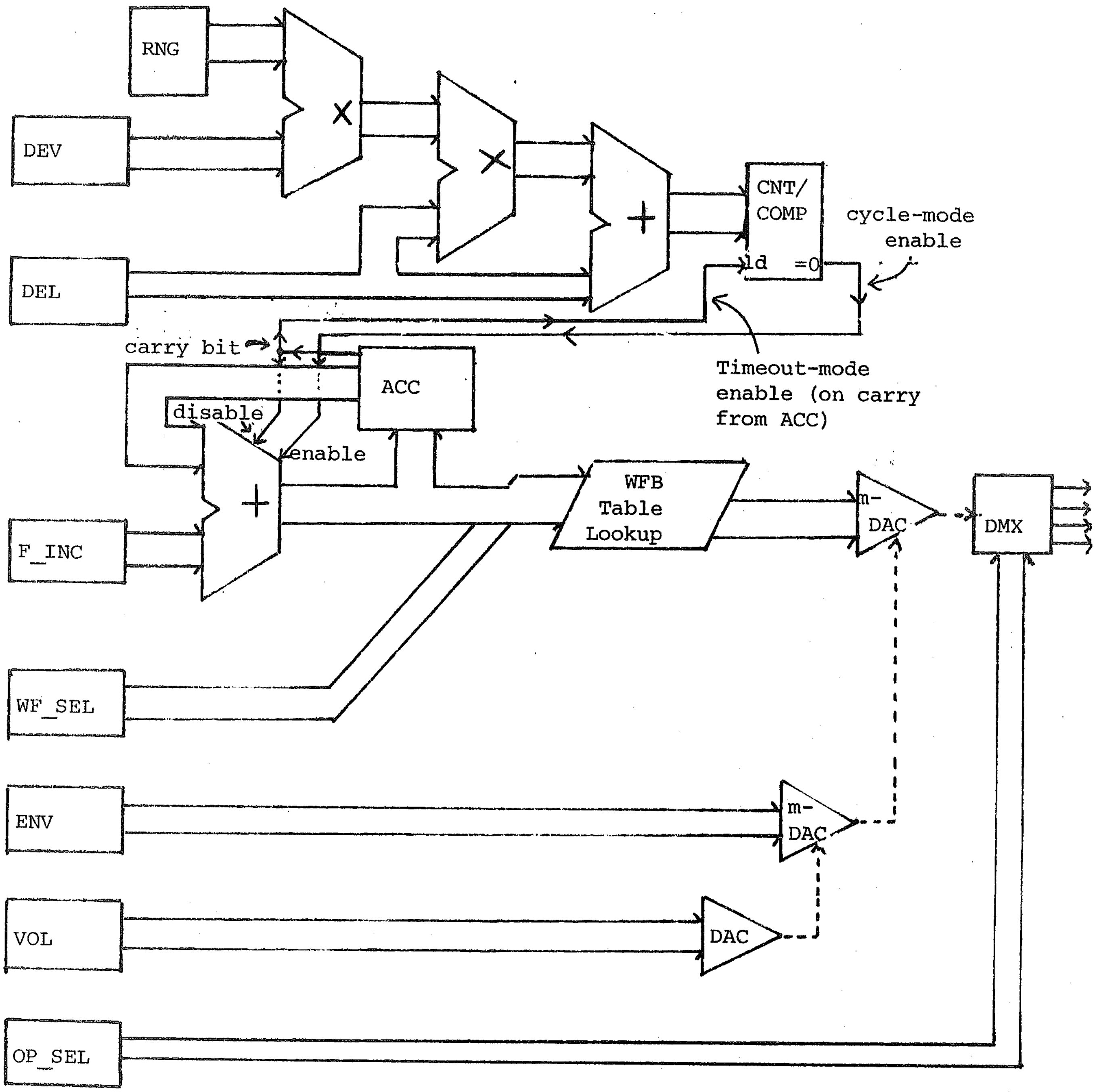


Figure 6.5: The VOSIM Mechanism

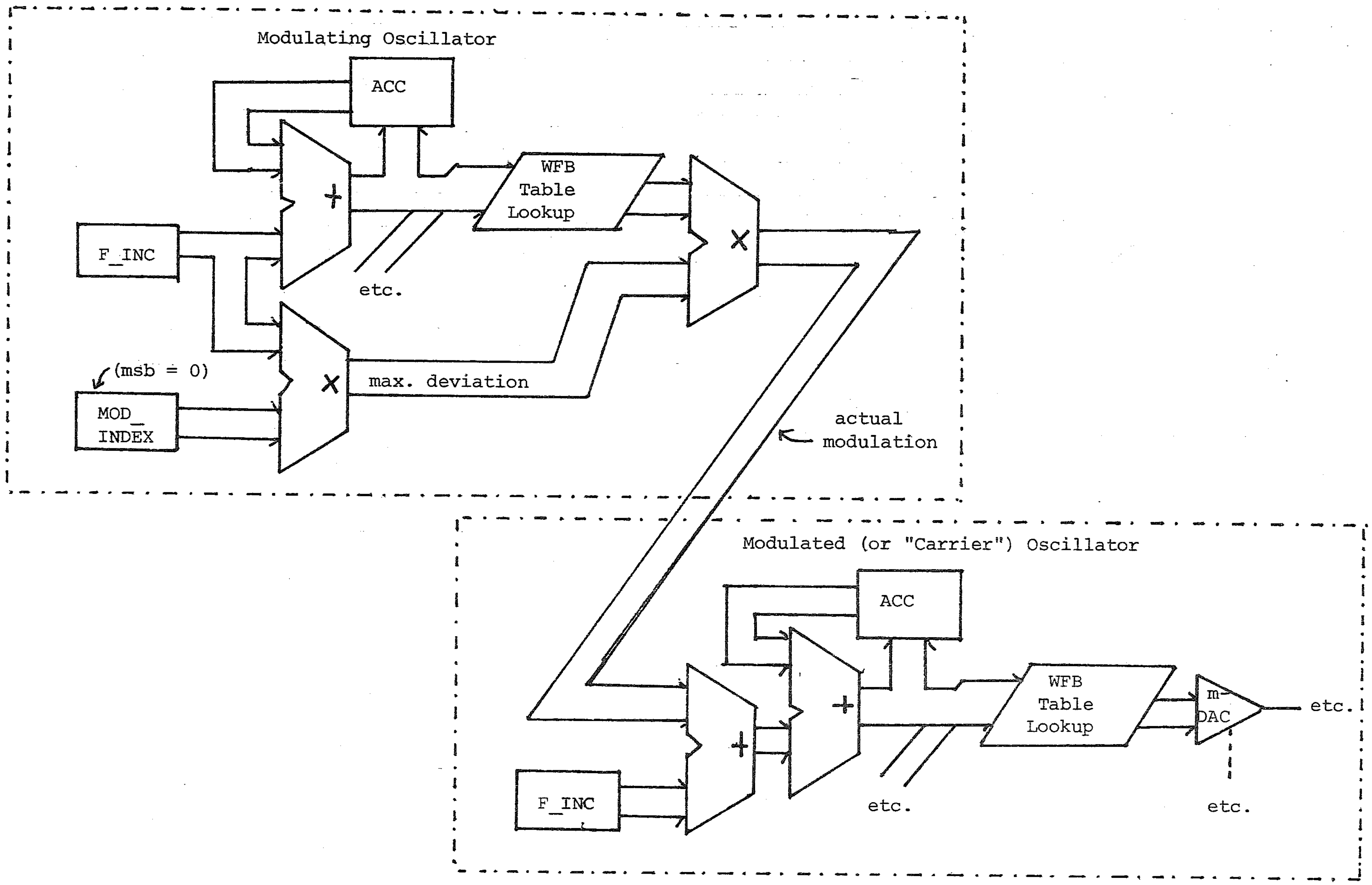


Figure 6.6: Simplified view of Frequency Modulation mechanism

effectively switching off the modulation. Finally, note that the richness of sound obtainable through FM is gained at the expense of two oscillators per sound. In complex structures this makes the low number of oscillators (sixteen) felt rather strongly. (This is equally true in waveshaping mode, and even more true in additive synthesis.) The user must, therefore, often resign himself to resources comparable to a quartet or octet rather than that of a symphony. Given the other benefits of the system, however, these limitations seem not so serious. There has, after all, been a great deal of "acceptable" chamber music written over the years.

6.2.7. *Waveshaping*

Inspired by LeBrun (1977), we decided to determine if we could incorporate waveshaping into the hardware structure. This turned out to be rather easier than expected and the result is a slight variation on the FM mode already described. A functional representation of the waveshaping implementation is shown in Figure 6.7. Like Figure 6.6, we see only the critical components of two oscillators. However, instead of referring to the oscillators as "modulator" and "carrier" as in FM, we will refer to them as "excitation" and "distortion", respectively.

Starting with the excitation oscillator, note that the configuration is just as in FM except that the multiplication of the contents of the F_INC and MOD_INDEX registers is missing. The contents of MOD_INDEX, therefore, function as a simple scaling factor for the samples taken from the WFB. Secondly -- concerning the distortion oscillator -- notice that the entire ramp generating mechanism for WFB address calculation (including ACC) is missing. In this mode, the contents of F_INC is set to a value such that (by itself) it addresses the sample midway into the WFB. The output of the excitation is then simply added onto this constant and the sum is used as the address into the distortion WFB. The sample thus addressed is then output to a DAC, thereby completing the basic waveshaping process.

It still remains, however, to present how the mechanism shown in Figure 6.7 is obtained using that shown in Figure 6.6. The key to doing so lies in having the MOD_INDEX register function in two different modes. The current mode is determined by the most significant bit (msb) of the MOD_INDEX register. When this bit is zero, we have regular FM as described in the preceding section. When, however, the msb is equal to one for a particular oscillator, the following happens:

1. In the multiplication of the contents of the F_INC and MOD_INDEX registers, the F_INC factor is replaced by the constant "one". This effectively nulls the effect of the multiply. F_INC still controls the frequency of the excitation oscillator.
2. The ACC register of the next oscillator (i.e., that of the distortion oscillator in the pair) is cleared *after every sample*. This effectively disables the ramp generator in the way diagrammed in Figure 6.7.

Thus, the requirements of waveshaping are seen to be straightforward, and the implementation not difficult.

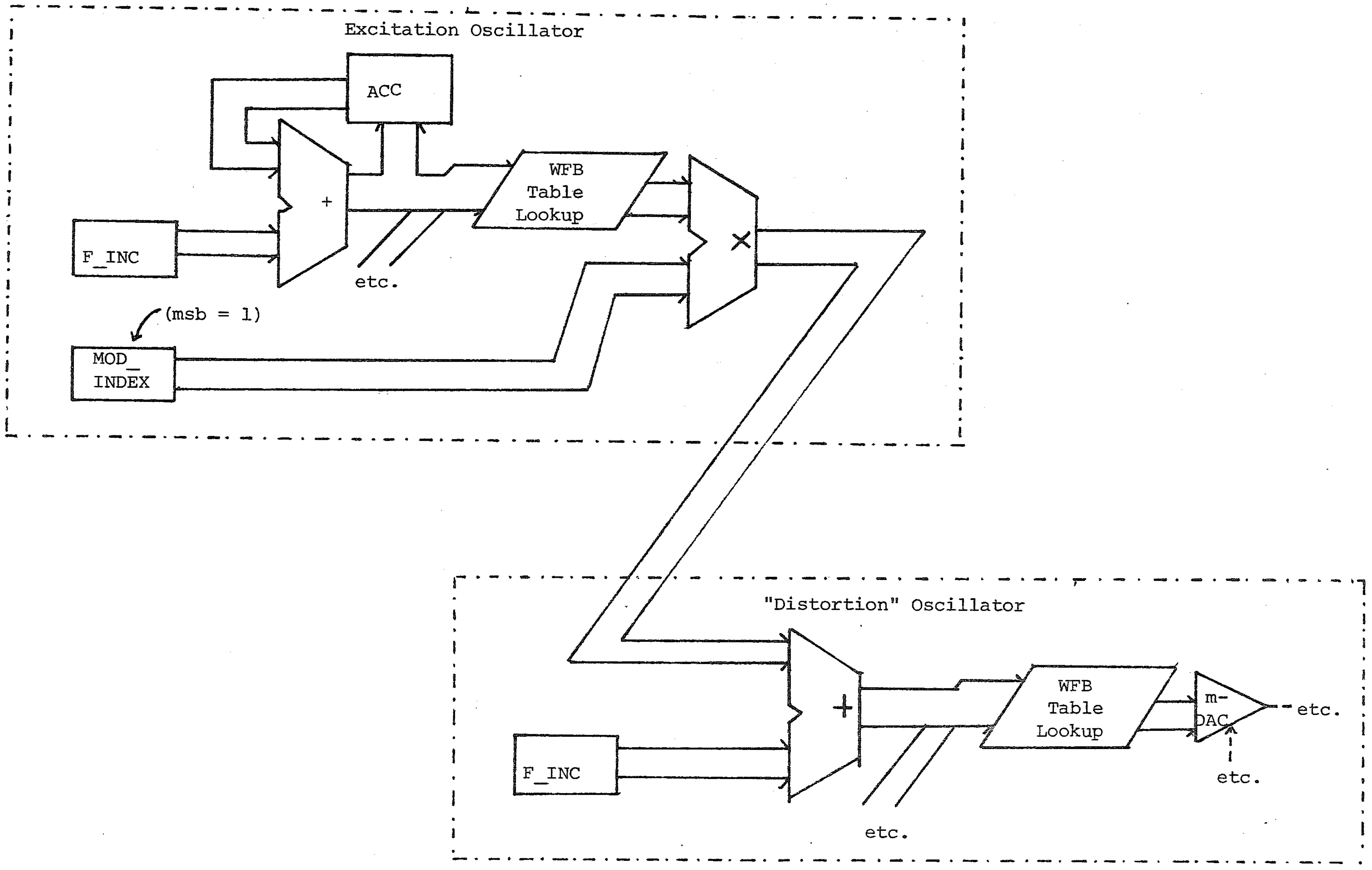


Figure 6.7: Functional view of waveshaping mechanism

6.3. *Summary*

An outline of the implementation of the various synthesis modes used in the digital synthesizer has been presented. Perhaps the most significant point about the implementation is that all of the oscillators are, in effect, functioning in all modes at all times. The effects are simply made invisible by "nulling" appropriate registers. The effect of this is that the synthesizer's resources can be easily distributed so as to permit sounds utilizing different acoustic models to be synthesized simultaneously with very little overhead on the CPU. In addition, while FM mode does not allow the use of multiple modulators for a single carrier, the hardware does allow for obtaining similar effects by using the output of waveshaping mode as the modulator in FM, or waveshaping the output of FM. The device is designed so as to produce good audio quality, and is able to be easily interfaced to a large number of different computers.

The prime drawbacks of the device are the limited number of oscillators and the lack of generality -- when compared, for example, to MUSIC V. These are not felt to be too serious in our context, however, when one recalls that one of our prime interests is the relationships among sounds, rather than the intrinsic value of the sounds themselves. This is something which we feel can be effectively exploited with the available resources.

One other aspect of the design which is now somewhat questionable is the method of implementing the digital-to-analogue conversion process. While we would retain the control mechanism -- that is, the method of scaling the waveform using the ENV and VOL registers -- the actual scaling would be done digitally rather than using multiplying DACs. Furthermore, digital scaling would then enable us to use only four DACs in total. Again, the current design was made in the interest of economy, and has served well in terms of fidelity and reliability. If changes are made in the future, it is worth noting that the modular nature of the architecture enables us to change one module with only minimal trauma to the others.

Finally, the reader is directed to Appendix One which presents a summary of the synthesizer registers discussed. Also in this regard, the reader is redirected to Figure 6.8 in which these registers are collectively illustrated.

7. *Data Structures*

7.1. *Introduction*

This chapter presents a data-structure which meets the criteria presented in previous chapters. A version of the structures has been implemented and utilized with success at the University of Toronto. An overview of the structures is presented in Figure 7.1. Here we see that there are four main types of structures, each of which constitutes a particular type of file. These are:

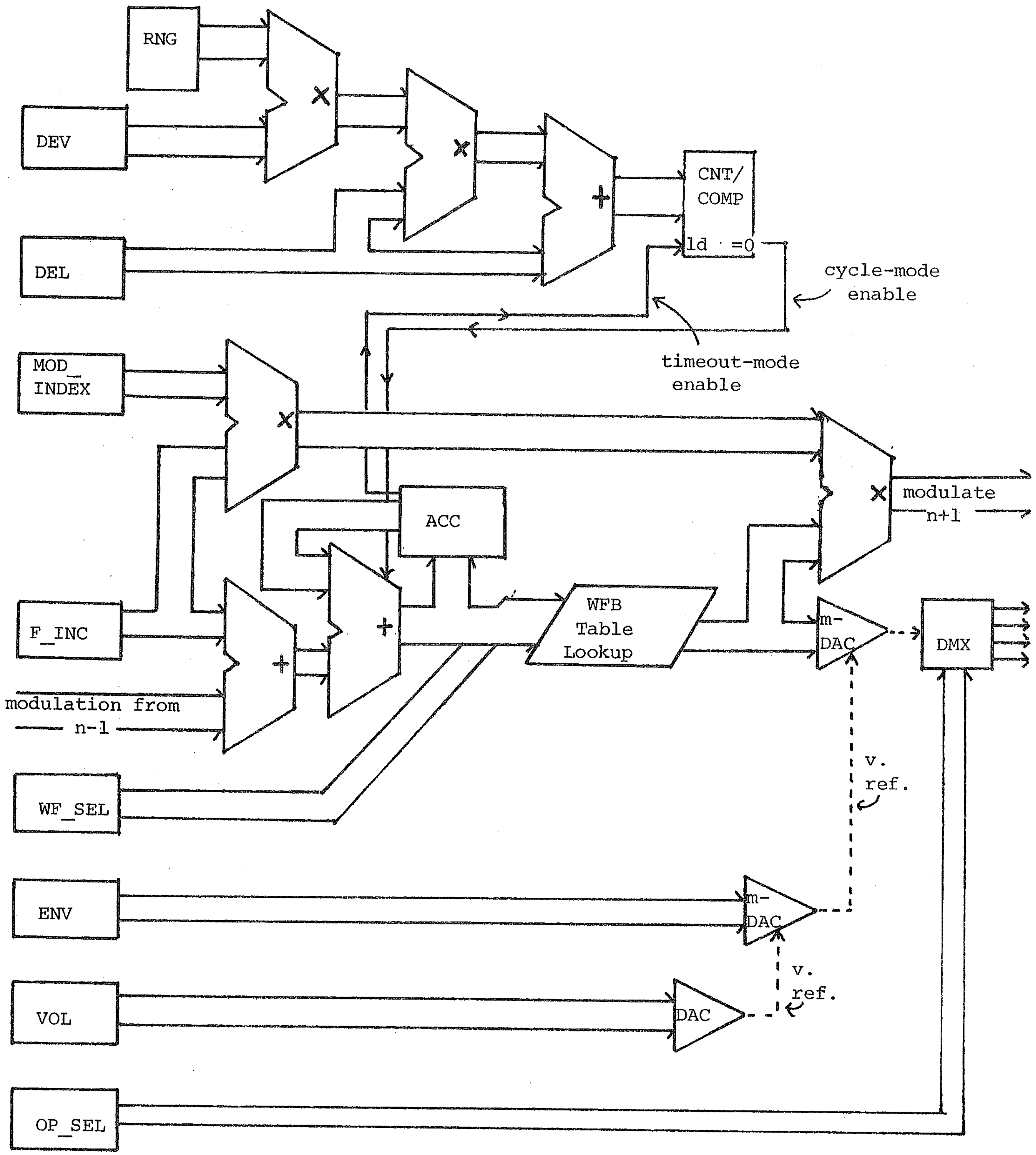


Figure 6.8: Composite view of digital oscillator

1. Scores
2. Objects
3. Functions
4. Waveforms

In addition, each of these file types is made up of various composite structures. The purpose of this chapter, therefore, is first: to present the internal representation of each of these four types of files, and second: to define the methods of communication, or links, among these files.

