

pedelaar

Music Software User's Manual

by

William Buxton

*Structured Sound Synthesis Project
Computer Systems Research Group
University of Toronto
Toronto, Ontario
Canada
M5S 1A1*

C.S.R.G. Technical Note 22

June 24, 1981

ACKNOWLEDGEMENTS

This manual has benefited from the input of virtually every person that has used the SSSP system. If it is at all understandable, it is due to this input. The system itself owes a great deal to Bill Reeves, Guy Fedorkow, Sanand Patel, Chris Retterath, Rob Pike, K. C. Smith, Ron Baecker and Leslie Mezei. In preparing this documentation, Steve Hull, Martin Lamb, Leslie Gondor, Paul Ziolo, Gayle Young, Susan Frykberg, Bob Pritchard, James Montgomery, Wesley Lowe, Danny Shoskes, and William Matthews have been of particular assistance.

Finally, we would like to acknowledge the continued support of the Social Sciences and Humanities Research Council of Canada which makes the work of the SSSP possible.

1)

1+
:
a

ie

s

y

1



NAME

tar - tape archiver

SYNOPSIS

tar [key]. [name ...]

DESCRIPTION

Tar saves and restores files on magtape. Its actions are controlled by the key argument. The key is a string of characters containing at most one function letter and possibly one or more function modifiers. Other arguments to the command are file or directory names specifying which files are to be dumped or restored. In all cases, appearance of directory name refers to the files and (recursively) sub-directories of that directory.

The function portion of the key is specified by one of the following letters:

- r The named files are written on the end of the tape. The c function implies this.
- x The named files are extracted from the tape. If a named file matches a directory whose contents had been written onto the tape, this directory is (recursively) extracted. The owner, modification time, and mode are restored (if possible). If no file argument is given, the entire content of the tape is extracted. Note that if multiple entries specifying the same file are on the tape, the last one overwrites all earlier.
- t The names of the specified files are listed each time they occur on the tape. If no file argument is given, all of the names on the tape are listed.
- u The named files are added to the tape if either they are not already there or have been modified since last put on the tape.
- c Create a new tape; writing begins on the beginning of the tape instead of after the last file. This command implies r.

The following characters may be used in addition to the letter which selects the function desired.

- 0, ..., 7 This modifier selects the drive on which the tape is mounted. The default is 1.
- v Normally tar does its work silently. The v (verbose) option causes it to type the name of each

CONTENTS

1. INTRODUCTION	1
2. TECHNICAL DETAILS	4
2.1 Introduction	4
2.2 Modes of Interaction	5
2.2.1 General 5	
2.3 Graphics	6
2.4 Typing	7
2.4.1 General 7	
2.4.2 Special Characters 7	
2.4.3 Naming Conventions 7	
2.4.4 Miscellaneous 8	
2.5 Getting Started	8
2.6 Documentation Conventions	8
3. MUSIC COMMANDS	10
3.1 Objects	10
3.1.1 General 10	
3.1.2 Definition 10:	
3.1.2.1 bank 10	
3.1.2.2 objed 10	
3.1.3 Auxiliary Commands 10:	
3.1.3.1 pobj 10	
3.2 Scores	11
3.2.1 General 11	
3.2.2 Direct Specification 11	
3.2.2.1 record 11	
3.2.2.2 sced 12	
3.2.2.3 scriva 13	
3.2.3 Compositional Routines 13	
3.2.3.1 makescore 13	
3.2.3.2 prod 14	
3.2.4 Auxiliary Commands 14:	
3.2.4.1 cksize 15	
3.2.4.2 dur 15	
3.2.4.3 orch 16	
3.2.4.4 prune 16	
3.2.4.5 invert 16	
3.2.4.6 iscale 16	
3.2.4.7 makeff 17	
3.2.4.8 map 17	
3.2.4.9 mix 18	
3.2.4.10 mode 18	
3.2.4.11 pscore 19	
3.2.4.12 rand 20	
3.2.4.13 relpos 22	
3.2.4.14 retro 22	
3.2.4.15 rotate 22	
3.2.4.16 scorch 23	
3.2.4.17 sdelay 23	
3.2.4.18 setchan 24	
3.2.4.19 setvol 24	
3.2.4.20 splice 26	
3.2.4.21 transp 26	

3.2.4.22 tscalr	27	
3.3 Performance		28
3.3.1 conduct	28	
3.3.2 lsiply	28	
3.3.3 play	28	
3.4 Functions		30
3.4.1 Definition	30	
3.4.1.1 funced	30	
3.4.2 Auxiliary Commands	30	
3.4.2.1 finv	30	
3.4.2.2 fretro	31	
3.4.2.3 pfunc	31	
3.5 Waveforms		31
3.5.1 Definition	31	
3.5.1.1 fconvr	31	
3.5.1.2 wavedr	31	
3.5.1.3 wavemix	32	
3.5.1.4 wavesum	33	
3.5.2 Auxiliary Commands	35	
3.5.2.1 pwave	35	
3.5.2.2 wfist	35	
3.5.2.3 wfload	35	
3.5.2.4 wfsave	36	
3.6 Miscellaneous Music Commands		36
3.6.0.1 cleanup	36	
3.6.1 pitch	36	
3.6.2 swit	37	
3.6.3 radio	37	
4. UNIX COMMANDS		37
4.1 General		37
4.2 Commands		38
4.2.1 cp	38	
4.2.2 file	38	
4.2.3 gwsnap	38	
4.2.4 ls	38	
4.2.5 mail	38	
4.2.6 mv	39	
4.2.7 rm	39	
4.2.8 who	39	
4.2.9 write	39	
5. REFERENCES		40

CONTENTS

APPENDIX A - A Tutorial on Editing Objects	41
1. INTRODUCTION	41
2. ON ENTERING OBJED	41
3. AUDITIONING AN OBJECT	42
4. WAVEFORM SELECTION	43
5. CREATING NEW WAVEFORMS BY SPECTRUM	45
6. FUNCTION DEFINITION	47
7. NAMING, SAVING AND RETRIEVING OBJECTS	49
8. FM OBJECTS	50
9. MORE ON AUDITIONING MODES	51
10. COMPARING OBJECTS	52
11. VOSIM AND WAVESHAPING	53
12. ADDITIVE SYNTHESIS AND BANK	54
13. REFERENCES	58
APPENDIX B - A Tutorial Introduction to SCRIVA	60
1. INTRODUCTION	60
2. GENERAL	60
3. ADDING NOTES	63
3.1 General	63
3.2 Ludwig	64
3.3 Roll	66
4. WHY DID IT SOUND LIKE THAT?	67
5. MIX AND SPLICE	68
6. DELETING MATERIAL	68
7. AN ASIDE: THE CONCEPT OF SCOPE	69
8. SAVING MATERIAL	69
9. ADDING BY READING	70
10. ORCHESTRATION	71
11. SETTING VOLUME	71
12. NAVIGATION	71
13. TEMPORARY ESCAPE	72
14. SUMMARY OF LIGHT BUTTON FUNCTIONS	73
14.1 Column 1: EDITOR	73
14.1.1 General	73
14.1.2 Notation	73
14.1.3 Display	73
14.1.4 Input	73
14.1.5 Join	74
14.1.6 Score	74

14.1.7 Key	75	
14.1.8 MM	75	
14.1.9 Page	75	
14.2 Column 2: MUSIC		75
14.2.1 General	75	
14.2.2 Object	75	
14.2.3 Volume	76	
14.2.4 Channel	76	
14.3 Column 3: SCOPE		76
14.3.1 General	76	
14.3.2 Whole Score	76	
14.3.3 Circle	77	
14.3.4 Collect	77	
14.3.5 Clear	77	
14.4 Columns 4 & 5: OPERATORS		77
14.4.1 General	77	
14.4.2 Add	77	
14.4.3 Delete	77	
14.4.4 Play	77	
14.4.5 Save	78	
14.4.6 Orchestrate	78	
14.4.7 Scorechestrato	78	
14.4.8 Set Volume	78	
14.4.9 QUIT	78	
APPENDIX C - A Tutorial Introduction to SCED		80
1. INTRODUCTION		80
2. DISCLAIMER		80
3. GETTING STARTED		80
4. SPECIFYING NOTES - the Append Command 'a'		80
5. LISTENING TO A SCORE - the Listen Command 'l'		82
6. ERROR MESSAGES		82
7. LEAVING sced - the Quit Command 'q'		83
8. PRINTING THE BUFFER CONTENTS - the Print Command 'p'		83
9. PLAYING THE SCORE - More on the 'l' Command		85
10. SCOPE - Consolidating a Concept		85
11. THE CURRENT NOTE - 'Dot' or '.'		85
12. MORE ON APPEND - Details on Notes		89
12.1 Pitch/Frequency		90
12.2 Duration		90
12.3 Object		90
12.4 Volume		91
12.5 Delay		91
12.6 Channel		92
12.7 Rests		92
13. SAVING SCORE FILES - The Write Command 'w'		93
14. READING SCORES FROM A FILE - the Edit Command 'e'		94
15. MORE ON NAMES - the File Command 'f'		95

16. MIX, SPLICE AND THE JOIN MODE	95
17. DELETING NOTES - the Delete Command 'd'	97
18. ON-LINE HELP - the Help Command 'h'	98
19. READING SCORES FROM A FILE - the Read Command 'r'	98
20. CONTROLLING TEMPO - the Metronome Command 'mm'	100
21. MODIFYING ATTRIBUTES OF PREVIOUSLY SPECIFIED EVENTS	100
21.1 The Change Command 'c'	100
21.2 The set Commands	101
21.2.1 setfreq 102	
21.2.2 setdur 103	
21.2.3 setobj 105	
21.2.4 setvol 105	
21.2.5 setdel 106	
21.2.6 setchan 106	
21.2.7 settime 107	
22. MORE ON SCOPE: Conditionals	107
23. SEARCHING	110
24. ESCAPING TO THE SHELL - the '!' Command	111
25. MOVING NOTES AROUND - the Move Command 'm'	112
26. COPYING SCORE MATERIAL - the Copy Command 't'	112
27. SCORCHESTRATION - the 'scorch' Command	113
28. REHEARSAL MARKINGS	115
29. MACROS	116
30. SUMMARY OF COMMANDS	117
-00 .)B -00 .LC 00	

CONTENTS

APPENDIX D - A Tutorial Introduction to PROD	119
1. INTRODUCTION	119
2. THE USE OF PROD IN COMPOSITION	119
3. PRODUCTIONS	119
4. SCORE GENERATION AND PROD USAGE	120
5. NOTES	121
6. A NOTE ON TYPING	123
7. NON-TERMINALS	123
8. PARAMETER PASSING	124
9. THE SCORE TERMINAL	126
10. NON-DETERMINISTIC GRAMMARS	126
11. WEIGHTED PROBABILITIES	127
12. RANGES	128
13. RECREATING RANDOM SCORES	129
14. RECURSIVE PRODUCTIONS	130
APPENDIX E - An Introduction to CONDUCT	132
1. THE NATURE OF A CONDUCTABLE SCORE	132
2. CONDUCTABLE PARAMETERS	132
2.1 Octave:	132
2.2 Tempo	132
2.3 Articulation	132
2.4 Amplitude	133
2.5 Richness	133
2.6 Cycle	133
2.7 On/Off	133
3. TECHNIQUES OF CONTROL	133
3.1 General	133
3.2 Direct Control	134
3.2.1 Switches 134	
3.2.2 Continously Variable Parameters 134	
3.2.2.1 Typing 134	
3.2.2.2 The 'Last-Typed' Technique 134	
3.2.2.3 Default Set 134	
3.2.2.4 Dragging 134	
3.3 Indirect Control	135
3.3.1 Triggers 135	
3.3.1.1 Manual Triggers 135	
3.3.1.2 End of Score Triggers 136	
3.3.2 Groupings of Continously Variable Parameters 137	
3.3.2.1 Groups 137	
3.3.2.2 Group Control Transducers 138	
3.3.2.3 Negative Groups 139	
3.4 Additional Performance Variables	139
3.4.1 Score Selection 139	

3.4.2 The Rate Control	140
3.5 Concluding Comments on the Control Structure	141
APPENDIX F - User Defined Shell Commands	142
APPENDIX G - Synthesizer Specifications	145
APPENDIX H - SSSP PUBLICATIONS . MONOGRAPHS	146

Music Software User's Manual

William Buxton

*Structured Sound Synthesis Project
Computer Systems Research Group
University of Toronto
Toronto, Ontario
Canada
M5S 1A1*

A user manual for the music system developed by the SSSP is presented. The various programs available are described, as are the various techniques and conventions required in order to work creatively with the computer. Many of the programs available are graphics based, whose step-by-step use is rather straightforward. Basic documentation of all programs is given in the main body of the manual. Many of the larger program packages are documented in detail in the appendices to this manual.

1. INTRODUCTION

This document is intended to serve as a handbook for musicians using the computer-music system developed by the Structured Sound Synthesis Project of the University of Toronto. As such, its purpose is to document the entire system, as opposed to serving as a manual for an individual program. The scope is broad therefore, and includes a discussion of the various programs of the system, how they relate to each other, as well as other pertinent issues. For those users who just want to find out how a particular program works, this breadth will be more than they want or need. For them, the combination of this Introduction and the Table of Contents will direct them to the part of this manual relevant to their needs. In this regard, particular attention should be paid to the appendices of this document, where tutorials for some of the major programs can be found. For those who want to gain a broader understanding of the SSSP system, we hope that the organization of this manual provides a suitable guide.

The SSSP system has been designed so as to provide a set of powerful tools to composers. The facility is not, however, a production studio for computer music. It exists in a research and development environment, which means that it is not dedicated to music-making activities. However, musicians are welcome, and serious work can be undertaken. In the lab where the SSSP system is housed, one of the main research interests is investigating ways of making the benefits of technology more accessible to people who are unfamiliar with computers. The music system is, in this regard, a case study.

The SSSP system has been designed so as to provide tools to facilitate the undertaking of three main musical tasks:

1. Defining and editing a palette of timbres to be used in a composition. This we call *object definition*.
2. Defining musical structures, or *scores*, including the process of *orchestrating* the