Since it is a structure common to various file types, and since it is the prime medium of inter-file communication, we shall begin by presenting the form of the *symbol table* data structure [23].

7.2. *Inter-file Communication*

7.2.1. *The Symbol Table*

The symbol table is the method of linking auxiliary files to both object files and score files. Therefore, both score and object files contain symbol tables.

A symbol table is an array of *symbol* structures, where the size of the array, or table, corresponds to the number of symbols, or entries, in that table [24]. The symbol structures for a particular table are stored in contiguous memory. Each entry in a symbol table has the following structure format: [25]

---

[23] Note that in the discussion which follows, any name or value specified entirely in upper-case characters (such as OBJECT, UNDEFINED, etc.) is a defined constant for the music system.

[24] In the implementation described, the symbol table size is limited to 256 entries, which has not proven to cause any constraints on the user. We can, therefore, take advantage of a space saving in that indices into the table can be represented by a single byte of information.

[25] Note: all examples are given in the programming language "C" (Kernighan and Ritchie, 1978). In the examples, a structure is an aggregate of data. The name following the *label* "struct" is the name of the aggregate. The names within the curly brackets define a template for the data in the aggregate. The first value in each row indicates data-type (char: 1 byte; int: 2 bytes; float: 4 bytes), while the second value is the variable name. Values preceeded by a "*" are pointers to data of the indicated type (such as a structure). Pointers occupy one word of memory. Memory for such structures may be dynamically allocated or freed, and several structures of the same type may be allocated space in contiguous memory to form a table, or vector, of structures (as with a symbol table). Finally, variables terminating with a value in square brackets ("[" and "]") are arrays whos dimensions are contained within the brackets.

```
struct symbol
        {
        char name[FNAME_SIZE];  - File name
        int stype;              - Type of symbol
        int svalue;             - Value of symbol
                                  (may be pointer)
        };
```

The *name* field simply contains the name of the file associated with the symbol in question. The *stype* field then indicates the type of file this entry in the symbol table is. Valid symbol types include:

1. OBJECT
2. SCORE
3. FUNCTION
4. WAVEFORM

Each of these symbol types corresponds to one of the file types mentioned above, and will, therefore, be dealt with in more detail below. Finally, if the file in question is in primary memory, the third field of the symbol entry -- the *svalue* -- contains a pointer to the file's core image.

We see, therefore, that access to subordinate files is accomplished through a symbol table; via the *name* fields for files not in primary memory (i.e., those requiring system i/o), and via the *svalue* field for others (thereby avoiding the time-consuming i/o) [26].

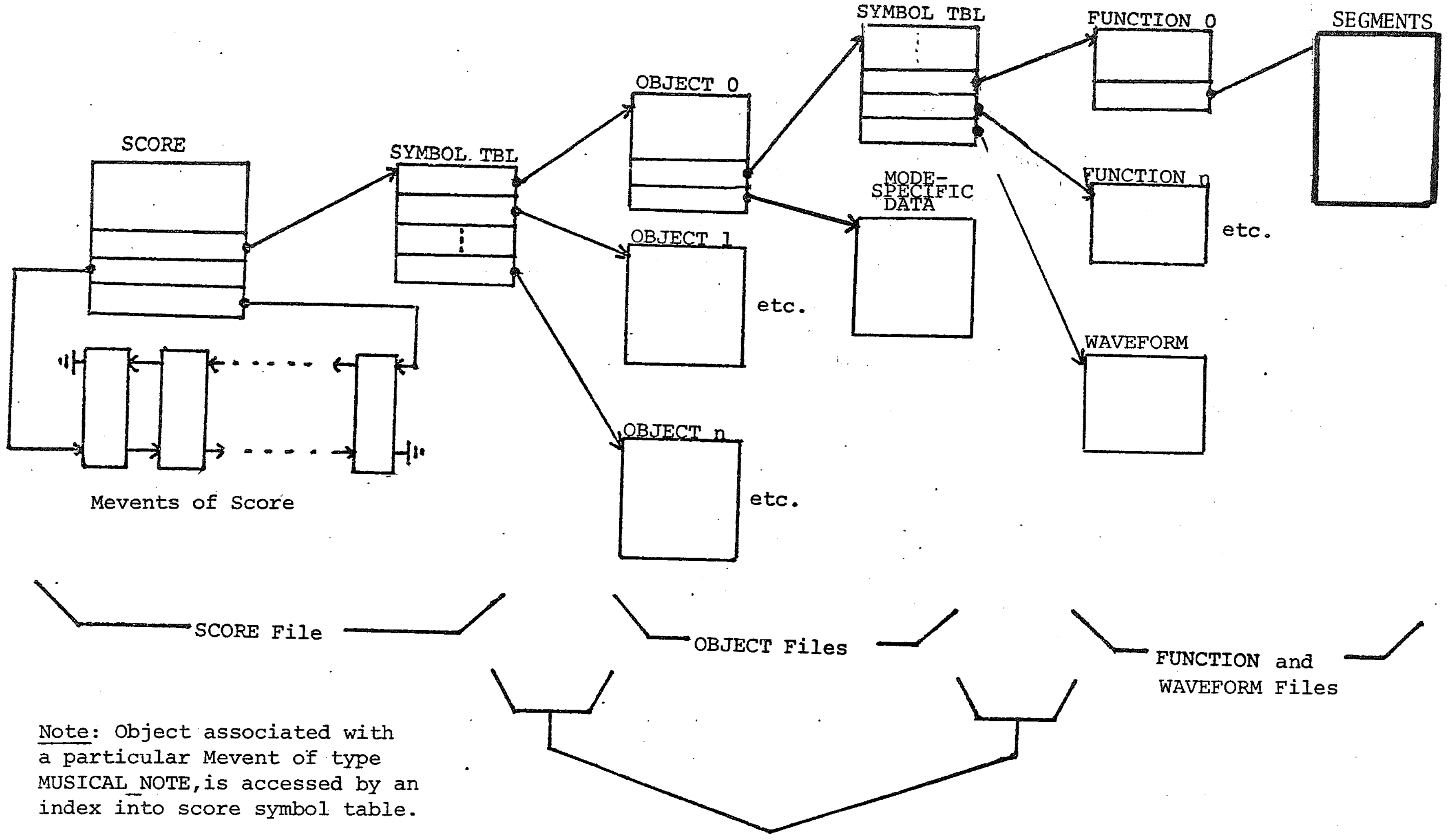
NOTE: A particular symbol entry is accessed by providing an index into the table. An important convention to note in this regard is that the first entry in the table is accessed by an index of one (1) *not* zero (0). An index of zero into the symbol table has the special meaning that that symbol to be referenced is not yet defined; a default symbol of the appropriate type (context dependent) is substituted. Thus, the mechanism for handling *default* situations is provided, the user never having to provide details beyond his current concern.

7.3. *File Types*

7.3.1. *SCORE Files*

For our purposes, a *score* is essentially a list, or sequence of musical events, called *Mevents*. Thus, it can be seen as a performance script for a composition. A great deal of effort has been spent in providing the flexibility in the data structures of a score to enable the structuring of a score in a hierarchic manner.

[26] Note the special case for WAVEFORM files, where we interpret primary memory as the eight 2k word waveform buffers in the synthesizer. Thus, a non-NULL *svalue* for a WAVEFORM entry indicates which of the buffers (1-8) contains the waveform..



Note: Object associated with a particular Mevent of type MUSICAL_NOTE, is accessed by an index into score symbol table.

File Links:

from disk: via file name in symbol table.

in memory: via pointers in svalue in symbol table.

Figure 7.1: Overview of Data Structures

A SCORE file consists of three main data structures: a *score* structure, a linked list of *Mevent* structures, and a *symbol table*. We will now proceed to present the details of each of these structures.

7.3.1.1. *'score' Structures*

The *score* structure functions as the header to the SCORE file. Besides storing the file name and a "magic" number identifying the file as type SCORE, it contains pointers to the head and tail of its associated list of Mevents. As well, it contains a field indicating the total duration of the score, and links to functions affecting the score's performance.

When the score is saved on disk, all of the score structure is written first, followed by the symbol table, and then the Mevents in sequential order. Thus, the link fields are not needed on disk.

The detailed composition of the score structure is as follows:

```
struct score
        {
        int magic;                  - Magic number
        char fname[FNAME_SIZE]; - File name
        int nsyms;                  - Number entries in table
        struct symbol *sym_table; - Pointer to symbol table
        float tot_dur;              - Total duration of score
        int nMevents;               - Total number Mevents
        struct Mevent *head;        - Pointer to start of Mevent list
        struct Mevent *tail;        - Pointer to end of Mevent list
        char dyn_ind;               - dynamic (volume) variation
        char tempo_ind;             - tempo variation
        char deltat_ind;            - entry-delay (articulation)
                                      variation.
        };
```

The last three fields are indices into the SCORE symbol table accessing functions controlling global features of the score: dynamics, tempo, and articulation, respectively.

7.3.1.2. *The Mevent*

As stated above, an essential component of a score is a sorted list of musical events which we call Mevents. While there are various recognized types of Mevents allowable in this list, they all conform to the following structure template: [27]

[27] It is important to note that the Mevent structure is simply a template. It functions as a generalization for the different types of events which may occur in a score, and is included for purposes of convenience.

```
struct Mevent
        {
        struct Mevent *flink;       - Pointer to next Mevent
                                      Last Mevent flink = NULL.
        struct Mevent *blink;       - Pointer to previous Mevent
                                      First Mevent blink = NULL.
        int tag;                    - User definable tag
        float delta_t;              - Entry delay before start of
                                      next Mevent.
        char type;                  - Mevent type
        char MEfld1;                - Field is type dependent
        int MEfld2;                         "
        int MEfld3;                         "
        int MEfld4;                         "
        int MEfld5;                         "
        int MEfld6;                         "
        };
```

As can be seen from the above, Mevents are represented as a doubly linked list (i.e., pointers to both the previous and next Mevents). This is to facilitate insertions, searching and other transformations on the list (playing the score in retrograde, for example) [28]. In the list, the order of the linking specifies the order in which the Mevents are to be played. The *delta_t* field in each structure specifies the time between the start of the current Mevent and the start time of the next. If this value is zero, the Mevents are played simultaneously (such as with a chord). On the other hand, if the *delta_t* value exceeds the duration of the Mevent, the result is a rest whose duration is equal to their difference.

The currently available types of Mevents (as specified in the *type* field) are: MUSICAL_NOTE, and SCORE. An Mevent of the MUSICAL_NOTE type is simply a single sound event. A SCORE-type Mevent is just that, a (sub)-score which commences at a particular point in a composition. It is this implementation of the notion of *sub-score* which enables us to create scores which are hierarchically structured. (See Figure 8.3, for example.) More formally, we can view a score as a tree structure in which a SCORE Mevent (called Mscore) constitutes a non-terminal node, and each MUSICAL_NOTE Mevent (called Mnote) contitutes a terminal, or "leaf", in the tree.

In the above structure, one feature is of particular note. This is the choice of using "delta" rather than "absolute" values for time (i.e., the entry delay value *delta_t*). This choice is based on the ease with which several instances of the same sub-score can be

[28] Many music systems, for example Tucker et al. (1977), avoid linked lists in the score. Instead, "notes" are stored in contiguous memory; the pointers then being implicit. While such a representation provides a more compact representation and a more efficient perform program, editing -- which is the prime function of our system -- is considerably less efficient. Furthermore, if in using the linked list approach the performance is too complex for the system to keep up with in real-time, we have found that enabling a score to be "compiled" into a more efficient representation is adequate for handling these special cases.

merged into another "master" score.

Since the interpretation of the Mevent structure fields MEfld1-6 are dependent on the Mevent type, we shall now consider the individual types in more detail.

7.3.1.2.1. *The MUSICAL_NOTE: Mnote*

An Mevent of the MUSICAL_NOTE type is a single sound event which has certain characteristics (or parameters) as defined by the following *Mnote* structure. (Note that this structure exactly follows the template of the Mevent structure.)

```
struct Mnote
        {
        struct Mevent *flink;      - Pointer to next Mevent
        struct Mevent *blink;      - Pointer to previous Mevent
        int tag;                   - User definable tag
        float delta_t;             - Entry delay before start of
                                     next Mevent.
        char type;                 - Mevent type: set to MUSICAL_NOTE
        char volume;               - Volume of note
        float frequency;           - Note frequency
        char object_ind;           - Index into symbol table to
                                     access object (timbre).
        char chan_no;              - Output channel of note
        float duration;            - Duration of note
        };
```

7.3.1.2.2. *The SCORE: Mscore*

An Mevent of the SCORE type is called an Mscore. The fields of the Mscore structure are given below. Again, note that the structure format follows that of the Mevent.

```
struct Mscore
        {
        struct Mevent *flink;        - Pointer to next Mevent
        struct Mevent *blink;        - Pointer to previous Mevent
        int tag;                     - User definable tag
        float delta_t;               - Entry delay before start of
                                       next Mevent.
        char type;                   - Mevent type: set to SCORE
        char vol_factor;             - Relative shift for sub-score.
                                       (vol. is log, so is addition)
        float pitch_trans;           - Relative shift for sub-score.
                                       (i.e., pitch transposition)
        char  score_ind;             - Index into symbol-table
                                       to access sub-score.
        char time_interp;            - Determines if time_factor affects
                                       entry_delay, duration, or both
        float time_factor;           - Temporal transformation
                                       (augmentation/diminution)
                                       of sub-score.
        };
```

There are a few very important points to note regarding the use of sub-scores. First, note that each appearance of a particular sub-score constitutes an *instance* rather than *master copy* of that sub-score. The difference is that there is only one master-copy of the sub-score (accessed through the symbol table) and any changes to the original are reflected in each instance during a composition. Therefore, if we view a score as a tree structure, then the *pitch_trans*, *vol_factor*, and *time_factor* fields of the Mscore structure will effect transformations on the sub-score (or sub-tree) below them. Musically, therefore, these fields allow for the occurence of the sub_score starting at any pitch (i.e., transposition), the dynamics to be scaled, and the augmentation and diminution of the time structure [29]. The result is that we can obtain several versions of a single "score", while maintaining only one copy of the original.

7.3.1.3. *SCORE Symbol Table*

The types of symbols which are legal in a score's symbol table are: FUNCTION, (sub) SCORE, and OBJECT. If the stype field of a symbol's entry is UNDEFINED, a default symbol is substituted. If the entry's *svalue* is non-zero (viz., not UNDEFINED), there is an image of the symbol in primary memory and the *svalue* is a pointer to it. Otherwise, the *svalue* must be UNDEFINED.

Finally, if the nsyms field of the associated score structure equals 0 (zero), there is no symbol table, the field **sym_table* should equal NULL, and default values of the appropriate type will be inserted during performance. The implications of this are (musically) important in that no ordering of operations is imposed on the composer. He may, for example, perform the pitch/time structure of a composition before any

[29] Note that space/time trade-offs dictate that the *time_factor* field affects either Mevent *durations* or *delta_ts*, or both (as determined by the *time_interp* field).

thought is given to orchestration. Furthermore, he may orchestrate the score with yet undefined objects (see below), and still audition the work with default objects substituted. Finally, in either case the default object(s) substituted may be user defined (i.e., the user may personalize the system by over-riding the system defined defaults).

7.3.2. *OBJECT Files*

One of the aims of the music system is to provide a facility whereby a composer can specify his own palette of timbres to be used in a composition. Each set of timbral characteristics defined by the composer, called an *object*, is then stored in a file named by the composer. Notes in a score may then be "orchestrated" by establishing an association with a particular object file. This is accomplished via the object_ind field of the Mnote structure, in combination with the score symbol table (as outlined previously).