CHAPTER 1

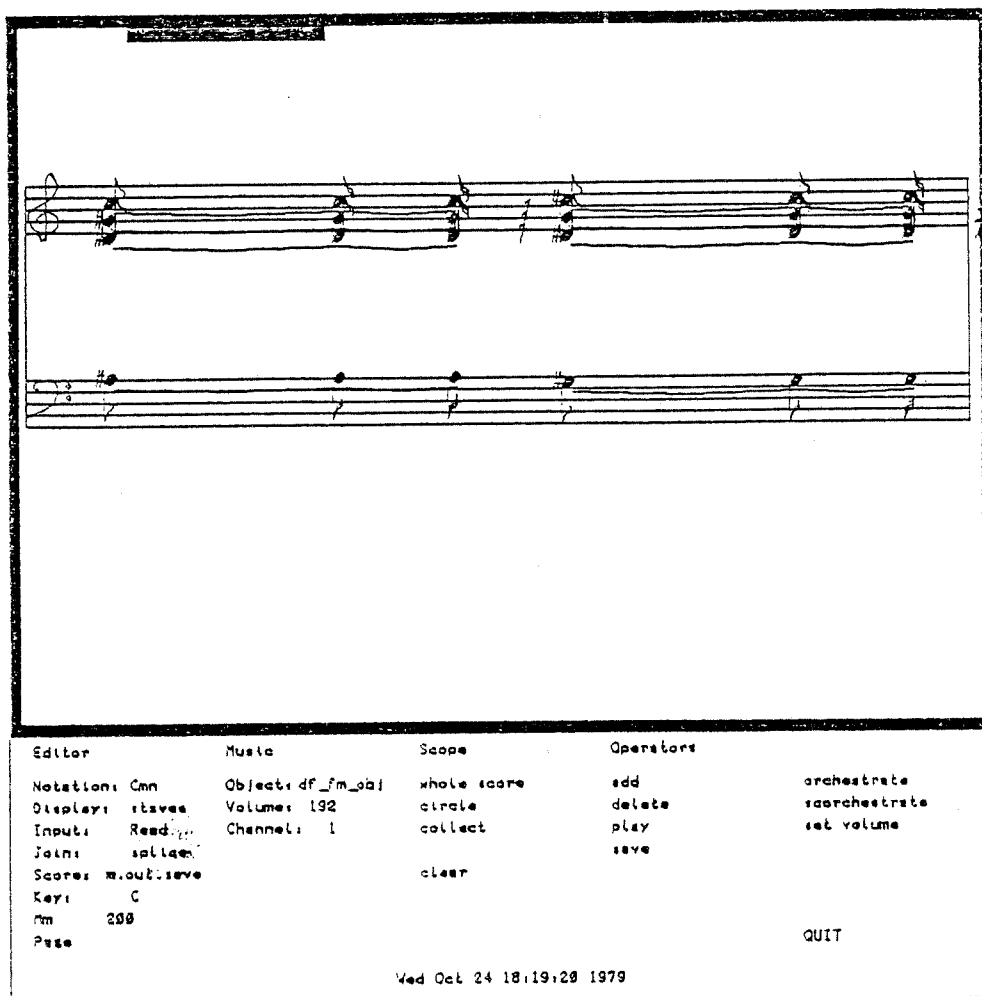
notes of the scores using the timbres defined during object definition.

3. Performing the composition, which may include *conducting* or interpreting the scores which have been composed.

Every effort has been made to free the musician from technical worries, thereby enabling concentration on the more important musical problems.

In thinking about the three activities outlined above, it is important for the musician to realize that there is no need to make use of all of the available tools, or of any of them in any particular order. One can concentrate on those programs which are of interest or importance, and ignore the others until needed. Scores can be composed and auditioned without ever explicitly specifying the timbres with which the notes are to be played. Similarly, timbres can be invented and auditioned independently from any score material.

The system is quite modular, and each module more-or-less stands on its own. In addition, there is often more than one way of doing something, so the musician can use the method that seems most appropriate. This is seen in score notation, for example, where the composer has the choice of working in common music notation (CMN) or various alternative forms of graphic notation. This is illustrated in Figures 1 and 2.



Editor	Musica	Scope	Operators	
Notation: Cmn	Object: df_fm_pbl	whole score	add	orchestrate
Display: staves	Volume: 192	circle	delete	saorchestrate
Input: Read...	Channel: 1	collect	play	set volume
Join: split			save	
Scores: m.out:save		clear		
Key: C				
fm 200				
Page				QUIT

Wed Oct 24 18:19:28 1979

Figure 1. Example of Common Music Notation

Editor	Music	Scope	Operators	
Notation: Roll	Object: df_fm_obj	whole score	add	orchestrate
Display: staves	Volume: 192	circle	delete	scorechestrate
Input: Read	Channel: 1	collect	play	set volume
Join: splice			save	
Scores: m.out.save		clear		
Key: C				
fm: 200				
Page:				QUIT

Wed Oct 24 18:18:04 1979

Figure 2. Example of "Piano-Roll" Notation

The rest of this document is intended to provide the user with a guide to the resources available. Following this Introduction, Chapter Two presents some unavoidable non-musical information relating to working with the system and using this document. A description of the music commands is then presented in Chapter Three. These commands are grouped according to the tasks outlined above: working with *objects*, *scores*, and *performance*. In addition, there are subsections on working with *waveforms*, *functions*, and other miscellaneous music commands. Chapter Four presents a brief description of certain UNIX commands (i.e. commands not specifically for the music system, but useful nevertheless). Finally, tutorials for major programs and other supplementary information are presented in the appendices. This includes the technical specifications of the synthesizer (Appendix G) and information on how users may create their own commands by making aggregates of already existing ones (Appendix F). Throughout, where more information is available, the source is indicated.

Those readers wishing to obtain more information about the SSSP system are referred to the documents listed in the Reference section of this manual, and to the list of SSSP publications and monographs found in Appendix H. In particular, those interested in the system at the programming level are referred to Buxton, Reeves, Patel, & O'Dell (1979); those interested in a general introduction to computer music are referred to Buxton (1977).

CHAPTER 1

At this point, the beginner not familiar with the system is liable to become overwhelmed by the bulk of new information. This can, however, be avoided by trying to relate everything to the three tasks outlined above. In addition, this document should *not* be read sequentially the first time through. Rather, the beginner is encouraged to start by looking only at the following key programs: *objed* (object definition), *scriva* (score definition), and *play* (performance). These are documented in Appendix A, Appendix B and Chapter 3.3, respectively. As experience is gained, read about other commands available and see how they can be used to your musical advantage.

One important point: an underlying principle of the system is that all users contribute to its betterment according to their ability. Therefore, any comments from users regarding the system and, especially this document, are solicited. These should be communicated by sending mail to music (see UNIX mail command in Chapter Four), or verbally to the author.

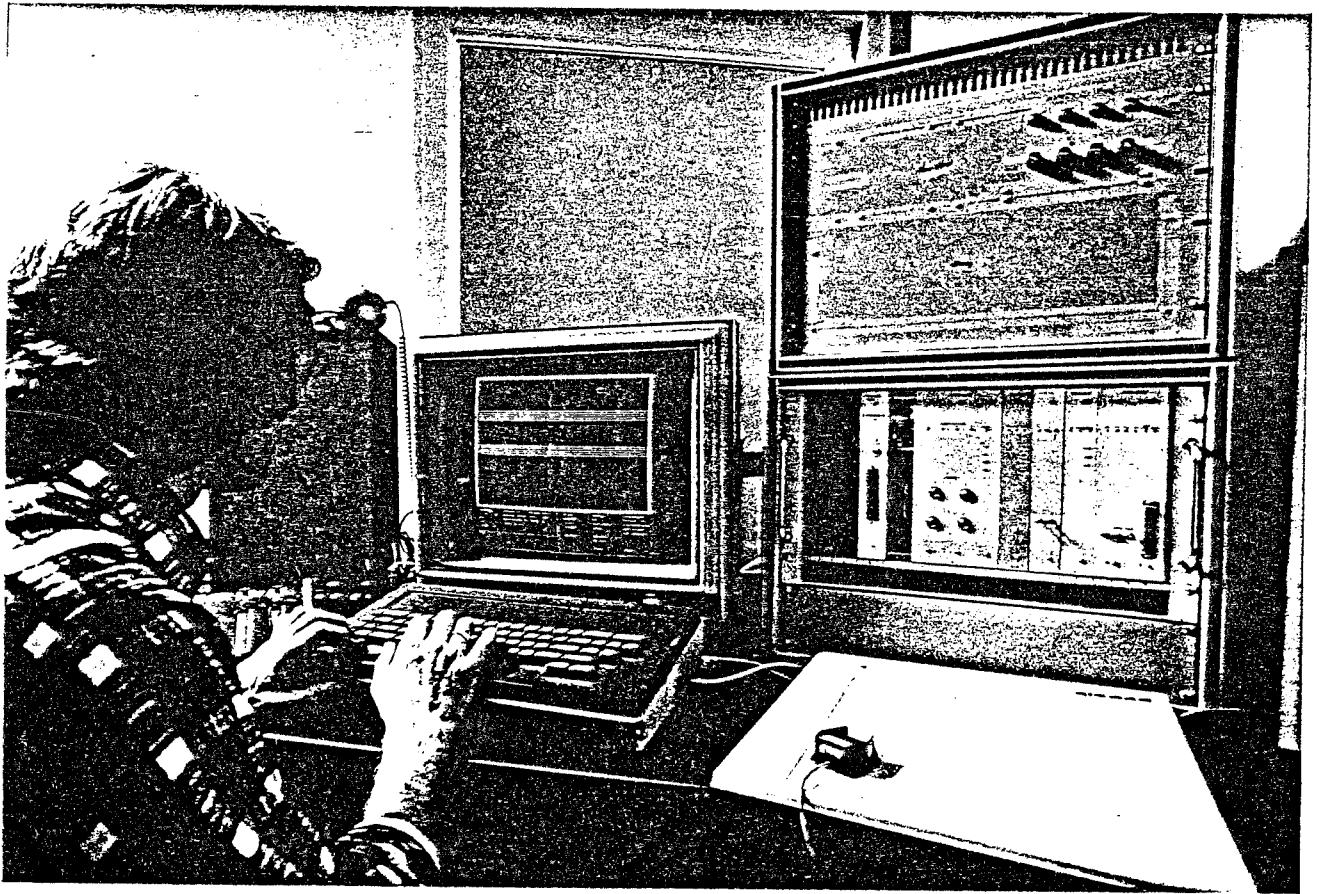


Figure 3. The Working Environment

2. TECHNICAL DETAILS

2.1 Introduction

This section is intended to provide the basic information required to make use of the system. Information will be presented on how to get "logged in" (i.e. started), how to read the documentation, and how to cope with other strange things which one does not encounter in, for example, a string orchestra.

2.2 Modes of Interaction

2.2.1 General

There are two main methods of interaction used in SSSP programs: interaction using graphics based techniques and that based on typing on a keyboard. The working environment is shown in Figure 3. The devices used in communicating with the computer are shown in Figure 4. Besides the CRT, on which information is displayed, there is a drawing device known as a *tablet*, both typewriter and piano type keyboards, a touch sensitive panel called the *touch tablet*, and two potentiometer-like devices called *sliders*. In addition, there are loudspeakers which enable the musician to hear the sounds generated by the *synthesizer*¹ (Buxton, Fogels, Fedorkow, Sasaki, & Smith, 1978).

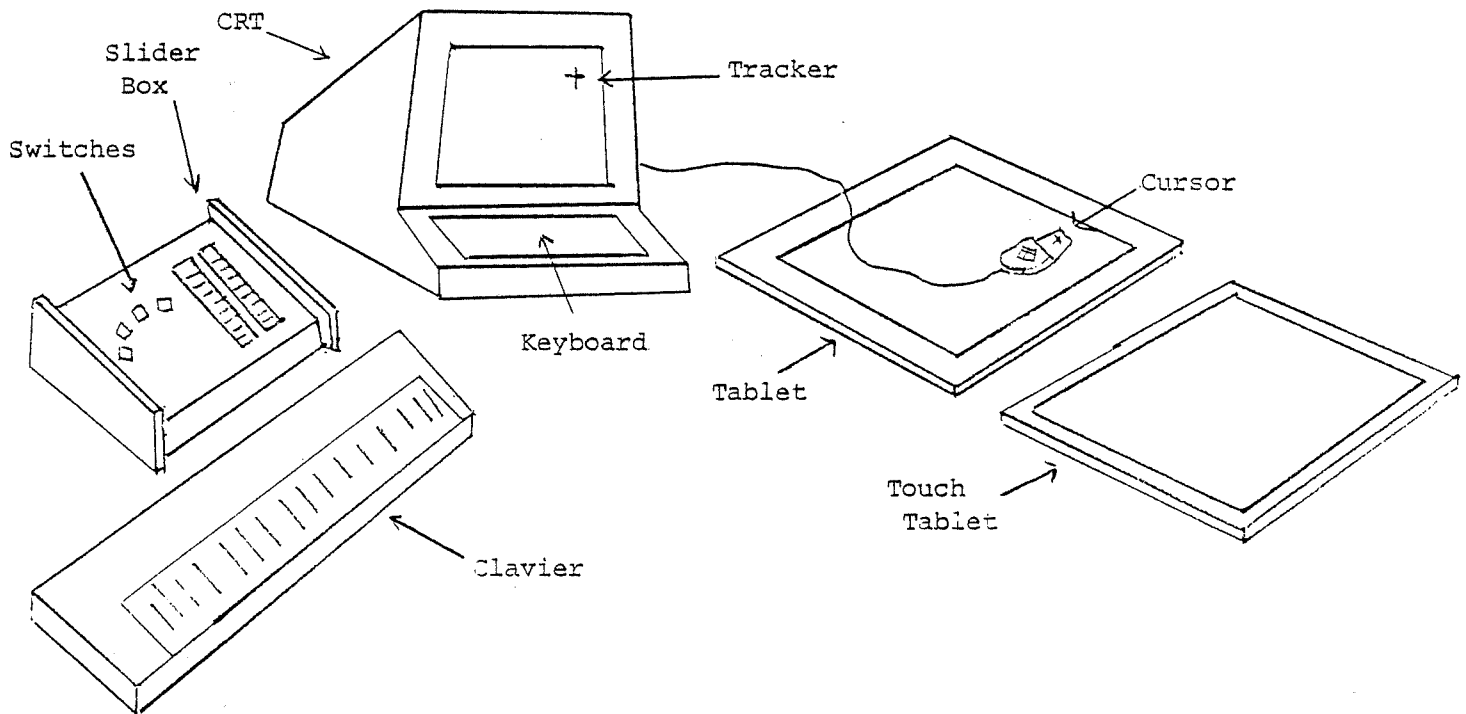


Figure 4. Communication Devices

In graphics programs, information is displayed pictorially on the television-like CRT. This has been illustrated in Figures 1 and 2.² With such programs drawing and pointing gestures can be used to define musical data and carry out various other tasks.

Typing, however, can play an important role in the composer-computer dialogue. For example, typing the following line would cause the score "minuet" to be orchestrated with the "timbre" sax:

orch minuet sax

1. The technical specifications of the synthesizer are summarized in Appendix G of this document.

2. These images were generated by the program *scritva*, which is one of the main tools for defining scores.

CHAPTER 2

As will be seen in Chapter Three, there are many such commands. Besides simply offering an alternative mode of working, one of the prime reasons for using typed commands is that there is only one graphics terminal on the system. On the other hand, there are several *alpha-numeric* terminals (i.e. terminals only capable of outputting text). Therefore, in order to allow as many people as possible to use the system at once, many functions are available using both the graphic *and* alpha-numeric modes. However, in order to protect the composer from having to memorize two sets of commands, there is only one name which applies to *both versions* (graphical and non-graphical) of a command. The system will automatically execute the correct version, depending on which terminal is being used. For example, when calling the command *funced* from the graphics terminal, you will get the graphics version; otherwise, you will get the alpha-numeric one.