We saw above that there may be several instances of the same (sub)-score in a composition. Similarly, there may be numerous Mnotes of various durations, pitches, and amplitudes, all deriving their timbral characteristics from the same object. Furthermore, any change of the object file will cause that change to be reflected in all instances of that object in a score. We see, therefore, that the object functions as a type of timbral "template". Finally, due to this template nature of the object, the only restriction on how many instances of that object which may occur simultaneously is the number of oscillators in the synthesizer. This is in contrast with the notion of "instrument" as developed in MUSIC IV (Mathews, 1969), for example.

While all objects serve the same musical purpose of timbral control, there exist different internal representations for object data. These differences primarily reflect the different modes -- or acoustic models -- whereby sound can be generated by the ZYSP digital synthesizer. We will see, therefore, that there are three main data structures in an object file. These are: the *object* structure and *symbol* (table) structure (both common to all objects, regardless of mode); and the *type_object* structure, which contains the data peculiar to the mode of that particular object.

7.3.2.1. '*object*' *Structures*

The object structure contains information common to all objects, regardless of mode. Such information includes the objects's name, mode, and a "magic" number to distinguish OBJECT files (from, for example, SCORE files) during various operations such as reading and writing. The structure also specifies the number of critical resources (i.e. synthesizer oscillators) required by that object. This information is represented as follows:

```
struct object
        {
        int magic;                  - Magic number
        char fname[FNAME_SIZE]; - File name
        int nsyms;                  - Number symbols in table
        struct symbol *sym_table; - Pointer to symbol table
                                      nsyms long.
        int mode;                   - Designates type of object
        int noscils;                - Number of oscillators needed.
        union type_object {
                struct fixedwf_object fwfobj;
                struct fm_object fmobj;
                struct bank_object bankobj;
                struct ws_object wsobj;
                struct vosim_object vosimobj;
                } *data;            - Defines "*data" as a pointer to
                                      data peculiar to one of the possible
                                      object types (see below).
        char rigidfunc_ind;         - Index into symbol table
                                      to access basis of function
                                      time-scaling.
        };
```

One field of the object structure warrants special attention. This is *rigidfunc_ind.* As will be seen below, each mode of object specification includes the specification of functions which determine how parameters vary over time. The time base of such functions, however, must be able to be scaled over Mevents (eg., Mnotes) of various durations. This is in keeping with the notion of an object being a general template for timbre. One problem is, however, is that in compressing and expanding functions we do not always want the scaling to be linear. That is to say, if we consider the x (or time) axis of a stored function as a spring, we do not always want the spring to be of uniform stiffness. In imitating sounds which occur in nature, for example, we would want the attack and decay portion of the amplitude function to be more "stiff" than the steady-state. Similarly, in other objects we might want just the opposite. In view of this problem, each object has associated with it a user-definable "rigidity" function, which determines how the functions of that object are to be scaled -- in time -- in their various instances throughout a composition. The *rigidfunc_ind* field provides, therefore, an index into the symbol table which identifies the "rigidity" function for that object.

7.3.2.2. *Object Types*

As was stated above, there are different methods of representing objects which reflect the method of sound synthesis used. These modes are as follows:

1. Fixed Waveform (FIXEDWF)
2. Frequency Modulation (FM)
3. Pulse Modulation (VOSIM)
4. Additive Synthesis (BANK)
5. Waveshaping (WAVESHP)

The amount and type of data required is different for each of these modes. Therefore, there is a different type of structure used for each. The definition of the structure peculiar to each object type is given below. The appropriate structure for a particular object's type-specific data is accessed *via* the **data* field in the *object* structure, whose *mode* field indicates the structures type.

7.3.2.2.1. *FIXEDWF Objects*

The fixed waveform synthesis mode utilizes a single oscillator as a function generator. The only parameters at the object level in this mode are: the waveform used, the amplitude contour (or "envelope"), and the frequency contour (or deviation over time). The amplitude and frequency contours are stored functions (see FUNCTION files, below) and are accessed through the object symbol table. The format for FIXEDWF data is as follows:

```
struct fixedwf_object
        {
        char fwf_ind;           - Index into object symbol
                                  table to define waveform.
        char envel_ind;         - Index into object symbol
                                  table for amp. function.
        char freq_ind;          - Index into object symbol
                                  table for freq. function.
        };
```

7.3.2.2.2. *FM Objects*

The FM mode of object specification enables sound to be synthesized by having one oscillator ("m") modulate the frequency of another (the "c", or carrier oscillator). The resulting relevant parameters include: the ratio between the frequencies of the two oscillators (the "c:m" ratio), the maximum degree of modulation and how modulation varies in time, and the amplitude and frequency contours (as seen with FIXEDWF objects). The format of FM mode data is as follows:

```
struct fm_object
        {
        struct fixedwf_object car;  - Carrier waveform
                                      (as in FIXED_WAVEFORM)
        char mfwf_ind;              - Index into object symbol
                                      table defining mod. waveform.
        char mdev_ind;              - Index into object symbol
                                      table for mod. function.
        int maxindex;               - Max. Modulation Index
        int cval;                   - C term in C:M freq. ratio.
        int mval;                   - M term in C:M freq. ratio.
        };
```

7.3.2.2.3. *VOSIM Objects*

The VOSIM mode enables voice-type synthesis via a form of pulsewidth modulation. There are different degrees of complexity possible; generally, the more complex, the more oscillators or "VOSIM functions" must be used. Besides the pulse-shape (waveform) select and the amplitute and frequency functions, each VOSIM function also has the following parameters: the pulse-width, how the pulse-width varies in time, and the degree of randomization (to produce consonants, or noisy spectra). The format for the VOSIM data is as follows:

```
struct vosim_object
        {
        struct fixedwf_object vosfn;- As in FIXEDWF.
        char maxdev;                - Maximum deviation (i.e.,
                                      noise) factor.
        char dev_ind;               - Index into object symbol
                                      table for dev./time function.
        char pw_ind;                - Index into object symbol
                                      table for pulse-width (i.e.,
                                      formant) change function.
        int pwf;                    - Pulse-width expressed as
                                      frequency.
        };
```

Note: Complex VOSIM objects utilize more than one VOSIM function or oscillator. When this is the case a table of *vosim_object* structures is kept in a contiguous portion of memory. The number of entries in this table is given by the *noscils* field in the parent *object* structure.

7.3.2.2.4. *BANK Objects*

This mode enables the use of several generators together, such that each oscillator functions as one component, or partial, in a complex tone. The frequency and amplitude of each component may vary over time. The actual frequency of any component is its partial number times the fundamental frequency (where the fundamental frequency is considered partial number 1). The data for the various partials in a particular object are stored in a table of *bank_object* structures. The format of these table entries is given below. The number of entries -- which are stored in contiguous memory -- is given by the *noscils* field of the object structure.

```
struct bank_object
        {
        struct fixedwf_obj bnkmd; - As in FIXEDWF.
        float partial;             - Partial number (fund. = 1).
        };
```

Note that the partial number is specified as a "float" value so as to enable arbitrary partial structures.

7.3.2.2.5. *WAVESHP Objects*

Waveshaping is a technique which enables the synthesis of complex sounds having time-varying spectra. The technique makes use of a form of controlled non-linear distortion. Essentially, the output of one oscillator is scaled by an index (which may be a time-varying function), and then used as an address into a look-up table. The sample taken from the table (which contains the "distortion" function) is then used as a waveform sample and therefore scaled in amplitude and sent to a digital-to-analogue converter. The technique utilizes two oscillator modules, and has its parameters stored in the following structure format:

```
struct ws_object
        {
        char fwf_ind;             - Index into object symbol
                                    table to define waveform
                                    previous to distortion.
        char envel_ind;           - Index into object symbol
                                    table defineing envelope
                                    of waveform after distortion.
        char freq_ind;            - Index into object symbol
                                    table for frequency function.
        int dindex;               - Index of distortion to which
                                    dist_ind function is scaled.
        char dindfn_ind;          - Index into object symbol
                                    table specifying time-
                                    varying function for dindex.
        char distfn_ind;          - Index into object symbol
                                    table indicating waveform
                                    buffer containing distortion
                                    function.
        };
```

7.3.2.3. *OBJECT Symbol Table*

The only valid symbol types for object symbol tables are: FUNCTION and WAVEFORM. Functions at the object level provide the means of specifying how parameters of the micro-structure vary in time. This is essential for sounds to be of musical interest.

Just as with the score symbol table, if object.nsyms equals 0, or if any function named in the table is UNDEFINED, default functions will be substituted. Again, the user is able to over-ride the system defined defaults.

7.3.3. *FUNCTION Files*

Stored functions are used throughout the various hierarchies of the music system to control the variation of parameters over time. Just as there may be many instances of the same score or object in a composition, there may be several instances of the same function. In addition, each instance of the same function could quite conceivably be affecting a different parameter. Functions are stored as a set of straight line segments which approximate a continuous curve. Each file has a *unique, user defined name.* In performance, each function is scaled in both the x and y domains according to application. Since there is no set number of segments in a function, we resort to using two different types of structures for their representation. These are outlined below.

7.3.3.1. *'function' Structure*

This is the "header" structure of a function. It contains the function's name, a "magic" number to identify the file as type FUNCTION, the total number of segments, and a link to the segment data. The actual format of the structure is as follows:

```
struct function
        {
        int magic;                  - Magic number defining
                                      function
        char fname[FNAME_SIZE];  - File name
        int nsegs;                  - Number of segments
        char starty;                - Initial y value
        struct segment *breakpoint;- Pointer to breakpoints.
                                      Functions as:
                                      breakpoint[nsegs]
        };
```

7.3.3.2. *'segment' Structure*

The data for the actual segments is stored in a table of *segment* structures. There is one structure for each segment and all structures are in contiguous memory. Rather than represent segments by integer breakpoints, we have chosen a slightly different approach which is computationally more efficient (when scaling functions during real-time performance). Simply, the y value is stored as would be expected, but the x value is stored as a fractional value representing that segment's relative duration with respect to the complete function. The format of the structure is as follows:

```
struct segment
        {
        float reldur                - Relative duration of
                                      segment
                                      (0. <= reldur <= 1.)
        char yval;                  - End Y value of segment
        };
```

7.3.4. *WAVEFORM Files*

Waveforms are a particular form of function which we choose to treat differently than FUNCTION files. In the case of a waveform, we store the function as a series of point samples, where the number of points equals the size of a synthesizer waveform buffer (i.e., 2048). Consequently, only the y value of the function need be stored, the x value being the index into the table. Besides storing the actual function data, the *waveform* structure also contains a "magic" number to distinguish it as type WAVEFORM, and the actual waveform name. The data format for waveforms is:

```
struct waveform
        {
        int magic;                   - Magic number.
        char fname[FNAME_SIZE]; - File name.
        int wfsamp[WFB_SIZE];       - Waveform samples.
        };
```

8. *The Base Structure in Practice: Selected Examples*

8.1. *Introduction*

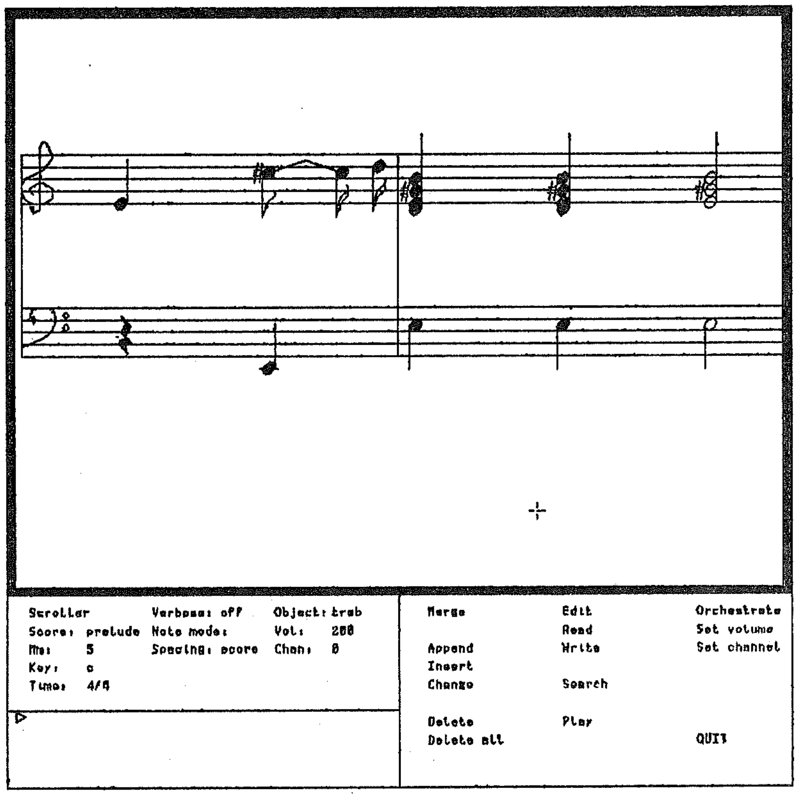
The bulk of this report has dealt with the foundation structures. The purpose of this chapter is to present selected examples of high-level structures which can be built on this base. It is *not* our intention to present a comprehensive description of a music system. Rather, we hope to tie together some of the concepts presented, as well as demonstrate that the foundation structures developed through the course of this report do, in fact, function as desired.

The format that we will follow will be to illustrate specific features by taking examples from programs which have been written on the system. Virtually all of the programs described are in prototype form, for which we make no apology -- given our attitude to an iterative approach to system development. Of the programs used as examples, many have not been written by the author. Where this is the case, we give appropriate acknowledgement. We justify their inclusion in this context on three points: all were written in consultation with the author; all make use of the structure developed in this report; and -- perhaps most important -- that several different programmers can write compatible software using these base structures is one of the strongest indications of their success.

8.2. *Examples*

A founding principle of our design approach has been that we should be able to represent data in different ways. This is important both in developing new representations, and where the user might simply want to view his data in a different way due to context. Figures 8.1 through 8.3 demonstrate the flexibility of our internal representation for scores. All three scores -- common music notation (CMN), "piano roll", and tree structure -- are drawn from the same type of data. No special graphic information is saved (or necessary).

In addition to the above, it is important to enable data to be represented at different *conceptual* levels. One example which we have discussed is the representation of object data in the acoustic domain, on one hand, and the perceptual domain, on the other. The next two figures demonstrate these two levels of representation. Figure 8.4 shows a graphic representation of the parameters of a frequency modulation object (in the acoustic domain) while 8.5 shows a vosim object represented using the cardinal quadrangle (perceptual space).



```
Scroller         Verbose: off    Object: treb
Score:  prelude  Note mode:      Vol:    200
Ms:     5        Spacing: score  Chan:   0
Key:    c
Time:   4/4
```

Figure 8.1: Representation of score in common music notation. (This, as well as all other examples using CMN are from the program "ludwig", by W. Reeves and R. Pike.)

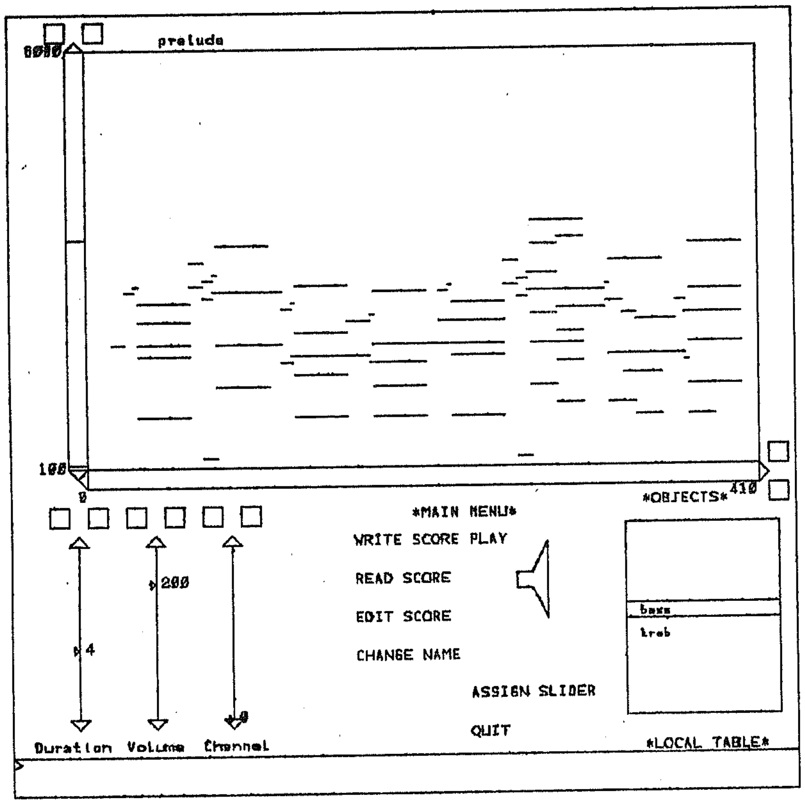


Figure 8.2: Representation of score using "piano-roll" type notation.
(From the program "scored", by S. Hume.)