2.3 Graphics

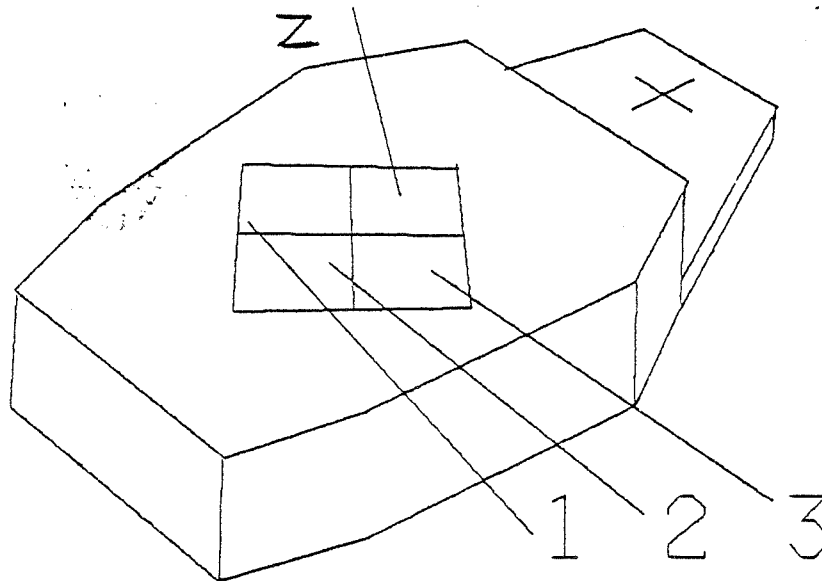


Figure 5. The 4-Button Cursor

The CRT terminal used in the music system has much in common with a television set in that it can present graphical information as well as text. Thus, data and controls can be spatially distributed on the screen, often in a pictorial manner (as has already been seen in some of the preceding figures). By displaying data in this way, it is easy for the operator to see the current status of the system and carry out transformations on the data.

By coupling the CRT with the graphics tablet we can exploit the full potential of the display. In this way, the tablet provides a means of pointing to, and interacting with, the displayed data. The CRT cursor (which we shall henceforth refer to as the *tracker*) can be made to follow, or *track* the relative position of the tablet *cursor*. This is seen in Figure 4, for example, where the cursor is positioned on the upper right-hand corner of the tablet, and the *tracker* on the upper right-hand corner of the display. We can point at anything which is displayed on the screen and, for example, select commands from the display in the same way you make selections from a menu.

Mounted on the cursor, which is shown in detail in Figure 5, are four buttons. In the remainder of this paper, they shall be referred to as the Z-button, and buttons 1-3. When the *tracker* is placed in a particular position, different things can be made to happen depending on which button is pushed. Thus, if you select something from the "menu" by pointing at it, depressing one button could signify that you want to play it,

and depressing another, that you want to save it..

2.4 Typing

2.4.1 General

To communicate with the computer -- even when using graphics programs -- a certain amount of typing is unavoidable. This section presents some important conventions which are used when entering information *via* the typewriter keyboard.

2.4.2 Special Characters

First, when typing, *always finish by pressing the "return" key*. Otherwise, the computer will never get your message. (Hitting "return" literally "wakes the computer up" regarding your existence!)

Second, if you make a typing mistake, you can simply backspace over the erroneous text (making use of the "backspace" or "BS" key), and retype from that point. If, on the other hand, the entire line is in error -- or not what you wanted -- you can erase it by typing an "@" character. You may then type the line which you intended.

Third, if you want to abandon what you are doing (for example, kill some program which you have invoked), you may do so simply by hitting the key labelled "RUBOUT" ("DELETE" on some terminals). This is especially useful when -- and unfortunately it does occasionally happen -- your program "dies". That is, your program will not respond to any stimulus from you. Note that when this happens, you should send mail to music explaining what program died, and under what circumstances. Hopefully we will then be able to clear up the problem.

Finally, in the drastic situation that all else fails, and your program will not even respond to "RUBOUT", then there is the final resort of typing CONTROL BACKSLASH. That is, the button labelled "CTRL" and the backslash key (\) should be pressed simultaneously.

In summary, there are certain "special" characters/buttons which you should know about when typing. These are:

RETURN: Push at end of *every* line.

BS: (Back-Space) Erase previous character(s).

@: Restart Current line.

RUBOUT: Kill current process.

CTL Backslash: Drastic killing of current process.

2.4.3 Naming Conventions

One common reason for using the typewriter keyboard is to specify names (for *scores* or *waveforms*, for example). Names are important, and anything which you create and want to save should be given a unique new name. Unless you really understand how things work, saving material under the system-generated names may result in that material being lost.

In naming material there are a few conventions to remember. First, *do not* use blank spaces in names: The name "pretty_score" is fine; "pretty score" is not. Second, make sure that your names are unique. For example, you may not have a *score* and a *waveform* of the same name. Finally, since nobody likes typing, keep your file names as short as possible. In particular, keep your names to 13 characters or less.

CHAPTER 2.

2.4.4 Miscellaneous

Unlike your normal typewriter, computer keyboards distinguish between the character "l" (lower case "L") and the number "1" (one). Therefore, when you are specifying numerical information, use the *number* "1" *not* the letter "l".

2.5 Getting Started

At this point, we shall give a summary of how to get started. With all but the graphics terminal, the first thing to do is push the red button mounted in the little black "Dmitri" box which is found beside the display. The question

Service?

will then be printed by the system, to which you should respond by typing the number "45" (without the quotes). On the graphics terminal, the first thing to do is to hit the key labelled "return". In both cases the computer will respond by outputting the prompt:

login:

In response you should type your user "ID". As always, terminate your line of typing by pushing the "return" key. The computer will then ask you for your "password", which is like a key to protect the privacy of your files. After typing your password (which will *not* appear on the screen), the system should respond with the "%" symbol. This indicates the system's readiness to receive commands from you (one of those described in Chapter Three). Note that the "%" character may be preceded by some messages to you, one of which might be "you have mail". If, for example, you do have mail - just type "mail" to read your messages. (See the description of the "mail" command in Chapter Four for more details.)

2.5 Documentation Conventions

There are several different commands described in this manual. These commands can be executed *when and only when* you receive the prompt: "%" from the computer. That is, the computer types a "%" character to indicate its willingness to accept a command. Once you have received the "%" prompt by the computer, you can invoke (i.e. cause to be executed) any of the commands described in this manual via the following general procedure: simply type the command name, which is then sometimes followed by one or more additional "words" or character strings called "arguments". If there are no arguments, one simply types the command name as in the following example:

who

which will give a list of all users currently working on the system. In general, the format for commands is as follows:

command argument1 argument2 argument3 ...

where the "arguments" communicate additional information when required by the command. An example of a command which takes one argument is:

play ussr

where "play" commands the system to perform the score "ussr". For ease and clarity of presentation we will now introduce a convention to be used in the documentation as regards command arguments. In the description of a command, arguments will be shown enclosed by two possible types of brackets: < > and []. The first pair is used to

indicate an argument which *must* be included with the command. An example would be the play command, whose simplest usage is:

```
play <scorename>
```

where the command must be followed by the name of a score.

The square brackets ("[" and "]") are used to indicate an optional argument which may or may not be included with the command. Again, we can use the "play" command as an example:

```
play [tempo] <scorename>
```

where we can provide an optional argument to indicate the tempo of performance. It is seen, therefore, that these two types of brackets provide a quick method of determining which arguments need and need not be specified with a command.

One common use of the commands described in this document is to take some existing musical material, transform it, and save the result. In many cases, the composer may want to replace the original material with the new, transformed version. This case is rather straightforward, as can be shown with an example:

```
retro fred
```

Here we use the command *retro* to make a new version of the score "fred" which is the retrograde of its original self. In this case, the original version is lost, and we are left with only the new, transformed, version. This is not what we always want, however. Working from the same example, we can examine the other case: where the composer wants to create a new (transformed) score, while still keeping the original:

```
retro fred : derf
```

Here we have a new score called "derf", which is a retrograde version of "fred" (which is preserved in its original form). In this example, we see one important convention used in the commands described in this manual. That is, the name of any new score being created is *always preceded by a colon (:)*. The colon simply indicates that the next argument is the name to be given to the new data being created. Combining what we have stated concerning both brackets and the use of the colon, we see that the usage of the *retro* command in the above example can be summarized as follows:

```
retro <scorename> [ ":" newscore]
```

where "scorename" is the name of the original score and "newscore" is the (optionally specified) name of the new score. Note that you do *not* type the quotes around the colon. These indicate that the colon must be typed exactly as it appears.