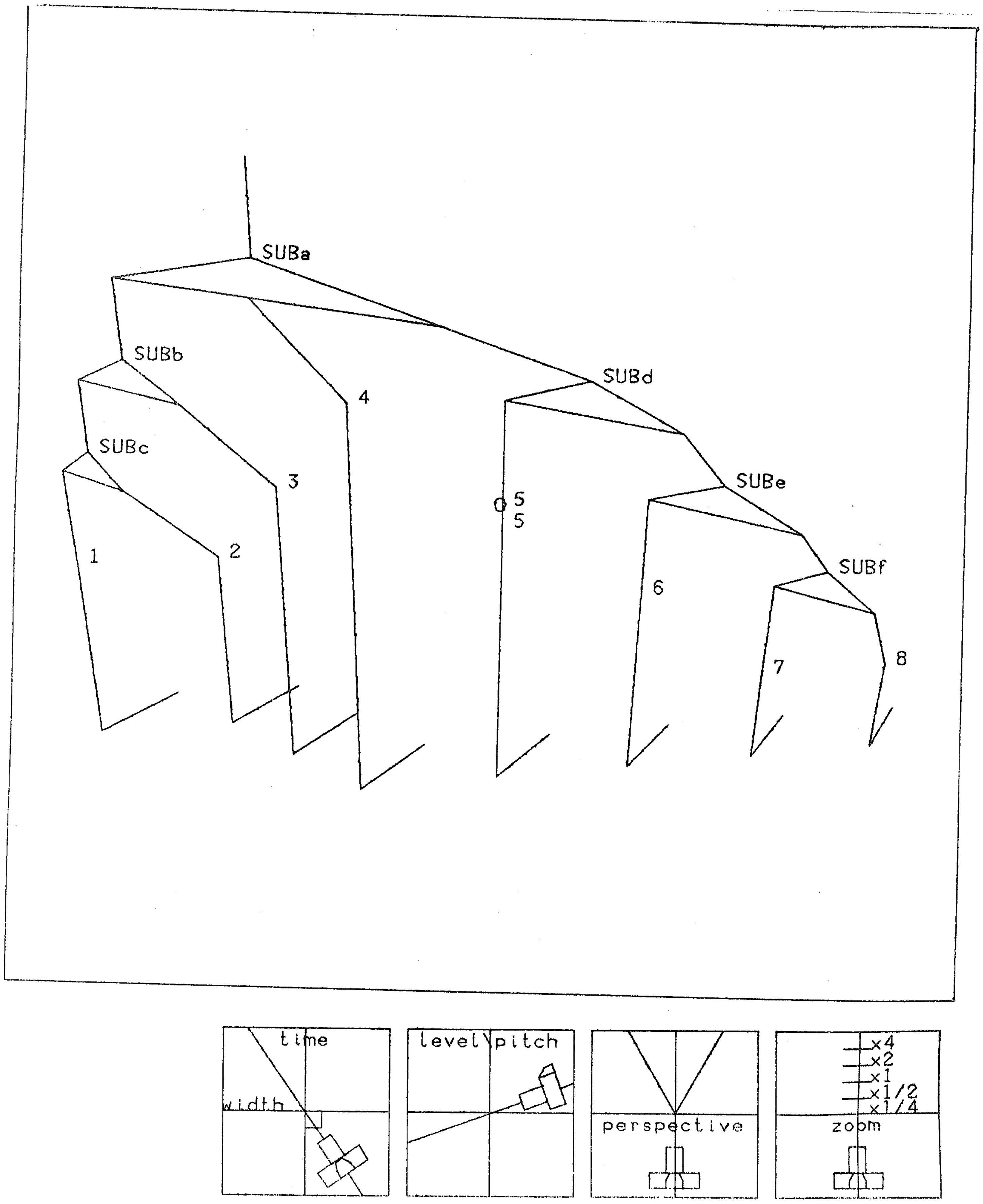


Figure 8.3: 3-D Tree representation of score
(From the program "treed", by A. Kwan.)

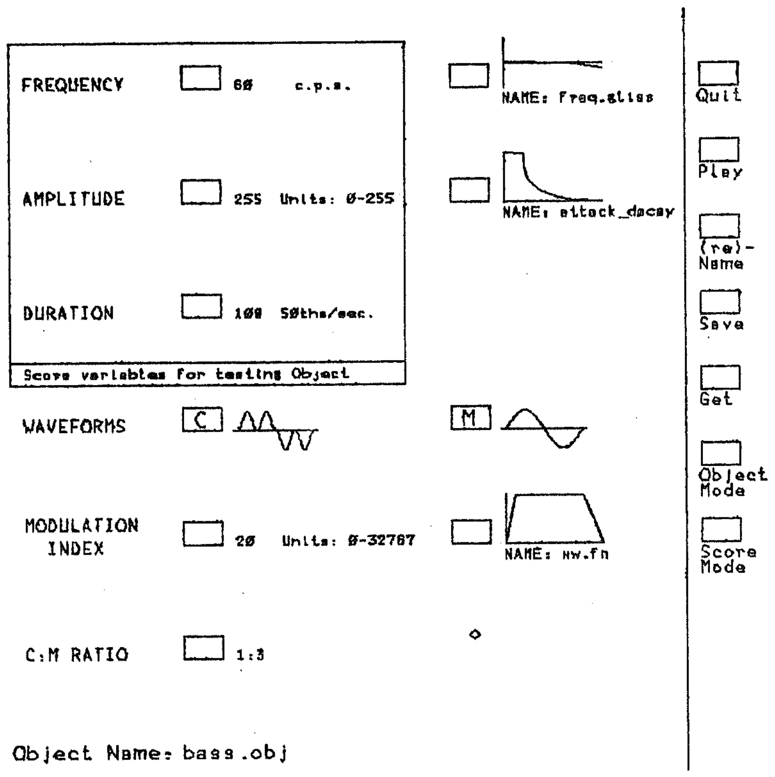


Figure 8.4: Representation of an (FM) object at the <u>acoustical</u> level.
(From the program "objed", by W. Buxton.)

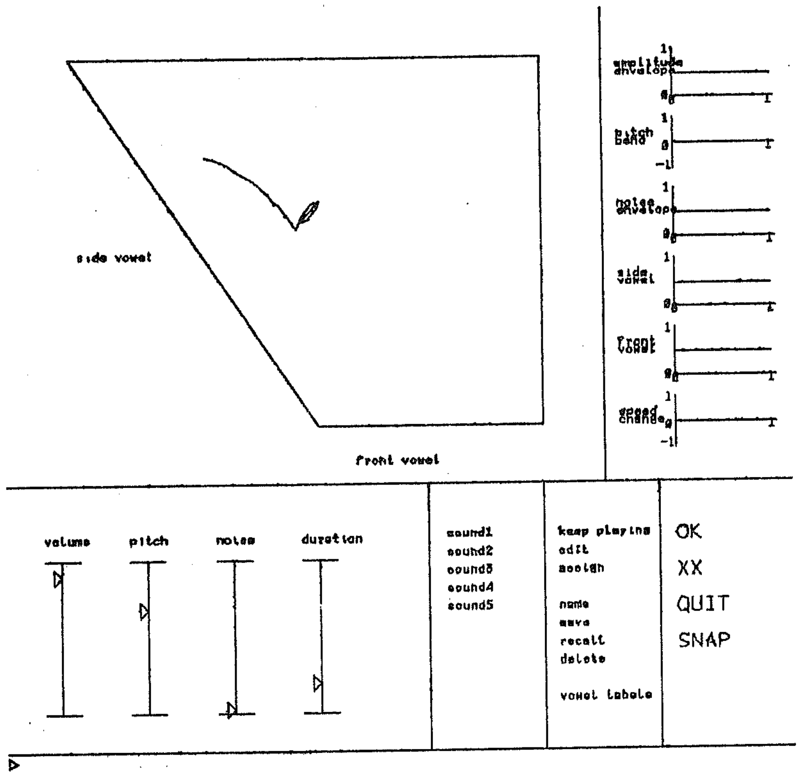


Figure 8.5: Representation of a VOSIM object at the <u>perceptual</u> level (making use of cardinal vowel quadrangle of Jones, 1956).
(From the program "voice" by P. Chow and D. Galloway.)

In many cases where the data structures are "fuzzily" defined, it is difficult for the user to make a clear distinction between *data* and *process*. Perhaps the best example of this is where a user can not keep separate the difference between a score and the process which created it. We attempt to emphasize the compatability of scores to different forms of notation (such as illustrated in previous examples) regardless of whether the score was created stochastically, using a composition program, or deterministicaly by the user, note-by-note. This is illustrated in the next two figures. Figure 8.6 shows the partial specification of a score (frequency characteristic using a tendency mask) to a compositional program. Figure 8.7 shows a CMN representation of part of the resulting score. There are several interesting features in this example. First, the pitches generated by the composition program come from the continuous frequency domain. Nevertheless, they can be displayed in relation to the discrete steps of a diatonic system. Most important, the data is not changed, and is displayed by a "best fit" paradigm. In addition, we see that the stochastically generated data can -- to the horror of purists -- be deterministically edited, merged, concatinated, etc. with other scores (regardless of how created).

Figures 8.4 and 8.5 illustrated the representation of data at different *conceptual* levels; however, except for Figure 8.3, none of the examples yet discussed have dealt with the representation of the *structural* hierarchies of scores, or the use of instantiation. The next three figures illustrate both. In Figure 8.8, we have three bars of music. Each bar is a separate score. The first functions as a *master score*, while the second bar -- a major chord -- functions as a *sub-score*. In the third bar is a new score which is an example of the master score being "scorchestrated" by the chord of the sub-score. The result is that we now have a sequence of major chords following the pattern defined by the master score. Thus, we see *scorchestrate* simply means "orchestrate with a score". Note, however, that while there are three *instances*, there is only one *copy* of the major chord in memory. Furthermore, each instance is a "sub-tree" in a hierarchy, whose "parent mode" is a note in the master score. In this regard, note how the pitch and duration information of the parent has been imposed on the descendents. Figure 8.9 further illustrates the benefits of instantiation. Here we have simply made the sub-score into a minor chord (one action), and as a result all instances of that sub-score are automatically modified. Figure 8.10 is similar to the above except in this case, the sub-score is a three note motif. We see, then, that the *scorcestrated* sub-score becomes a canon-like figure.

Up to this stage, the prime intent of the examples has been to illustrate points dealing with representation. In the examples which follow, we would like to change our approach and centre on issues of interaction. In so doing, we remind the reader of the importance of the dynamics of this interaction. Viewing the examples as frozen frames is somewhat akin to viewing a three-dimensional sculpture in a two-dimentional photograph. Most of the examples make use of an interactive score editor called "ludwig" (sic.), written by my colleague Bill Reeves. The choice of using this score editor for our examples does not imply an afinity for, or bias towards, CMN. Rather, just as with beginning users, CMN is helpful to present new concepts in a familiar environment. Other -- often more appropropriate -- forms of notation have already been demonstrated.

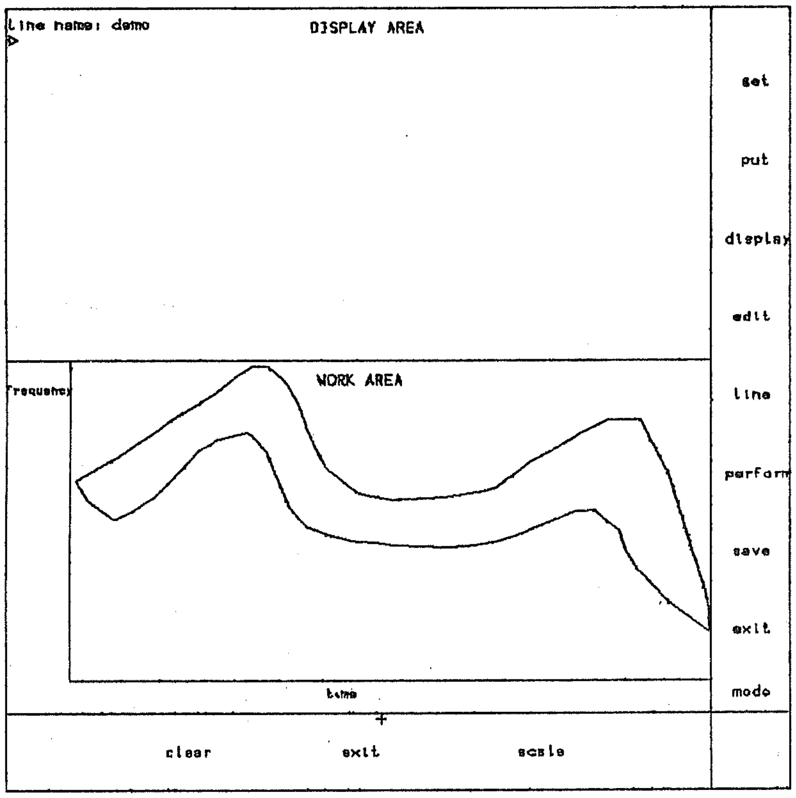
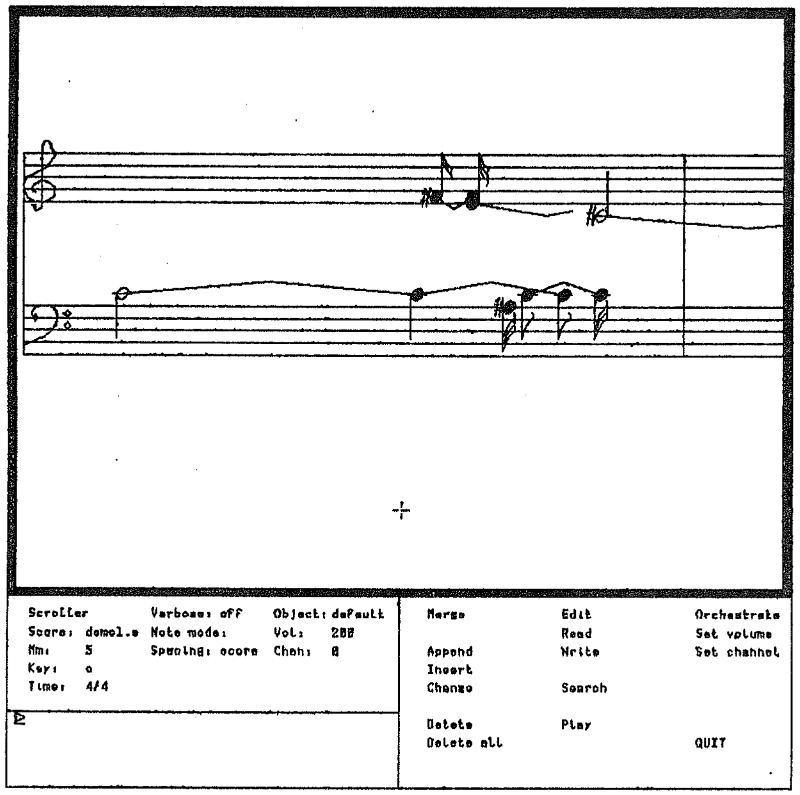


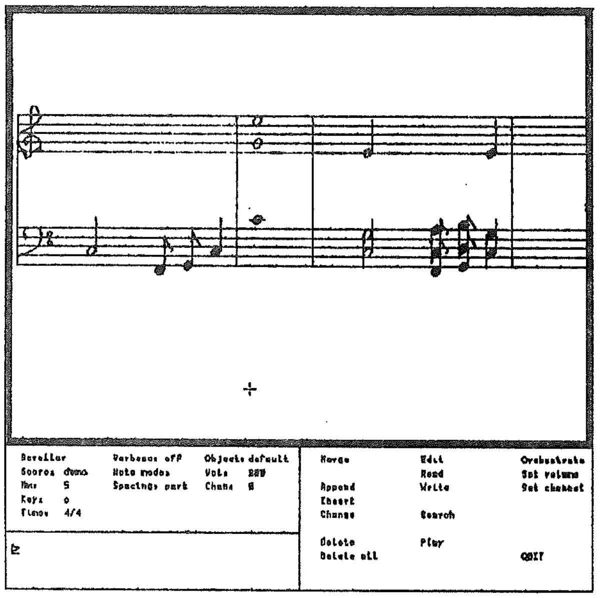
Figure 8.6: Generating a score using a compositional program. Note that data is compatible with that generated by other processes. Data generated is displayed in CMN, for example, in Figure 8.7.
(From the program "amucs" by M. Green.)



```
Scroller        Verbose: off     Object: default
Score:  demo1.s Note mode:       Vol:    200
Nm:     5       Spacing: score   Chan:   0
Key:    c
Time:   4/4
```

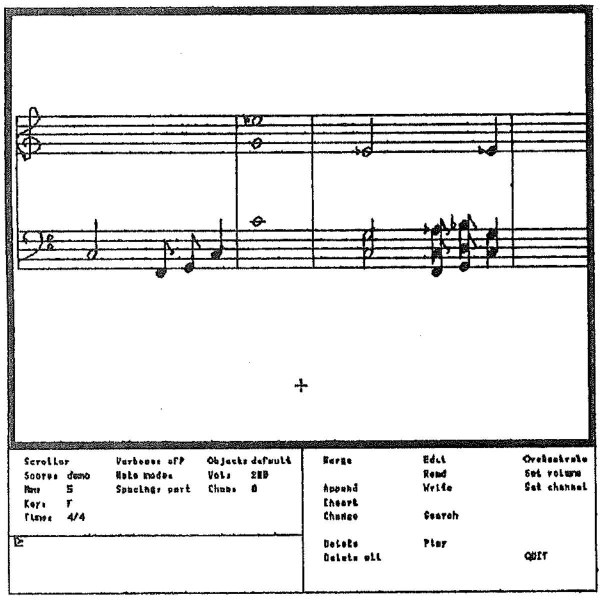
Figure 8.7: Computer generated music displayed using CMN. Data was generated using program illustrated in Figure 8.6.

Scorchestration: three examples.



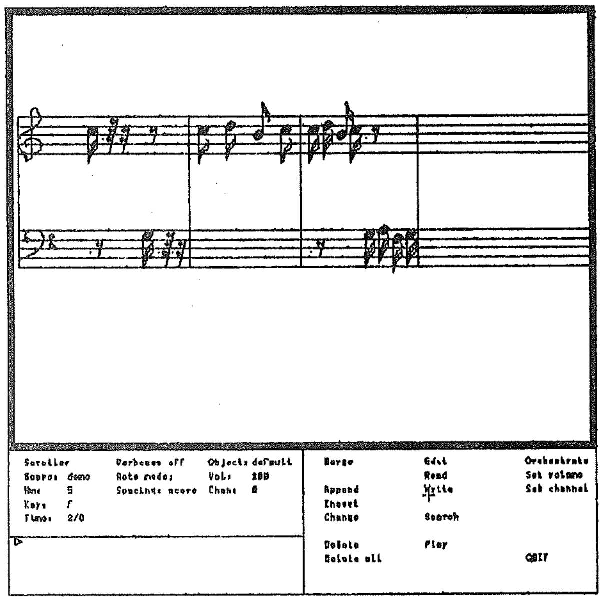
```
Scroller       Verbose: off    Objects default
Score: demo    Note mode:      Vol:  255
Mm:    5       Spacing: part   Chan: 0
Key:   C
Time:  4/4
```

Figure 8.8: "scorchestration", example 1. Notes in bar 1 are scorchestrated with those in bar 2. Resulting score is shown in bar 3. Note how major chord of bar 2 has been transposed in both pitch and time.



```
Scroller       Verbose: off    Objects default
Score: demo    Note mode:      Vol:  255
Mm:    5       Spacing: part   Chan: 0
Key:   F
Time:  4/4
```

Figure 8.9: "scorchestration" example2. Same as figure 8.8 except that e5 has been made flat. Note how this one change is reflected in each instance of the chord in bar 3.