Finally, there is the question of the *order* of the arguments in the command's sequence. We feel that the user should be freed of the burden of having to remember the order in which multiple arguments must be specified. Therefore, in most commands, the order of argument specification is unimportant (except for colon preceding the optional "newname" - c.f. above). Therefore, the effects of the following examples (one of which we have already seen) are exactly the same:

```
retro fred : derf
```

and

CHAPTER 2

retro : derf fred

Any exceptions which require special ordering are pointed out in the documentation. (Note that the exceptions "make sense" and therefore remembering them is far less of a memory burden than memorizing the argument orders.) Again, if the documentation seems unclear or inadequate, send mail to music so that subsequent users need not suffer the same frustrations as you.

3. MUSIC COMMANDS

3.1 Objects

3.1.1 General

Programs in this category enable the user to define a personal palette of timbres. For our purposes, we use the term *object* to describe a set of timbral characteristics. Each object defined has a name defined by the user. This is analogous to defining and naming instrumental timbres such as "flute", "trumpet", etc. As will be seen in Appendix A, the waveforms and functions making up an object also have names, and the user must be conscientious of giving new names to these elements when they are defined. There are various methods and modes of defining objects; however any object can be used in the orchestration of any note.

3.1.2 Definition

3.1.2.1 bank is an extension of *objed*. It is a command which allows the specification, modification, and testing of objects which are defined according to the technique of additive synthesis. With *bank*, the user may hand sketch (or type) time-varying functions affecting the amplitude (and frequency) of up to 16 partials. Figure 6 is an example taken from *bank* showing a set of functions controlling the amplitudes of a set of harmonics over time. Detailed usage of the command is presented in Appendix A.

3.1.2.2 objed is the command to activate the object "editor"; that is, an environment for defining, modifying, and auditioning *objects*, or timbres. In order to aid the user in the editing process, the program allows the user to audition objects at different pitches, amplitudes, and durations. The *objed* program enables the musician to define objects using different techniques of sound synthesis. The modes currently supported include: fixed waveform, frequency modulation (FM), waveshaping (non-linear distortion), and VOSIM (voice simulation). (The command *bank* is used to specify objects according to the technique of additive synthesis.) Fortunately, one does not have to understand the details of the techniques in order to use the programs. On entering the program, the user is presented with the *fixed waveform* mode, which is the most simple to use. What appears on the screen is shown in Figure 7. One can explore simply by pointing. For example, pointing at the command *play* on the display will enable the sound to be heard. Pointing at the "potentiometer" labelled "frequency" permits the pitch to be changed, and so on. Usage:

objed

More detailed information on *objed* can be found in Appendix A of this manual.

3.1.3 Auxiliary Commands

3.1.3.1 pobj is a simple program which enables the listing of an object's data on a terminal. The data are listed in alpha-numeric (rather than graphic) form. Usage:

pobj <objectname>

where "objectname" is the name of some valid object.

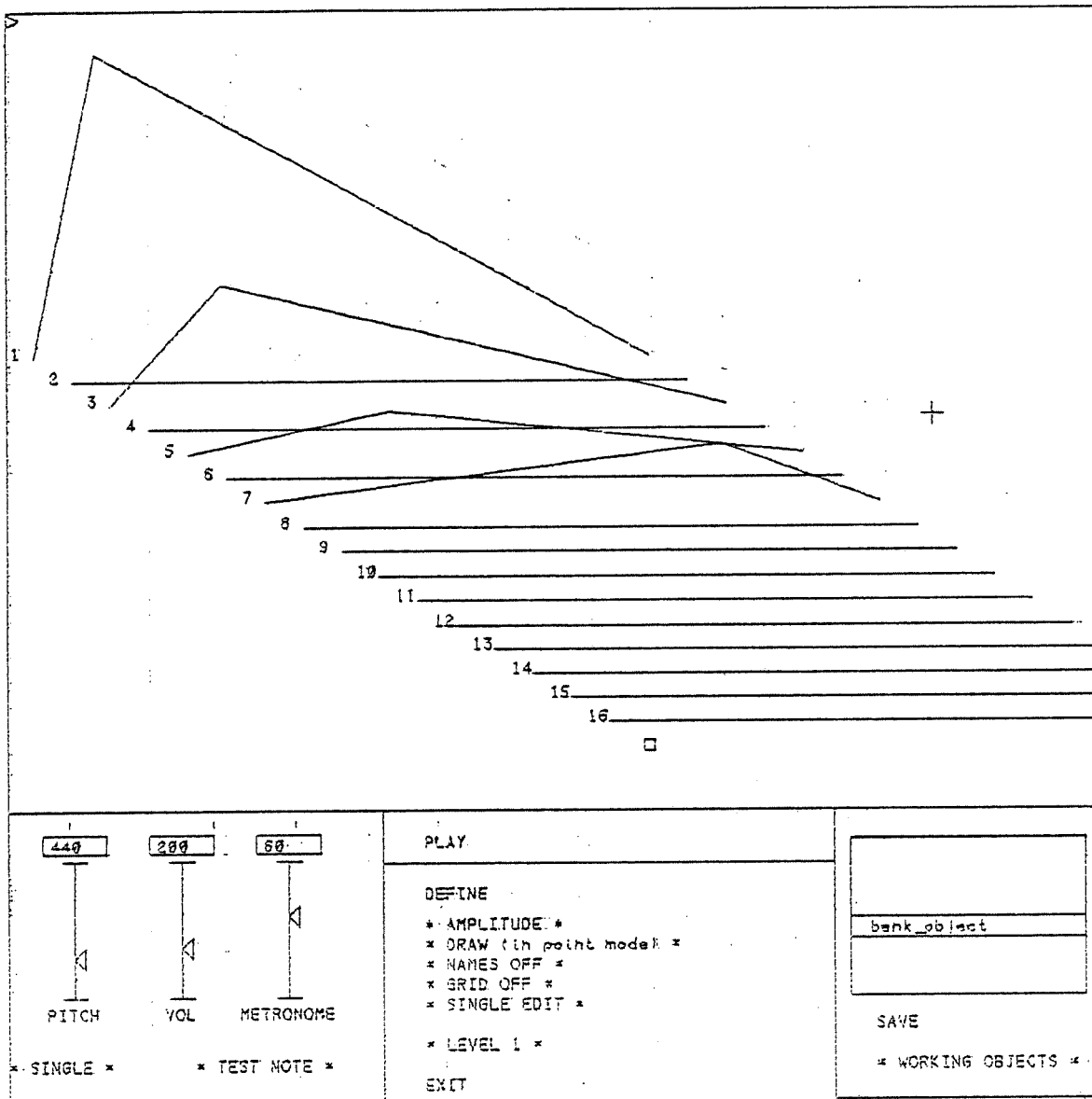


Figure 6. Additive Synthesis using 'bank'

3.2 Scores

3.2.1 General

There are various methods of generating scores. The available programs fall into two main categories: programs within which each note (and its attributes) are directly specified by the user, and programs which use compositional procedures to "generate" aspects of the score. In addition, new scores may be constructed by combining two or more previously defined (sub)-scores.³

3.2.2 Direct Specification

3.2.2.1 *record* is a program which allows scores to be created by having the computer record what is played on the piano-type keyboard, the *clavier*. The command is invoked by typing:

`record`

3. It is important to note that regardless of how a score is created (using *CMN*, random procedures, graphic notation, etc.), all scores are in the same internal format and therefore compatible (for mixing, etc.). Thus, a score created using *scriva* can be edited using the program *scad*. All scores and score programs are compatible.