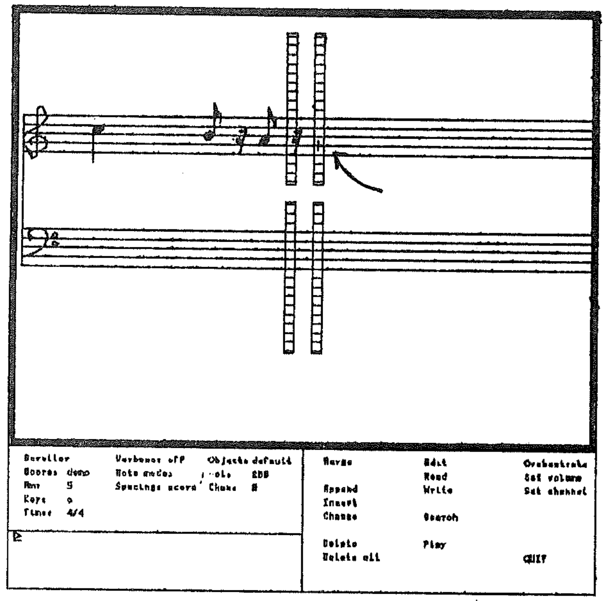
```
Scroller       Verbose: off    Objects default
Score: demo    Note mode:      Vol:  255
Mm:    5       Spacing: score  Chan: 0
Key:   F
Time:  2/8
```

Figure 8.10: "scorchestration" example 3. As above two figures excepting that a melodic, rather than harmonic, structure is used in scorchestrating the master score.

While many systems utilize piano-like keyboards as input devices (eg., Vercoe, 1975), we feel that too heavy a reliance on such transducers "locks the user in" to a particular mode and notation in composing. We would like, therefore, to investigate different input techniques which do not go to the other extreme of typing (which is somewhat akin to an obscene gesture in our approach). In this regard, we find recourse in graphic techniques. The next three figures illustrate one type of such interaction which is supported by the base structures which we have provided. Figure 8.11 shows the *tracker - cross* being positioned over the desired pitch. This is accomplished simply by moving a *cursor* on the *graphics tablet*. On depressing a button on the cursor, a "marker" note appears at the indicated pitch. Concurrently, the tracking-cross is replaced by a sequence of notes. This is shown in Figure 8.12. This sequence of notes (shown to the right in Figure 8.12) "tracks" or follows the motion of the cursor on the tablet. By placing the note of the desired duration over the marker note, and releasing the cursor button, a note is input. This is shown in Figure 8.13. Ledger lines, tail direction, and spacing are -- naturally -- taken care of by the program. While our long-winded explanation may indicate otherwise, after about a five minute introduction a new composer can input notes as fast or faster than on manuscript paper. This has been verified with several musicians, including Yehudi Menhuin (one of our first "guinea pigs").

Since one of our functions is to provide an efficient score editor, it is important to enable the composer to "get around" the score; to access notes both in or out of the current *viewport*. Examples of techniques for doing so are presented in the next three figures. To begin with, any note in the current viewport can -- for editing purposes -- become the *current* note by pointing at it and depressing a button on the cursor. That is to say, the graphics system is flexible enough to do hit detection with immediate response. If, on the other hand, the note we want is not in the current viewport, we can *scroll* the score across the screen in real-time at the touch of one of the hardware *sliders*. This is how, for example, we moved the quarter rest from the left of the viewport (Figure 8.14) to the right (Figure 8.15). This ability to scroll through the score -- especially during performance -- is a good example of the importance of our reliance on a display capable of dynamic graphics. A storage tube or most video devices could simply not support this type of interaction. This is also the case for the next example. Again, we want to examine part of the score in the current viewport. This time we take an alternative approach. We point at the *light button* "search", and press a button on the cursor. What appears on the screen (seen in Figure 8.16) is a "time-line" representing the entire duration of the score. (This appears at the bottom of the score viewport in the example.) On this time-line, we see two "angle brackets" which indicate what portion of the score is in the current viewport. By placing the tracking-cross (which has become a mini magnifying-glass icon -- to indicate "searching") anywhere on the timeline, and depressing a button on the cursor, the viewport will move and centre on that portion of the score. The use of labels (not shown) further strengthens this technique, providing something analogous to orchestral rehearsal marks. Of course all of these techniques also apply to forms of notation other than CMN.

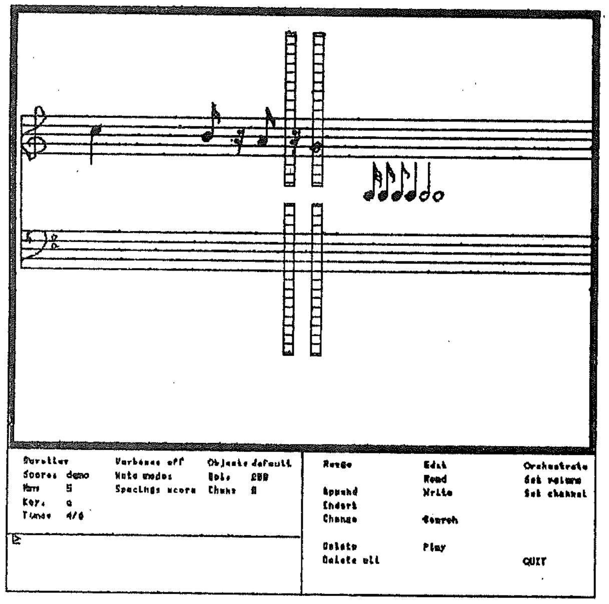
In the above example, we saw a case where the tracking-cross was replaced by an "icon" in order to convey some information in a pictoral manner. The graphics package used enables us to make great use of this facility in order to communicate without wordy messages. Other icons used, for example, are: a buddah (to indicate patience when a user must wait), a "?" (when an unintelligible request has been made), a loudspeaker (to show "perform" program is active), and a picture of a terminal (to cue



```
Scroller          Verbose: off     Objects default
Score: demo       Note mode:       Vol:   200
Ann:   5          Spacing: score   Chan:  0
Key:   0
Time:  4/4
```

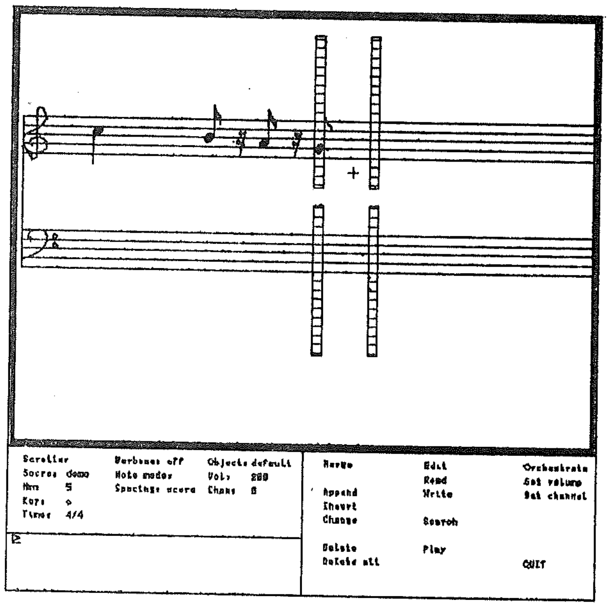
Appending a Note to a Score; an example.

Figure 8.11: step 1: the tracking cross is placed over the desired pitch (g4), as indicated by the arrow.



```
Scroller          Verbose: off     Objects default
Score: demo       Note mode:       Vol:   200
Ann:   5          Spacing: score   Chan:  0
Key:   0
Time:  4/4
```

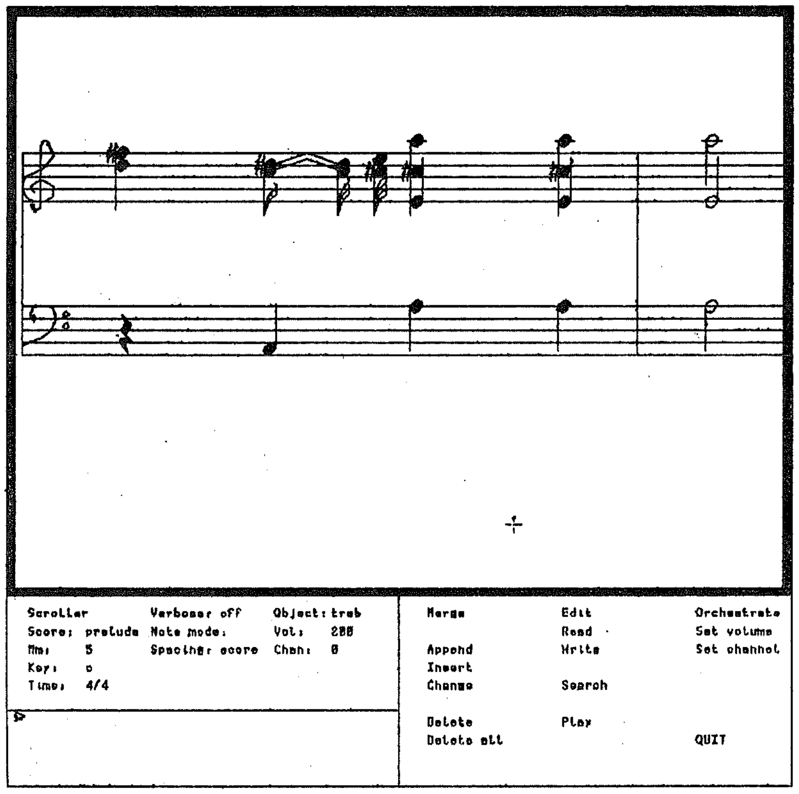
Figure 8.12: step 2: on pushing a button on the cursor a "marker" note appears over g4, and the tracking cross becomes the series of notes seen to the right.



```
Scroller          Verbose: off     Objects default
Score: demo       Note mode:       Vol:   200
Ann:   5          Spacing: score   Chan:  0
Key:   0
Time:  4/4
```

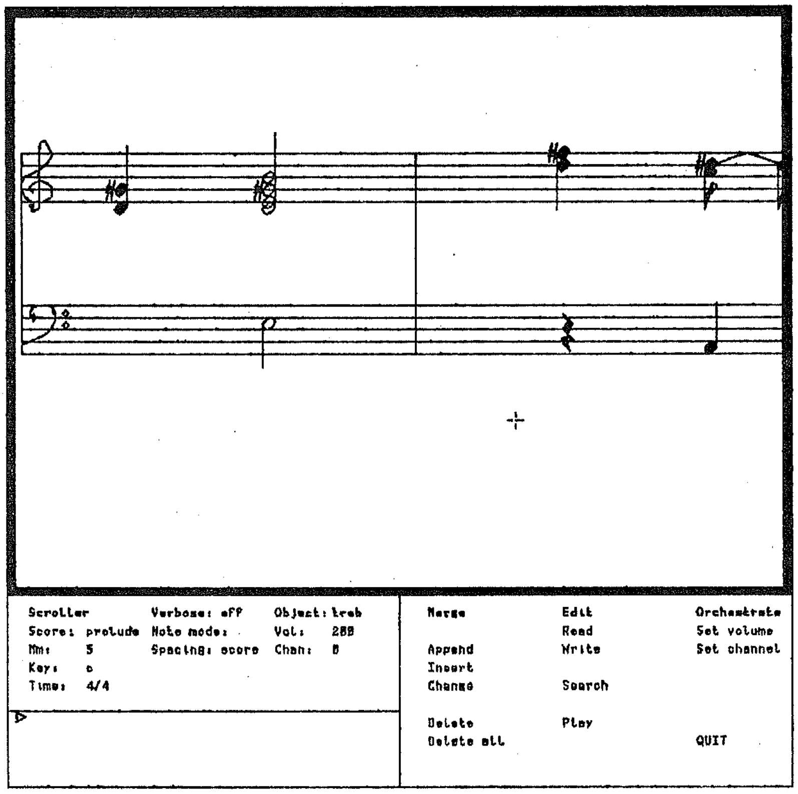
Figure 8.13: step 3: on placing eighth-note of "tracking-notes" over the "marker" note and then pushing a button on the cursor, an eighth-note g4 is entered. Note that the tracking cross is restored.

Scrolling: an example.



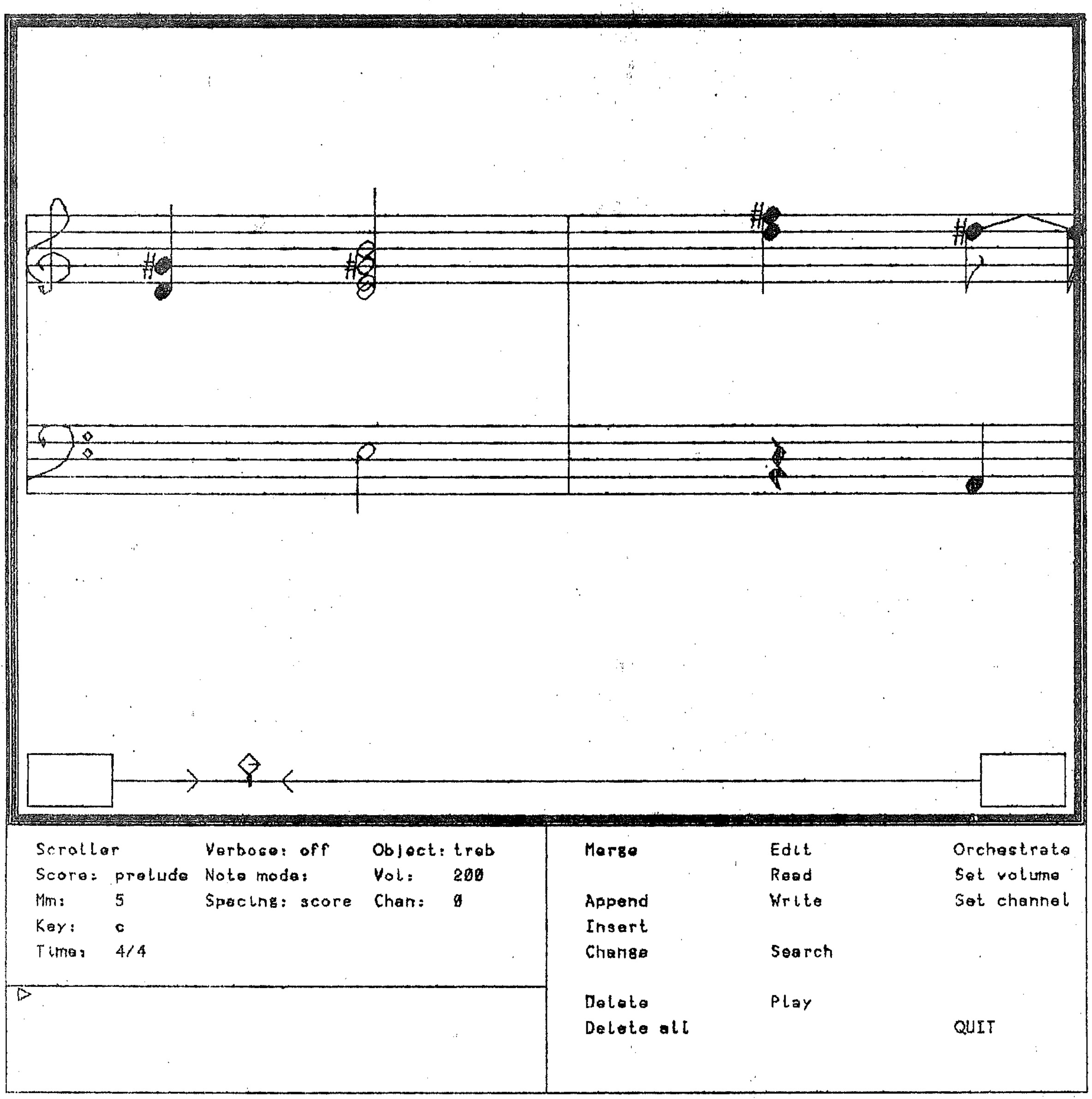
```
Scroller       Verbose: off    Object: treb
Score: prelude Note mode:      Vol:    200
Mm:    5       Spacing: score  Chan:   0
Key:   c
Time:  4/4
```

Figure 8.14: The "before" situation. On moving the slider (not shown), the score will scroll across the screen. Thus, we see that we have, in effect, a moving window into the score.



```
Scroller       Verbose: off    Object: treb
Score: prelude Note mode:      Vol:    200
Mm:    5       Spacing: score  Chan:   0
Key:   c
Time:  4/4
```

Figure 8.15: The "after" situation. Note how quarter-note rest (and rest of score) has scrolled to the right. Scrolling is just one of the benefits of dynamic graphics.



```
Scroller        Verbose: off     Object: treb
Score:  prelude Note mode:       Vol:    200
Mm:     5       Spacing: score   Chan:   0
Key:    c
Time:   4/4
```

Figure 8.16: searching a score. Time-line at bottom of viewport represents entire score. Section of score currently in wiewport is that delimited by angle brackets on time-line. Viewport can move by placing "magnifying glass" at any possition on time-line and pushing button on the cursor.

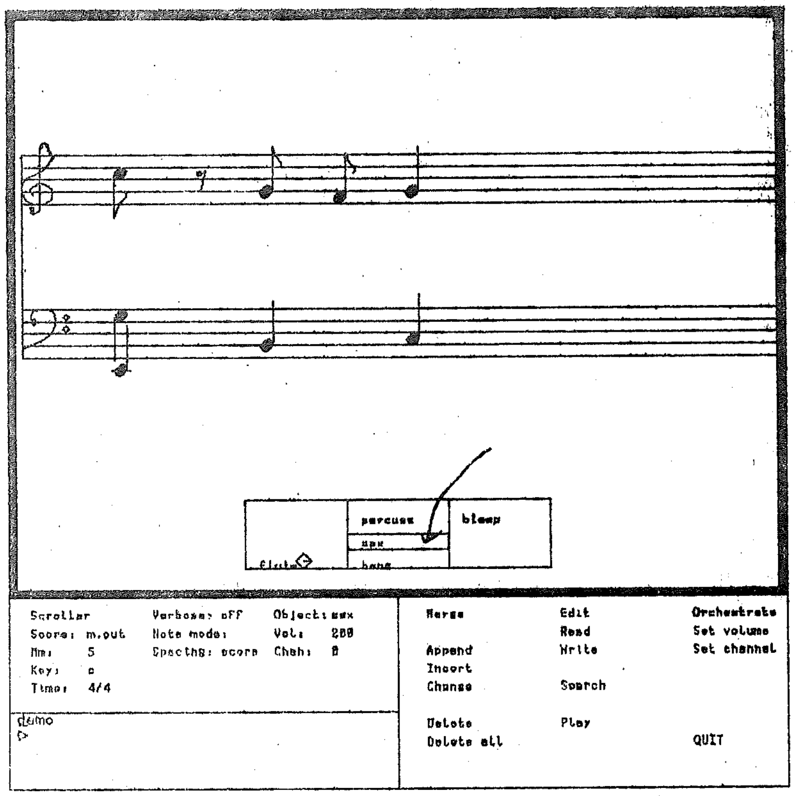
the user that he may type a message). While on the topic of typing, it is worth pointing out that in *ludwig*, the user has continual access to system level commands. The bottom left-hand corner provides a space for scrolling alpha-numeric text. At nearly any time the user may type a system command which will suspend *ludwig*, execute, and then return control. Again, this is an invaluable feature for non-linear thinkers (i.e., most users), which is provided by the UNIX operating system.

As any good teacher knows, one of the best aids in presenting new concepts is a good analogy. This is especially true for concepts used in human interfaces to computers. One technique that we use which has a good analogy in the "real world" is what is often called a "paint pot" technique (Baecker, personal communication). The next set of figures shows how this technique can be used in orchestrating the notes of a score. After selecting the "orchestrate" lightbutton, the tracking-cross becomes a paint-brush, and our palette of timbral colours (objects) appears in the box at the bottom of the viewport (Figure 8.17). The "colour" currently on the brush is that object between the double bars in the centre of the "palette". (In this case "sax".) Simply pointing at notes with the brush and depressing a button on the cursor, will orchestrate them with the current object. (In this case, an analogy to a spray-can would be more appropriate). At any time, the user is able to change the "colour" on his brush either by dipping into his palette (i.e., pointing at the desired object in the box), or scrolling through the list of objects -- using a hardware slider -- until the desired object lies between the double bars in the centre. The figures show the former case: the object pointed at in Figure 8.17 ("flute")appears as the current object in figure 8.18. The object pointed at in Figure 8.17 appears in the centre box in Figure 8.18.

There are a few significant points which arise out of the previous example. First, notice that in accessing the various object files, no typing was done. (For that matter, notice the lack of typing in all of the previous examples.) Second, notice that there was no burden on the user's memory to recall the names of the objects in his directory. In this we see the exploitation of our ability to tag all file types with a *magic number*. (A process is invoked which makes use of this magic number in extracting all *object* files from the directory and listing them -- and them alone -- as the "palette".) Another point to notice is how a simple transducer, the *slider* has been used for several different functions in different contexts. Here we have an example of where forethought to custom-built but generally-applicable hardware has paid off. In the last example, for instance, the use of sliders to change the "current" object in the palette meant that the cursor need not move down. Orchestration is carried out with the cursor in one hand, and the object selection -- using the sliders -- with the other. The resulting economy of motion results in a more smooth, efficient, and congenial interface. Finally, it is important to note that what was described is an instance of a general protocol. Once learned, the same technique can be used for several other parameters; channel number and volume, for example. Again, this type of consistency -- enabled by the underlying structure -- can only accelerate the rate at which the novice user becomes acclimatized to the system. The benefits resulting from this concentration on ergonomics is obvious, and must pervade the entire human interface.

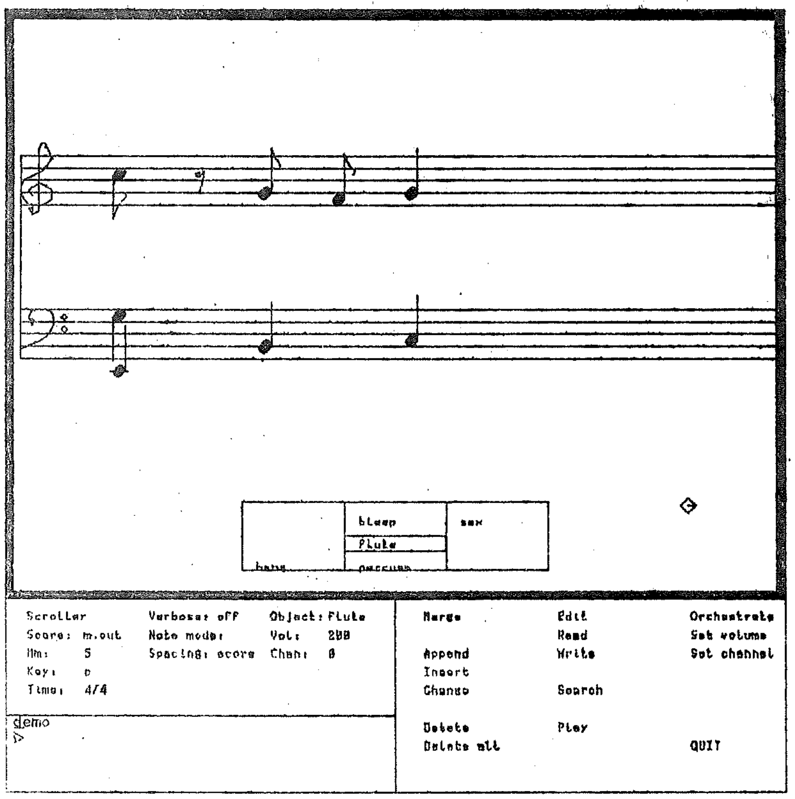
In our final example, there is one additional point which we would like to make. Too often systems are designed to accomodate only "rational" behavior. That is, when the user does something "wrong" the process either aborts (as in most "batch" systems) or

Orchestration using "paint-pot"



```
Scroller      Verbose: off     Object: sax
Score: m.out  Note mode:       Vol:    200
Nm:    5      Spacing: score   Chan:   0
Key:   c
Time:  4/4
demo
▷
```

Figure 8.16: palette of timbres (list of objects) appears at bottom of viewport. "Colour" currently on brush is object between double lines in palette (sax - see arrow). Notes are orchestrated simply by pointing at them with the "brush", i.e., tracking-cross.



```
Scroller      Verbose: off     Object: flute
Score: m.out  Note mode:       Vol:    200
Nm:    5      Spacing: score   Chan:   0
Key:   c
Time:  4/4
demo
▷
```

Figure 8.17: object to be used in orchestration is changed simply by pointing at the desired object name in the palette. "Flute", being pointed at in Figure 8.16 is now "current" object.

suspends until a legal command is given. All of this can only lead to the user building-up a fear of making mistakes. This, we feel, is completely antithetic to the establishment of a congenial human interface. The system must not only accept "irrational" commands, but attempt to "fix" them and advise the user about what he did wrong. We believe that the beginner learns by doing, and if he is paranoid about doing something wrong, or crashing the system, he does -- and therefore learns -- very little. Again, let us emphasize, five minutes of doing something -- mistakes and all -- usually surpasses half an hour of reading about how to do it. We conclude by looking at a specific example.

Many parameters in the musical hierarchy can vary over time and are controlled by stored functions. Such parameters range from the micro-level (amplitude envelopes) to the macro-level (the dynamic contour of an entire score). Many other systems also make use of stored functions in a similar manner. For illustration we will take an example from the batch-processing based MUSIC 4BF system (Howe, 1975b). Here we find that Howe (pp 194-200) takes about six pages to describe the workings and definition of stored functions [30]. Even then, if the user makes an error, the chances are that the job will be aborted. Alternatively, by taking a limited-but-strong approach, we can use graphics in combination with computer-initiated dialogue to guide the user in carrying out his task. For example, Figure 8.19 shows a function being drawn as a free-hand curve using the graphic tablet. If, as is shown in Figure 8.20, a meaningless curve is drawn by the user, the system can attempt to "understand" what was drawn and do something reasonable. The result in this case is shown in Figure 8.21. The user will clearly see that what the computer returned was not what he drew. A little thought or a well placed question will clear up his problem. He may then either accept what the program returned (which is, in fact, a legal function) or try again -- reassured that he not only may make mistakes, but he might even learn something if he does!

9. *Conclusions*

9.1. *Summary*

The general theme this report has been that of making computer-based tools accessible to the computer-naive but application-sophisticated user. It has been concerned, therefore, with problems in human engineering. Specifically, it has dealt with determining how to evolve the base structures to support such tools. In this regard, we have oriented our discussion around the problems of a particular application area: that of music composition.

The first section of the report (Chapter Two) dealt with issues which were mainly general in application. The main point stressed, however, was the importance of understanding the user for whom the tool is intended. The appropriateness of different forms of dialogue was discussed in terms of different types of users. It was suggested that a particular methodology be adopted so as to accumulate the knowledge to enable

[30] We do not take this example for the purpose of criticism. MUSIC 4BF has functioned as well as could be expected and made good use of the resources available. The point is, we now have different resources and should make better use of them.

Error Correction: an example

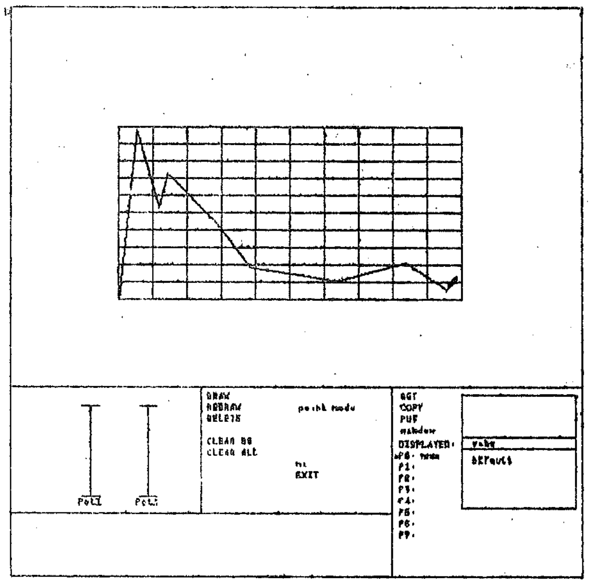


Figure 8.19: Functions can be drawn free-hand.

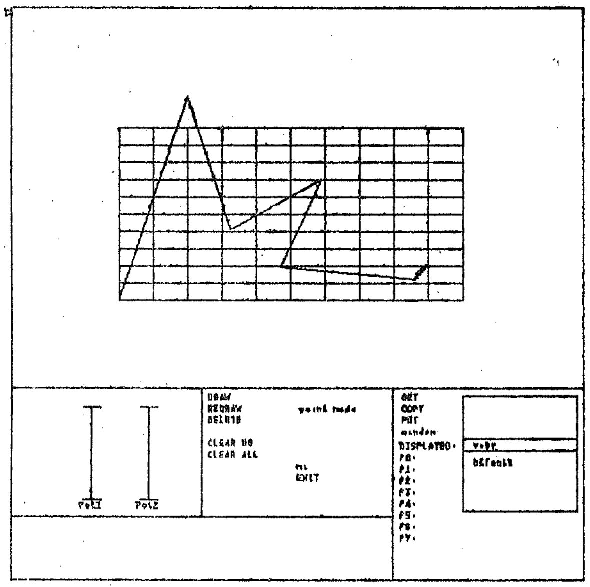


Figure 8.20: Beginner users may draw envelopes which are not functions, such as the one illustrated -- which goes backwards in time.

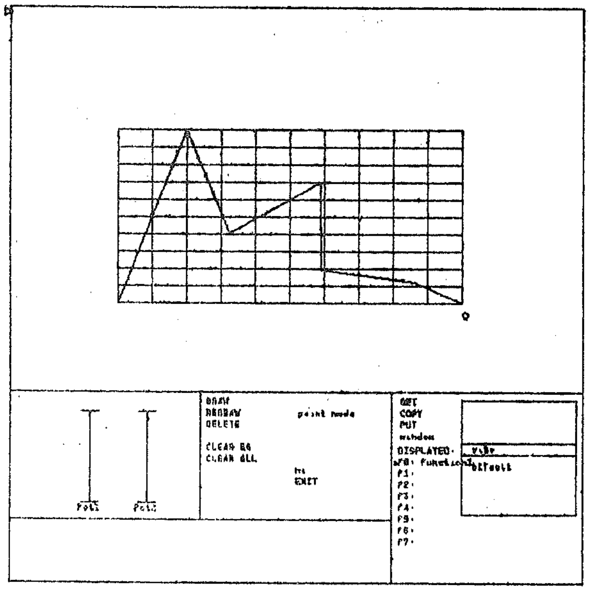


Figure 8.21: Envelope drawn in Figure 8:20 is corrected by the system. The function is legal, and the user probably learns something. More important, he loses his fear of making mistakes.

the design of a congenial user interface. This methodology was based on an iterative strategy of "quick-and-dirty" implementation followed by extensive evaluation and redesign. It was argued that, given suitable tools, such an approach would be economical in many cases. Some of the main considerations -- such as the operating system, interaction, and special-purpose hardware -- were then discussed.

In order to establish a context for the application-specific aspects of our discussion, we then proceeded to present a review of the literature in computer music. This was divided into a discussion of compositional problems, on the one hand, and problems of sound-synthesis, on the other. In terms of the former, problems were pointed out with respect to musical decision making on the part of computer programs. This was contrasted with systems in which the computer plays the role of a composer's "assistant". In terms of sound synthesis, the trade-offs between the various approaches (digital, hybrid, and mixed-digital) were discussed. In particular, the appeal of the high sound quality and interaction available from the mixed-digital approach was emphasized.

The combined effect of the chapters discussed thus far was to motivate the major design decisions made in the next section, Chapter Four. Here a basic task taxonomy for music composition was presented. The taxonomy considers composition in terms of four tasks: definition of a palette of timbres, definition of a pitch/time structure, orchestration, and performance. The simplicity of the conceptual framework provided by this analysis is one of its most important features. Within this framework, we went on to develop several important points. Central among these was the importance of providing base structures which enable the user to have as much freedom as possible in the strategies which he employs in musical design. In particular, it was shown how flexibility could be provided in the order in which the basic tasks were undertaken, how the composer could work with and perform incomplete scores, and how he could be allowed to address himself -- with equal ease -- to any structural unit within the score. These features were made possible as the result of several key design decisions. These include the adoption of a sophisticated system of defaults and the introduction of a hierarchical representation of scores. In addition, the choice of a strong-specific -- as opposed to weak-general -- approach to sound synthesis contributed a great deal of power to the design. The main result was to minimize the problems of acoustics thereby enabling the composer to focus attention on higher-level musical decisions.

Details of the basis for the above structures was presented in the next section of the report (Chapters Five to Seven). The key features of this foundation were seen to be: the ability to support a high degree of interaction, complex control and data structures, good graphics support (to allow flexibility in external representations), and the digital synthesizer. All of these features were then brought together and demonstrated in selected examples in Chapter Eight.

9.2. *Discussion*

It is felt that the prime contribution of the work described in this report is in the chosen application area, computer-music. The most significant points will be discussed below. However, in doing so, it is important to keep in mind the attitude and approach to design which has made these contributions possible. This is the iterative approach, with the bulk of our attention focused on the user. We feel that the success of the approach in the current system is indicative of a more general applicability. It is hoped that this work may stimulate other designers to adopt the same approach.

In terms of the application area, it is felt that the heart of our approach lies in the basic division of composition into the four sub-tasks mentioned. A clear context is provided in which the both the composer and designer can view their activities. Having so divided the tasks, data were then structured accordingly. This gave rise to the notion of *object*, *score*, and *musical event*, all of which are key features of the design.

In terms of *objects*, we feel that limiting the system to a few well chosen acoustic models has been a key design decision. This enables the composer to escape the all too common domination of "sound" over "music". By limiting ourselves thus, we have been able to develop a state-of-the-art sound synthesizer which has proven reliable, economical, and of high audio-quality, and which enables us to achieve the high degree of interaction which we feel is so necessary.

In terms of scores, one of the key concepts which we have introduced is the ability to deal with any (user-defined) sub-set of a score in the same manner, and with the same ease, as with a single note. This we have been able to accomplish by allowing scores to be structured as arbitrary hierarchical (tree) structures. Scope of operators (such as "play" or "delete") then simply apply to the descendants of the indicated node. Furthermore, with both *scores* and *objects* we have been able to make use of instantiation. Thus, for example, an entire score can be re-orchestrated simply by changing a file name.

Throughout, our intention has been to facilitate the use of different forms of external representation *without* any changes to the internal representation of the data. The preliminary results shown in Chapter Eight demonstrate a reasonable degree of success. This success, however, is equally due to the basic graphics and computing environment, as to the actual information structures used. Acknowledging this fact brings us to a final conclusion of our report: the importance of developing better tools for systems development, tools which enable the designer to view and define his problems in ever more succinct terms. This will only come about when systems for research and development are designed as integrated *environments*. Without such integration, the economics of design will continue to dictate a continuation of the "get it right the first time" attitude.