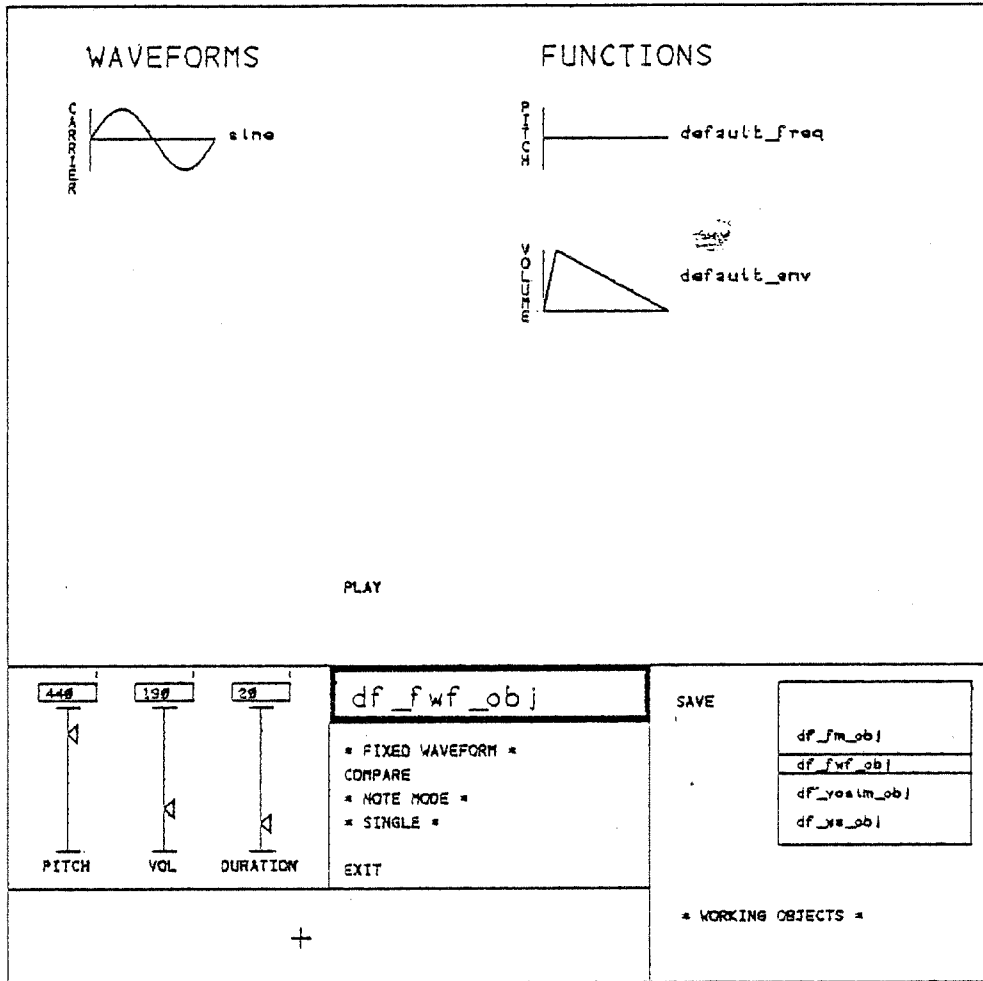


Figure 7. OBJED: Fixed Waveform Mode

This command requires no arguments. Once *record* is invoked, you will be asked to identify where you are working (upstairs or downstairs), in order for the system to know which clavier it should be monitoring. The system will then prompt you to switch to the appropriate "conducting" or "Lsi" terminal. This is the terminal on which all subsequent messages from the *record* command will appear:

The use of the program is straightforward, the options being explicitly given to you on the terminal. Simply, you may play on the clavier, and have what you performed played back (more than once if desired). You have the option of saving what you have played, re-recording it, or of playing something new. If you choose to save material, the program asks you for the name by which you would like the score to be known. Several different scores may be generated in one "record" session, each of which may be *fully polyphonic* (up to 16 voices), and of an arbitrary number of notes. The scores thus generated may be subsequently edited or modified using any of the commands found in this manual. To exit the program, switch back to your original terminal and depress the "RUBOUT" button.

3.2.2.2 *sced* like *scriva*, is a score editor; that is, an environment in which a composer can specify and modify the data making up a musical score. Unlike *sced*, the environment provided by *sced* is alphanumeric, rather than graphics based. Therefore, the

program can be used from any terminal, graphics or otherwise.

sced is based on the UNIX text editor (Kernighan, 1975b) which is called *ed*. That is, the commands and working methods are very similar; if you can use one, you can use the other. Usage for invoking the program is:

```
sced [scorename]
```

where *scorename* (optionally) specifies the name of the score to be edited. If the score indicated by the *scorename* argument exists in the composer's current directory, it is automatically loaded into the program.

A detailed tutorial explaining the program's use is given in Appendix C of this manual. A summary of the program's main commands is given below:

.	print current note.
.=	print current note's number.
a	add notes to score after current note.
i	add notes to score in front of current note (insert).
l	listen: play score on synthesizer..
f	print file (score) name.
h	help: ask anytime.
c	change the current note.
<int>	absolute or relative change of current note..
w	write (i.e. save) score..
p	print current note's data on screen.
q	quit editor..
[cr]	make next note the current note and list it.
e	edit a score.
d	delete indicated note(s).
\$	move to last note.
!<cmd>	temporary escape to shell.
r	read score and append after the current note.
t	copy note(s).
m	move notes.
mm	examine or define metronome marking.
lm	switch reset mode for interrupted listen.
jm	switch the join mode between mix and splice.

3.2.2.3 scriva is an editor for scores. It is a graphics program that allows you to add, delete, view, play, etc., notes in a score. The program allows scores to be edited and notated using various forms of notation, including CMN and "piano-roll" notation. Figures 1 and 2, which were taken from *scriva*, illustrate two different ways of notating the same musical material. Figures 8 and 9 give two other examples of *scriva*'s notational flexibility. Usage:

```
scriva
```

Additional information on the use of *scriva* can be found in Appendix B of this manual.

3.2.3 Compositional Routines

3.2.3.1 makescore makes a score of a specified number of notes. All notes are pitched at a4 and have a duration of a quarter note. The command is useful in combination with other commands such as *map* or *rand*. Usage is:

```
makescore <number of notes> [":" scorename]
```