The music system described in this paper has been designed and implemented in a relatively short period of time (eighteen months for both hardware and software). This simply would not have been possible in most other computing environments. Systems such as our own and that developed by the Learning Research Group at XEROX Parc (Kay, 1977 and Ingalls, 1978) are a good examples of what is possible when a system is designed as an integrated whole. It is hoped that the work described in this report might in some way stimulate further research in this direction. Then -- and perhaps only then -- we may see computers realizing their full potential in *serving* non-specialists -- such as composers.

10. *References and Bibliography*

Alles, H. (1978). A Modular Approach to Building Large Digital Systems. *Computer Music Journal* 1.4: 10-13.

Alles, H. & di Giugno, P. (1978). A One-Card 64 Channel Digital Synthesizer. *The Computer Music Journal* 1.4: 7-9.

Alonso, S., Appleton, J., & Jones, C. (1976). A Special Purpose Digital System for Musical Instruction, Composition and Performance. *Computers and the Humanities* 10: 209-215.

American Standards Association (1960). American Standard Acoustical Terminology. *S1.1-1960*, New York: American Standards Association Inc.

Appleton, J. & Perera, R., Eds. (1975). *The Development and Practice of Electronic Muisic.* Englewood Cliffs, N. J.: Prentice-Hall Inc.

Backus, J. (1969). *The Acoustical Foundations of Music.* New York: W. W. Norton & Co.

Baecker, R. M. (1969). Interactive Computer-Meditated Animation. *TR-61*. Cambridge: MIT Project MAC.

Battier, M. & Arveiller, J. (1976). *Musique et Informatique: Une Bibliographie Indexee.* Paris, University of Paris VIII.

Beauchamp, J., Pohlamann, K., & Chapman, L. (1975). The TI 980A Computer Controlled Music Synthesizer. *Proceedings of the Second Annual Music Computation Conference*, Vol. 3, Hardware for Computer-Controlled Sound Synthesis. Urbana: University of Illinois, pp. 1-36.

Benade, A. (1976). *Fundamentals of Musical Acoustics.* New York: Oxford University Press.

Bennett, J. L. (1972). The User Interface in Interactive Systems. In Cuadra, C. A., Ed., *Annual Review of Information Science and Technology*, Vol. 7. Washington, D. C.: American Society for Information Science, pp. 159-196.

Buxton, W. (1975). *Manual for the POD Programs.* Utrecht: Institute of Sonology.

-------- (1977a). A Composer's Introduction to Computer Music. *Interface* 6: 57-72.

-------- (1977b). *Computer Music 1976/77: a directory to current work.* Ottawa, The Canadian Commission for Unesco.

Buxton, W. & Fedorkow, G. (1978). The Structured Sound Synthesis Project

(SSSP): an Introduction. *Technical Report CSRG-92*, Toronto: University of Toronto.

Byrd, D. (1974). A System for Printing Music by Computer. *Computers and the Humanities* 8: 161-72.

Chamberlain, H. (1976). Experimental Fourier Series Universal Tone Generator. *JAES* 24.4: 271-276.

Chomsky, N. (1972). *Syntactic Structures.* The Hague: Mouton.

Chowning, J. (1971). The Simulation of Moving Sound Sources. *JAES* 19.1: 2-6.

-------- (1973). The Synthesis of Complex Audio Spectra by Means of Frequency Modulation. *JAES* 21.7: 526-34.

Clough, J. (1969). Computer Music and Group Theory. *American Society of University Composers Proceedings* 4: 10-19.

-------- (1971). An Interactive System for Computer Sound Generation. *American Society of University Composers Proceedings* 6: 22-26.

Davis, R. M. (1966). Man-Machine Communication. In Cuadra, C. A., Ed., *Annual Review of Information Science and Technology*, Vol. 1. New York: Interscience Publishers, pp. 221-254.

Fedorkow, G. (1978). *Audio Network Control.* Toronto: M.Sc. Thesis, Dept. Electrical Engineering, University of Toronto.

Fedorkow, G., Buxton, W., & Smith K. C. (1977). A Computer Controlled Sound Distribution System for the Performance of Electroacoustic Music. Toronto, Unpublished Manuscript, CSRG/SSSP.

Franco, S. (1974). *Hardware Design of a Real-Time Musical System*, Urbana: Ph.D. Thesis, Department of Computer Science, University of Illinois.

Freeman, D. (1977). Slewing Distortion in Digital-to-Analogue Conversion. *JAES* 25.4: 178-183.

Friend, D. (1971). A Time-Shared Hybrid Synthesizer. *JAES* 19.11: 928-935.

Gabura, J., & Ciamaga, G. (1968). Computer Control of Sound Apparatus for Electronic Music. *JAES* 16.1: 49-51.

Grey, J. (1975). An Exploration of Musical Timbre. *STAN-M-2*, Stanford: Ph.D. Thesis, Department of Music.

Grogono, P. (1973). Software for an Electronic Music Studio. *Software Practice and Experience* 3: 369-83.

Hansen, F. E. (1977). Sonic Demonstration of the EGG-synthesizer. *Electronic*

Music & Musical Acoustics 3. Arhus, Denmark: Institute of Musical Acoustics, University of Arhus.

Helmholtz, H. (1954). *On the Sensations of Tone.* New York: Dover Publications.

Hiller, L. (1959). Computer Music. *Scientific American* 201.6: 109-120.

Hiller, L. & Isaacson, L. (1958). Music Composition with a High-Speed Digital Computer. *JAES* 6.3: 154-160.

-------- (1959). *Experimental Music.* New York: McGraw-Hill.

Howe, H. S. (1975a). Composing by Computer. *Computers and the Humanities* 9: 281-290.

-------- (1975b). *Electronic Music Synthesis: Concepts, Facilities, Techniques.* New York: W. W. Norton & Co. Inc.

Ingalls, D. H. H. (1978). The Smalltalk-76 Programming System Design and Implementation. *Proceedings of the Principles of Programming Languages Conference,* Tuscon, Arizona.

Iverson, K. (1960). *A Programming Language.* New York: John Wiley & Sons.

Jones, D. (1956). *The Pronunciation of English.* Cambridge: Cambridge University Press.

-------- (1972). *An Outline of English Phonetics.* Cambridge: Cambridge University Press.

Kaegi, W. (1973). A Minimum Description of the Linguistic Sign Repertoire: part 1. *Interface* 2: 141-53.

-------- (1974). A Minimum Description of the Linguistic Sign Repertoire: part 2. *Interface* 3: 137-57.

Kaegi, W. & Tempelaars, S. (1978). VOSIM - A New Sound Synthesis System. *JAES* 26.6: 418-425.

Kaehler, T. (1975). Some Music at XEROX Parc. *Proceedings of the Second Annual Music Computation Conference,* Vol. 1, Software Synthesis Techniques. Urbana: University of Illinois, pp. 53-57.

Kay, A. C. (1977). Microelectronics and the Personal Computer. *Scientific American* 237.3: 230 - 244.

Kernighan, B. & Ritchie, D. (1978). *The C Programming Language.* Englewood Cliffs, N. J.: Prentice-Hall Inc.

Kobrin, E. (1975). *HYBRID IV User's Manual.* La Jolla: unpublished manuscript, Centre for Music Experiment.

Koenig, G. (1970). PROJECT 2. *Electronic Music Reports* 3: 4-161.

Kritz, J. (1975). A 16-Bit A-D-A Conversion System for High-Fidelity Audio Research. *IEEE ASSP* 23.1: 146-149.

Laske, O. (1975). Towards a Theory of Musical Cognition. *Interface* 4: 147-208.

-------- (1977). *Music, Memory, and Thought.* Ann Arbour, Michigan: University Microfilms International.

-------- (1978). SYSTEM STRUCTURE AND HUMAN MEMORY: The Relevance of Understanding User Behaviour in Interactive Musical Task Environments. *SSSP Technical Memo 4.* Toronto: Computer Systems Research Group, University of Toronto.

Le Brun, M. (1977). Waveshaping Synthesis. Unpublished Manuscript, CCRMA, Stanford University.

Licklider, J. C. (1968). Man-Computer Communication. In Cuadra, C. A., Ed. *Annual Review of Information Science and Technology,* Vol. 3. Chicago: Encyclopedia Britannica, pp. 201-240.

Lions, J. (1977). *A Commentary on the UNIX Operating System.* Department of Computer Science, University of New South Wales, Australia.

Lycklama, H. (1978). UNIX on a Microprocessor. *The Bell Systems Technical Journal* 57.6, part 2: 2087-2102.

Lycklama, H. & Christensen, C. (1978). A Minicomputer Satellite Processor System. The Bell Systems Technical Journal 57.6, part 2: 2103-2114.

Manthey, M. (1978). The EGG: A Purely Digital Real Time Polyphonic Sound Synthesizer. *The Computer Music Journal* 2.2: 32-36.

Mathews, M. (1969). *The Technology of Computer Music.* Cambridge: MIT Press.

Mathews, M. & Moore, F. (1970). GROOVE - A Program to Compose, Store, and Edit Functions of Time. *Comm. ACM* 13.12: 715-721.

Martin, J. (1973). *Design of Man-Computer Dialogue.* Englewood Cliffs, N. J.: Prentice Hall.

Martin, T. H. (1973). The User Interface in Interactive Systems. In Cuadra, C. A., Ed., *Annual Review of Information Science and Technology,* Vol 8. Washington, D. C.: American Society for Information Science, pp. 203-219.

Mc Cracken, D. (1955). Monte Carlo Method. *Scientific American* 192.5.

Meadow, C. T. (1970). *Man-Machine Communication.* New York: John Wiley & Sons.

Meister, D. (1976). *Behavioral Foundations of System Development.* New York: John Wiley & Sons.

Meister, D. & Rabideau, G. F. (1965). *Human Factors Evaluations in Systems Development.* New York: John Wiley & Sons.

Melby, C. (1976). *Computer Music Compositions of the United States 1976.* Beverly Hills Va., Theodore Front.

Miller, L. A. & Thomas, J. C. Jr. (1977). Behavioral Issues in the Use of Interactive Systems. *Inernational Journal of Man-Machine Studies,* 9: 237-260.

Miller, R. (1953). A Method for Man-Machine Task Analysis. *Report WADC-TR-53-137.* Wright Air Development Center, Wright-Patterson AFB, Ohio.

Moorer, J. (1972). Music and Computer Composition. *Comm. ACM* 15.2: 104-113.

-------- (1977). Signal Processing Aspects of Computer Music - A Survey. *Proceedings of the IEEE* 65.8: 1108-1137.

Newell, A. (1969). Heuristic Programming: Ill Structured Problems. In Aronofsky, J., Ed. *Progress in Operations Research.* New York: John Wiley & Sons, pp. 371-372.

Newman, W. & Sproull, R. (1973). *Principles of Interactive Computer Graphics.* New York: Mc Graw-Hill.

Parsons, H. (1972). *Man-Machine System Experiments.* Baltimore: The John Hopkins Press.

Pinzarrone, J. (1977). Interactive Woman-Machine Interaction or Live Computer Music Performed by Dance. *Creative Computing* 3.2: 66.

Pulfer, J., K. (1970). Computer Aid for Musical Composers. *Bulletin of Radio and Electrical Engineering Division* 20.2: 44-48.

Rader, G. (1977). The Formal Composition of Music. *Technical Report 77-1.* Ile-Ife, Nigeria: Dept. of Computer Science, University of Ife.

Reeves, W. T. (1976). *A Device-Independent General Purpose Graphics System in a Minicomputer Time-Sharing Environment.* Toronto: M.Sc. Thesis, Department of Computer Science, University of Toronto.

-------- (1978). *GPAC User's Manual* (Second Edition). Toronto: CSRG/DGP, University of Toronto.

Risset, J. (1969). *An Introductory Catalog of Computer Synthesized Sound.* Murray Hill, N.J.: Bell Telephone Laboratories.

Risset, J., & Mathews, M. (1969). Analysis of Musical-Instrument Tones. *Physics Today* 22.2: 23-30.

Rosenboom, D. (1975). A Model for Detection and Analysis of Information Processing Modalities in the Nervous System Through an Adaptive, Interactive, Computerized, Electronic Music Instrument. *Proceedings of the Second Annual Music Computation Conference*, Vol. 4, Information Processing Systems. Urbana: University of Illinois, pp. 54-78.

Saunders, S. (1977). Improved F.M. Audio Synthesis Method for Real Time Digital Music Generation. *Computer Music Journal* 1.1: 53-55.

Schaefer, R., A. (1970). Electronic Musical Tone Production by Nonlinear Waveshaping. *JAES* 8.4: 413-416.

Schaeffer, P. (1966). *Traite des Objets Musicaux*, Paris: Editions du Seuil.

Schottstaedt, B. (1977). The Simulation of Natural Instrument Tones using Frequency Modulation with a Complex Modulating Wave. *Computer Music Journal* 1.4: 46-50.

Sheridan, B. & Ferrell, W. (1974). *Man-Machine Systems: Information, Control, and Decision Models of Human Performance*. Cambridge: MIT Press.

Singleton, W. T., Easterby, R. S. & Whitfield, D. C., Eds. (1971). *The Human Operator in Complex Systems*. London: Taylor & Francis Ltd.

Smith, J. & Kobrin, E. (1977). *Kobrin: Computer in Performance*. Berlin: DAAD.

Smith, L. (1972). SCORE - A Musician's Approach to Computer Music. *JAES* 20.1: 7-14.

Smoliar, S. (1971). A Parallel Processing Model of Musical Structures. *AI TR-242*. Cambridge: Ph.D. Thesis, MIT AI Laboratory.

-------- (1976). Schenker: A Computer Aid for Analysing Tonal Music. *Music Project Report No. 6*. University of Pennsylvania: Department of Computer and Information Science.

Sutherland, I. E. (1965). SKETCHPAD: A Man-Machine Graphical Communication System. *TR 296*. Cambridge: MIT Lincon Laboratory.

Tanner, P. (1972). MUSICOMP, an Experimental Aid for the Composition and Production of Music. *ERB-869*. Ottawa: N.R.C. Radio and Electrical Engineering Division.

Tempelaars, S. (1977). The VOSIM Signal Spectrum. *Interface* 6: 81-96.

-------- (in press). The VOSIM Oscillator. *Proceedings of the First International Conference on Computer Music*. Cambridge: MIT Press.

Thompson, K. (1978). UNIX Implementation. *The Bell Systems Technical Journal* 57.6, part 2: 1931-1946.

Thompson, K. & Ritchie, D. M. (1974). The UNIX Time-Sharing System. *Communications of the ACM* 17.7

Truax, B. (1973). The Computer Composition - Sound Synthesis Programs POD4, POD5, & POD6. *Sonological Reports* 2, Utrecht: Institute of Sonology.

-------- (1976). A Communicational Approach to Computer Sound Programs. *Journal of Music Theory* 20.2: 227-300.

-------- (in press) The Inverse Relationship Between Generality and Strength in Computer Music Programs. *Proceedings of the First International Conference on Computer Music.* Cambridge: M.I.T. Press.

Tucker, W. H., Bates, R. H. T., Frykberg, S. D., Howarth, R. J., Kennedy, W. K., Lamb, M. R. & Vaughan, R. G. (1977). An Interactive Aid for Musicians. *Int. J. Man-Machine Studies* 9: 635-651.

Van Cott, H. P. & Kinkade, R. G., Eds. (1972). *Human Engineering Guide to Equipment Design.* Washington, D. C.: U. S. Government Printing Office.

Vercoe, B. (1971). The MUSIC 360 Language for Sound Synthesis. *American Society of University Composers Proceedings* 6: 16-21.

-------- (1973). *Reference Manual for the MUSIC 360 Language for Digital Sound Synthesis.* Cambridge: unpublished manuscript, Studio for Experimental Music, MIT.

-------- (1975). Man-Computer Interaction in Creative Applications. Cambridge: unpublished manuscript, Studio for Experimental Music, MIT.

Vink, J. & Buxton, W. (1974). *Studio Manual.* Utrecht: Institute of Sonology.

Wiggen, K. (1972). The Electronic Music Studio at Stockholm; its Development and Construction. *INTERFACE* 1: 127-165.

Winnograd, T. (1968). Linguistics and the Computer Analysis of Tonal Harmony. *Journal of Music Theory* 12.1: 2-49.

Xenakis, I. (1971). *Formalized Music.* Bloomington: Indiana University Press.

Zinovieff, P. (1972). *VOCOM: A Synthetical Engine.* London: EMS London Ltd.

11. *APPENDIX 1: Synthesizer Register Summary*

DEL: Non-zero value implies VOSIM mode. DEL is the average delay, or time-out period, between instances of cycle-mode.

DEV: The % of random deviation in the value DEL. Is used to control degree of non-periodicity in VOSIM sounds.

ENV: The current value of the (as yet unscaled) amplitude envelope. ENV is scaled by VOL, and the product is used to scale the waveform sample to being output.

F_INC: Used for frequency control. F_INC is the increment added to the address of the last waveform sample output, in order to derive that of the next sample.

MOD_INDEX: Controls the amount of modulation that the current oscillator "n" effects on oscillator "n+1". When the msb is zero, mode is FM. When the msb is one, mode is waveshaping.

OP_SEL: Used to select to which of the four audio output busses the oscillator's output should be fed.

VOL: The maximum volume to which the envelope (ENV), and consequently waveform, is to be scaled. VOL is logarithmic scale.

WF_SEL: A value to select from which of the 8 waveform buffers the waveform sample is to be selected. Used to change waveforms.

UNIVERSITY OF TORONTO

COMPUTER SYSTEMS RESEARCH GROUP

BIBLIOGRAPHY OF CSRG TECHNICAL REPORTS+

* CSRG-1 EMPIRICAL COMPARISON OF LR(k) AND PRECEDENCE PARSERS
J.J. Horning and W.R. Lalonde, September 1970
[ACM SIGPLAN Notices, November 1970]

* CSRG-2 AN EFFICIENT LALR PARSER GENERATOR
W.R. Lalonde, February 1971 [M.A.Sc. Thesis, EE 1971]

* CSRG-3 A PROCESSOR GENERATOR SYSTEM
J.D. Gorrie, February 1971 [M.A.Sc. Thesis, EE 1971]

* CSRG-4 DYLAN USER'S MANUAL
P.E. Bonzon, March 1971

CSRG-5 DIAL - A PROGRAMMING SYSTEM FOR INTERACTIVE ALGEBRAIC MANIPULATION
Alan C.M. Brown and J.J. Horning, March 1971

CSRG-6 ON DEADLOCK IN COMPUTER SYSTEMS
Richard C. Holt, April 1971
[Ph.D. Thesis, Dept. of Computer Science, Cornell University, 1971]

CSRG-7 THE STAR-RING SYSTEM OF LOOSELY COUPLED DIGITAL DEVICES
John Neill Thomas Potvin, August 1971
[M.A.Sc. Thesis, EE 1971]

* CSRG-8 FILE ORGANIZATION AND STRUCTURE
G.M. Stacey, August 1971

CSRG-9 DESIGN STUDY FOR A TWO-DIMENSIONAL COMPUTER-ASSISTED ANIMATION SYSTEM
Kenneth B. Evans, January 1972 [M.Sc. Thesis, DCS, 1972]

* CSRG-10 HOW A PROGRAMMING LANGUAGE IS USED
William Gregg Alexander, February 1972
[M.Sc. Thesis, DCS 1971; Computer, v.8, n.11, November 1975]

* CSRG-11 PROJECT SUE STATUS REPORT
J.W. Atwood (ed.), April 1972

* CSRG-12 THREE DIMENSIONAL DATA DISPLAY WITH HIDDEN LINE REMOVAL
Rupert Bramall, April 1972 [M.Sc. Thesis, DCS, 1971]

* CSRG-13 A SYNTAX DIRECTED ERROR RECOVERY METHOD
Lewis R. James, May 1972 [M.Sc. Thesis, DCS, 1972]

+ Abbreviations:
DCS - Department of Computer Science, University of Toronto
EE - Department of Electrical Engineering, University of Toronto

* - Out of print

CSRG-14 THE USE OF SERVICE TIME DISTRIBUTIONS IN SCHEDULING
Kenneth C. Sevcik, May 1972
[Ph.D. Thesis, Committee on Information Sciences, University of Chicago, 1971; JACM, January 1974]

CSRG-15 PROCESS STRUCTURING
J.J. Horning and B. Randell, June 1972
[ACM Computing Surveys, March 1973]

CSRG-16 OPTIMAL PROCESSOR SCHEDULING WHEN SERVICE TIMES ARE HYPEREXPONENTIALLY DISTRIBUTED AND PREEMTION OVERHEAD IS NOT NEGLIGIBLE
Kenneth C. Sevcik, June 1972
[Proceedings of the Symposium on Computer-Communication, Networks and Teletraffic, Polytechnic Institute of Brooklyn, 1972]

* CSRG-17 PROGRAMMING LANGUAGE TRANSLATION TECHNIQUES
W.M. McKeeman, July 1972

CSRG-18 A COMPARATIVE ANALYSIS OF SEVERAL DISK SCHEDULING ALGORITHMS
C.J.M. Turnbull, September 1972

CSRG-19 PROJECT SUE AS A LEARNING EXPERIENCE
K.C. Sevcik _et al_, September 1972
[Proceedings AFIPS Fall Joint Computer Conference, v. 41, December 1972]

* CSRG-20 A STUDY OF LANGUAGE DIRECTED COMPUTER DESIGN
David B. Wortman, December 1972
[Ph.D. Thesis, Computer Science Department, Stanford University, 1972]

CSRG-21 AN APL TERMINAL APPROACH TO COMPUTER MAPPING
R. Kvaternik, December 1972 [M.Sc. Thesis, DCS, 1972]

* CSRG-22 AN IMPLEMENTATION LANGUAGE FOR MINICOMPUTERS
G.G. Kalmar, January 1973 [M.Sc. Thesis, DCS, 1972]

CSRG-23 COMPILER STRUCTURE
W.M. McKeeman, January 1973
[Proceedings of the USA-Japan Computer Conference, 1972]

* CSRG-24 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING
J.D. Gannon (ed.), March 1973

CSRG-25 THE INVESTIGATION OF SERVICE TIME DISTRIBUTIONS
Eleanor A. Lester, April 1973 [M.Sc. Thesis, DCS, 1973]

* CSRG-26 PSYCHOLOGICAL COMPLEXITY OF COMPUTER PROGRAMS: AN INITIAL EXPERIMENT
Larry Weissman, August 1973

* CSRG-27 STRUCTURED SUBSETS OF THE PL/I LANGUAGE
Richard C. Holt and David B. Wortman, October 1973

* CSRG-28 ON THE REDUCED MATRIX REPRESENTATION OF LR(k) PARSER TABLES
  Marc Louis Joliat, October 1973 [Ph.D. Thesis, EE 1973]

* CSRG-29 A STUDENT PROJECT FOR AN OPERATING SYSTEMS COURSE
  B. Czarnik and D. Tsichritzis (eds.), November 1973

* CSRG-30 A PSEUDO-MACHINE FOR CODE GENERATION
  Henry John Pasko, December 1973 [M.Sc. Thesis, DCS 1973]

* CSRG-31 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING
  J.D. Gannon (ed.), Second Edition, March 1974

* CSRG-32 SCHEDULING MULTIPLE RESOURCE COMPUTER SYSTEMS
  E.D. Lazowska, May 1974 [M.Sc. Thesis, DCS, 1974]

* CSRG-33 AN EDUCATIONAL DATA BASE MANAGEMENT SYSTEM
  F. Lochovsky and D. Tsichritzis, May 1974 [INFOR, to appear]

* CSRG-34 ALLOCATING STORAGE IN HIERARCHICAL DATA BASES
  P. Bernstein and D. Tsichritzis, May 1974 [Information Systems Journal, v.1, pp.133-140]

* CSRG-35 ON IMPLEMENTATION OF RELATIONS
  D. Tsichritzis, May 1974

* CSRG-36 SIX PL/I COMPILERS
  D.B. Wortman, P.J. Khaiat, and D.M. Lasker, August 1974 [Software Practice and Experience, v.6, n.3, July-Sept. 1976]

* CSRG-37 A METHODOLOGY FOR STUDYING THE PSYCHOLOGICAL COMPLEXITY OF COMPUTER PROGRAMS
  Laurence M. Weissman, August 1974 [Ph.D. Thesis, DCS, 1974]

* CSRG-38 AN INVESTIGATION OF A NEW METHOD OF CONSTRUCTING SOFTWARE
  David M. Lasker, September 1974 [M.Sc. Thesis, DCS, 1974]

CSRG-39 AN ALGEBRAIC MODEL FOR STRING PATTERNS
  Glenn F. Stewart, September 1974 [M.Sc. Thesis, DCS, 1974]

* CSRG-40 EDUCATIONAL DATA BASE SYSTEM USER'S MANUAL
  J. Klebanoff, F. Lochovsky, A. Rozitis, and D. Tsichritzis, September 1974

* CSRG-41 NOTES FROM A WORKSHOP ON THE ATTAINMENT OF RELIABLE SOFTWARE
  David B. Wortman (ed.), September 1974

* CSRG-42 THE PROJECT SUE SYSTEM LANGUAGE REFERENCE MANUAL
  B.L. Clark and F.J.B. Ham, September 1974

* CSRG-43 A DATA BASE PROCESSOR
  E.A. Ozkarahan, S.A. Schuster and K.C. Smith, November 1974 [Proceedings National Computer Conference 1975, v.44, pp.379-388]

* CSRG-44 MATCHING PROGRAM AND DATA REPRESENTATION TO A COMPUTING ENVIRONMENT
  Eric C.R. Hehner, November 1974 [Ph.D. Thesis, DCS, 1974]

* CSRG-45 THREE APPROACHES TO RELIABLE SOFTWARE; LANGUAGE DESIGN, DYADIC SPECIFICATION, COMPLEMENTARY SEMANTICS
  J.E. Donahue, J.D. Gannon, J.V. Guttag and J.J. Horning, December 1974

CSRG-46 THE SYNTHESIS OF OPTIMAL DECISION TREES FROM DECISION TABLES
  Helmut Schumacher, December 1974 [M.Sc. Thesis, DCS, 1974]

* CSRG-47 LANGUAGE DESIGN TO ENHANCE PROGRAMMING RELIABILITY
  John D. Gannon, January 1975 [Ph.D. Thesis, DCS, 1975]

* CSRG-48 DETERMINISTIC LEFT TO RIGHT PARSING
  Christopher J.M. Turnbull, January 1975 [Ph.D. Thesis, EE, 1974]

* CSRG-49 A NETWORK FRAMEWORK FOR RELATIONAL IMPLEMENTATION
  D. Tsichritzis, February 1975 [in Data Base Description, Dongue and Nijssen (eds.), North Holland Publishing Co.]

* CSRG-50 A UNIFIED APPROACH TO FUNCTIONAL DEPENDENCIES AND RELATIONS
  P.A. Bernstein, J.R. Swenson and D.C. Tsichritzis February 1975 [Proceedings of the ACM SIGMOD Conference, 1975]

* CSRG-51 ZETA: A PROTOTYPE RELATIONAL DATA BASE MANAGEMENT SYSTEM
  M. Brodie (ed). February 1975 [Proceedings Pacific ACM Conference, 1975]

CSRG-52 AUTOMATIC GENERATION OF SYNTAX-REPAIRING AND PARAGRAPHING PARSERS
  David T. Barnard, March 1975 [M.Sc. Thesis, DCS, 1975]

* CSRG-53 QUERY EXECUTION AND INDEX SELECTION FOR RELATIONAL DATA BASES
  J.H. Gilles Farley and Stewart A. Schuster, March 1975

CSRG-54 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING
  J.V. Guttag (ed.), Third Edition, April 1975

CSRG-55 STRUCTURED SUBSETS OF THE PL/1 LANGUAGE
  Richard C. Holt and David B. Wortman, May 1975

* CSRG-56 FEATURES OF A CONCEPTUAL SCHEMA
D. Tsichritzis, June 1975 [Proceedings Very Large Data Base Conference, 1975]

* CSRG-57 MERLIN: TOWARDS AN IDEAL PROGRAMMING LANGUAGE
Eric C.R. Hehner, July 1975

CSRG-58 ON THE SEMANTICS OF THE RELATIONAL DATA MODEL
Hans Albrecht Schmid and J. Richard Swenson, July 1975 [Proceedings of the ACM SIGMOD Conference, 1975]

* CSRG-59 THE SPECIFICATION AND APPLICATION TO PROGRAMMING OF ABSTRACT DATA TYPES
John V. Guttag, September 1975 [Ph.D. Thesis, DCS, 1975]

* CSRG-60 NORMALIZATION AND FUNCTIONAL DEPENDENCIES IN THE RELATIONAL DATA BASE MODEL
Phillip Alan Bernstein, October 1975 [Ph.D. Thesis, DCS, 1975]

* CSRG-61 LSL: A LINK AND SELECTION LANGUAGE
D. Tsichritzis, November 1975 [Proceedings ACM SIGMOD Conference, 1976]

* CSRG-62 COMPLEMENTARY DEFINITIONS OF PROGRAMMING LANGUAGE SEMANTICS
James E. Donahue, November 1975 [Ph.D. Thesis, DCS, 1975]

CSRG-63 AN EXPERIMENTAL EVALUATION OF CHESS PLAYING HEURISTICS
Lazlo Sugar, December 1975 [M.Sc. Thesis, DCS, 1975]

CSRG-64 A VIRTUAL MEMORY SYSTEM FOR A RELATIONAL ASSOCIATIVE PROCESSOR
S.A. Schuster, E.A. Ozkarahan, and K.C. Smith, February 1976 [Proceedings National Computer Conference 1976, v.45, pp. 855-862]

CSRG-65 PERFORMANCE EVALUATION OF A RELATIONAL ASSOCIATIVE PROCESSOR
E.A. Ozkarahan, S.A. Schuster, and K.C. Sevcik, February 1976 [ACM Transactions on Database Systems, v.1, n:4, December 1976]

CSRG-66 EDITING COMPUTER ANIMATED FILM
Michael D. Tilson, February 1976 [M.Sc. Thesis, DCS, 1975]

CSRG-67 A DIAGRAMMATIC APPROACH TO PROGRAMMING LANGUAGE SEMANTICS
James R. Cordy, March 1976 [M.Sc. Thesis, DCS, 1976]

* CSRG-68 A SYNTHETIC ENGLISH QUERY LANGUAGE FOR A RELATIONAL ASSOCIATIVE PROCESSOR
L. Kerschberg, E.A. Ozkarahan, and J.E.S. Pacheco April 1976

CSRG-69 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING
D. Barnard and D. Thompson (Eds.), Fourth Edition, May 1976

* CSRG-70 A TAXONOMY OF DATA MODELS
L. Kerschberg, A. Klug, and D. Tsichritzis, May 1976
[Proceedings Very Large Data Base Conference, 1976]

* CSRG-71 OPTIMIZATION FEATURES FOR THE ARCHITECTURE OF A
DATA BASE MACHINE
E.A. Ozkarahan and K.C. Sevcik, May 1976

* CSRG-72 THE RELATIONAL DATA BASE SYSTEM OMEGA - PROGRESS REPORT
H.A. Schmid (ed.), P.A. Bernstein (ed.), B. Arlow,
R. Baker and S. Pozgaj, July 1976

CSRG-73 AN ALGORITHMIC APPROACH TO NORMALIZATION OF
RELATIONAL DATA BASE SCHEMAS
P.A. Bernstein and C. Beeri, September 1976

* CSRG-74 A HIGH-LEVEL MACHINE-ORIENTED ASSEMBLER LANGUAGE
FOR A DATA BASE MACHINE
E.A. Ozkarahan and S.A. Schuster, October 1976

CSRG-75 DO CONSIDERED OD: A CONTRIBUTION TO THE
PROGRAMMING CALCULUS
Eric C.R. Hehner, November 1976

CSRG-76 "SOFTWARE HUT": A COMPUTER PROGRAM ENGINEERING
PROJECT IN THE FORM OF A GAME
J.J. Horning and D.B. Wortman, November 1976

CSRG-77 A SHORT STUDY OF PROGRAM AND MEMORY POLICY BEHAVIOUR
G. Scott Graham, January 1977

CSRG-78 A PANACHE OF DBMS IDEAS
D. Tsichritzis, February 1977

CSRG-79 THE DESIGN AND IMPLEMENTATION OF AN ADVANCED LALR
PARSE TABLE CONSTRUCTOR
David H. Thompson, April 1977 [M.Sc. Thesis, DCS, 1976]

CSRG-80 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING
D. Barnard (Ed.), Fifth Edition, May 1977

CSRG-81 PROGRAMMING METHODOLOGY: AN ANNOTATED BIBLIOGRAPHY FOR
IFIP WORKING GROUP 2.3
Sol J. Greenspan and J.J. Horning (Eds.), First Edition,
May 1977

CSRG-82 NOTES ON EUCLID
edited by W. David Elliot and David T. Barnard,
August 1977

CSRG-83 TOPICS IN QUEUEING NETWORK MODELING
edited by G. Scott Graham, July 1977

CSRG-84 TOWARD PROGRAM ILLUSTRATION
Edward Yarwood, September 1977 [M.Sc. Thesis, DCS, 1974]

CSRG-85 CHARACTERIZING SERVICE TIME AND RESPONSE TIME DISTRIBUTIONS IN QUEUEING NETWORK MODELS OF COMPUTER SYSTEMS
Edward D. Lazowska, September 1977
[Ph.D. Thesis, DCS, 1977]

CSRG-86 MEASUREMENTS OF COMPUTER SYSTEMS FOR QUEUEING NETWORK MODELS
Martin G. Kienzle, October 1977
[M.Sc. Thesis, DCS, 1977]

CSRG-87 'OLGA' LANGUAGE REFERENCE MANUAL
B. Abourbih, H. Trickey, D.M. Lewis, E.S. Lee, P.I.P. Boulton, November 1977

CSRG-88 USING A GRAMMATICAL FORMALISM AS A PROGRAMMING LANGUAGE
Brad A. Silverberg, January 1978
[M.Sc. Thesis, DCS, 1978]

CSRG-89 ON THE IMPLEMENTATION OF RELATIONS: A KEY TO EFFICIENCY
Joachim W. Schmidt, January 1978

CSRG-90 DATA BASE MANAGEMENT SYSTEM USER PERFORMANCE
Frederick H. Lochovsky, April 1978
[Ph.D. Thesis, DCS, 1978]

CSRG-91 SPECIFICATION AND VERIFICATION OF DATA BASE SEMANTIC INTEGRITY
Michael Lawrence Brodie, April 1978
[Ph.D. Thesis, DCS, 1978]

CSRG-92 "STRUCTURED SOUND SYNTHESIS PROJECT (SSSP): AN INTRODUCTION"
by William Buxton, Guy Ferdorkow, with Ronald Baecker, Gustav Ciamaga, Leslie Mezei and K.C. Smith, June 1978

CSRG-93 "A DEVICE-INDEPENDENT,GENERAL-PURPOSE GRAPHICS SYSTEM IN A MINICOMPUTER TIME-SHARING ENVIRONMENT"
William T. Reeves, August 1978
[M.Sc. Thesis, DCS, 1976]

CSRG-94 "ON THE AXIOMATIC VERIFICATION OF CONCURRENT ALGORITHMS "
Christian Lengauer, August 1978
[M.Sc. Thesis, DCS, 1978]

CSRG-95 PISA: "A PROGRAMMING SYSTEM FOR INTERACTIVE PRODUCTION OF APPLICATION SOFTWARE"
Rudolf Marty, August 1978

CSRG-96 "ADAPTIVE MICROPROGRAMMING AND PROCESSOR MODELING"
Walter G. Rosocha
[Ph.D.Thesis, EE, August 1978]

CSRG-97 DESIGN ISSUES IN THE FOUNDATION OF A COMPUTER-BASED TOOL FOR MUSIC COMPOSITION
William Buxton
[M.Sc. Thesis, CSRG, October 1978]