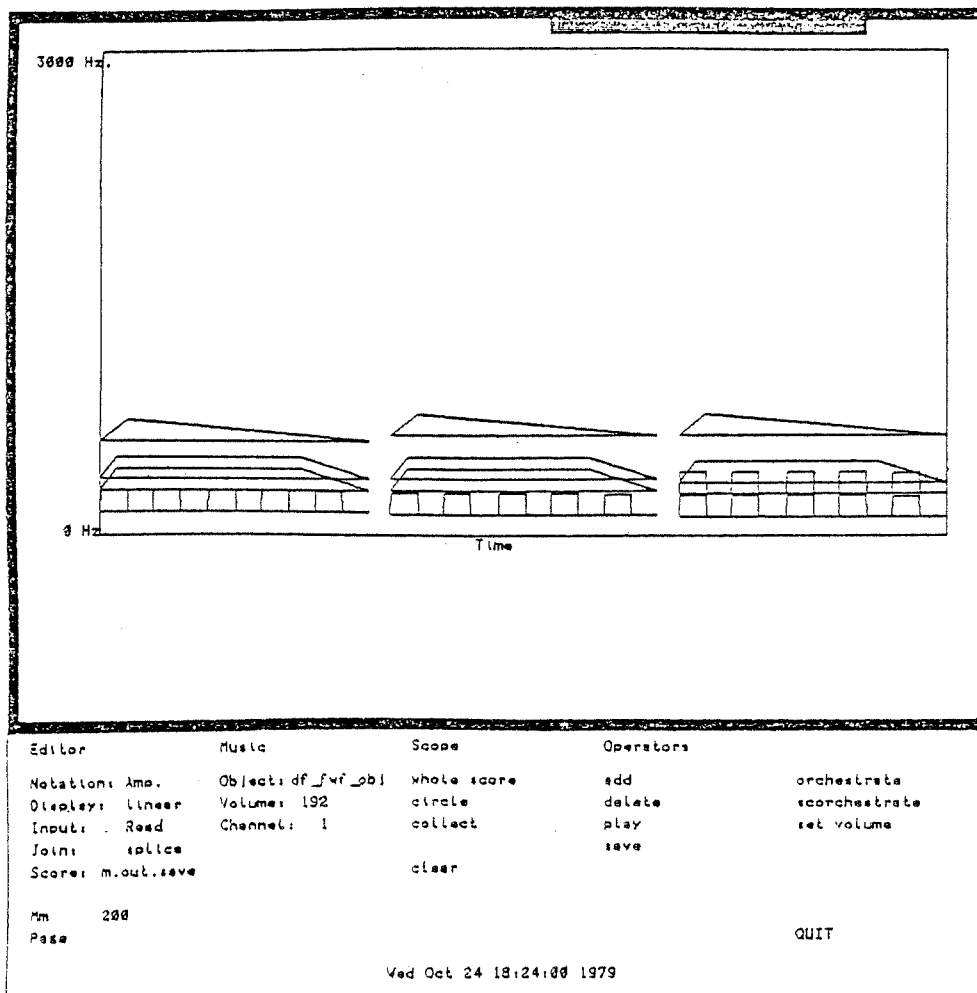


Figure 8. Envelope Notation in Cartesian Space

where the number of notes is specified as an integer. If the optional specification of the name for the created score is omitted, then the new score is named "m.out".

3.2.3.2 *prod* is a program for composing scores which permits random processes to be used in making decisions about the musical structure. An interesting aspect of the program is that it permits scores to be specified as tree-like hierarchic structures. This is in contrast to many programs which through-compose a work from beginning to end. The result is that relationships among structures at different points in the score can be easily defined. In addition, reoccurring material only needs to be specified once.

Prod is based on a linguistic model, that of formal *grammars*. In order to obtain a detailed tutorial on how to make use of the program, the reader is referred to Appendix D of this manual. One thing to note in passing, is that random selection features of the program need not be used. *Prod* can be a useful tool simply for assembling completed scores out of pre-composed fragments. In either case, it provides an elegant means of specifying the structural "recipe" of a composition.

3.2.4 Auxiliary Commands.

Editor	Music	Scope	Operators
Notation: Obj.	Object: obj.drone	whole score	add
Display: staves	Volume: 150	circle	delete
Input: Read	Channel: 1	collect	play
Join: mix			save
Score: save2.		clear	
Key: C			
Mm: 30			
Page			QUIT

Nov 2 09:14:31 1979

Figure 9: Notation Highlighting Orchestration

3.2.4.1 `cksize` is a utility routine which summarizes the amount of space taken up by objects, scores, functions, and waveforms for any score. Aside from interest's sake, the only use of this command has to do with the *conduct* program. In order to get optimal resource utilization, the first statistic given: "Number of Core Table Entries", is important. This number is the value that ideally should be input to the first question posed by *conduct*, "Sym Cnt?". The usage is:

```
cksize <scorename>
```

where "scorename" is the name of a valid score, including a "file folder". (See the command *makeff*.)

3.2.4.2 `dur` is a utility command which will give the duration of a score when performed at a specified metronome marking. The usage is:

```
dur <score> [mm] :
```

where "score" is the name of the score whose duration is to be found, and "mm" is the metronome marking. If the metronome marking is omitted, then mm=60 is assumed. The score duration is given in both beats (quarter-notes) and minutes and seconds.

CHAPTER 3

3.2.4.3 orch enables the user to orchestrate a score with one (and only one) of the objects which he has defined. Usage:

```
orch <objectname> <oldscore> [":" newscore]
```

where "objectname" is the name of the object with which the score "oldscore" is to be orchestrated. The new orchestrated score can be optionally placed in a new score file, thus leaving oldscore unchanged. More sophisticated tools for orchestration can be found in each of *scriva* and *sced*.

3.2.4.4 prune Unfortunately, the number of voices available at any one time is limited. This is due to the fact that the SSSP synthesizer has only 16 sound generators. Therefore, the maximum number of voices that can ever be performed simultaneously is 16. Furthermore, since some techniques of sound synthesis utilize more than one oscillator per voice (fm and waveshaping both require two oscillators per voice), the user is often limited to 8 or fewer voices. Sometimes this is frustrating; however, the composer should simply accept the limitations of working with resources more comparable to a chamber ensemble than to a symphony orchestra.

The *prune* command is provided to enable the composer to strip out of the score those notes which cannot be played by the synthesizer. Simply, the command goes through the indicated score and ensures that there are never more than a specified number of voices playing at one time. Excess notes are deleted, or "pruned" from the score. Usage of the command is as follows:

```
prune <score> [maxvoices] [":" newscore]
```

where maxvoices indicates the maximum density (if unspecified 16 is assumed), and newscore is the name of the new "pruned" score. If newscore is omitted, the pruned version is saved in score. Note: the *prune* command does not alter the timing of your composition; it merely cuts down on its size.

3.2.4.5 invert enables you to create an inverted version of a score. That is, every interval is inverted in direction. The first note of the score remains the same. Usage:

```
invert <score>
```

or

```
invert <oldscore> [":" newscore]
```

3.2.4.6 iscale is a command that enables you to scale the intervals in a score. That is, you can compress or expand the size of the score's pitch intervals. The effect of the command can be seen if we take a three note score as an example. Let us say that the second note is an octave above the first, and the third note is a major second below the second (e.g. a4 - a5 - g5). If we scale this score by .5 (1/2), the new score will consist of a tri-tone (augmented fourth) rise followed by a semi-tone fall (a3 - d4# - d4, in the above example). Simplest usage is as follows:

```
iscale <score> <factor>
```

Where "score" is the name of a valid score and "factor" is numerical value (4, .5, 3.4, etc.) by which the intervals are to be scaled.

Note that a negative scaling factor is perfectly legal. With a little thought, therefore, we see that the effect of using *iscale* with a scaling factor of "-1" is the same thing as inversion. Thus, the following two examples are musically identical:

```

iscale fred -1
and
invert fred

```

In the above examples, the actual intervals of the original score would be changed. If one wants to create a new score which is a scaled version of a preserved old score, this is also possible. Usage in this case is as follows:

```
iscale <oldscore> [":" newscore] <factor>
```

Where factor is as above, "oldscore" is the original score and "newscore" is the name for the new scaled version of "oldscore".

3.2.4.7 makeff is a command which lets you group several scores together into a single "file-folder." The purpose in so doing is to enable them to be shipped to the *conduct* program under a single name, rather than repeatedly typing the name of each score. The command usage is as follows:

```
makeff <file-folder name>
```

The command then prompts you as to how to proceed. Typing "h" at any time will provide help in the form of on-line documentation. In general, the command allows you to add and delete score names to be included in the file folder. If there already exists a file-folder of the name specified as the argument to the command, that file-folder will be edited; otherwise, a new file-folder will be created under the new name. Note that the name of all file-folders *must* end in the letters "ff;" furthermore, scores being used by the conduct system must *not* end in "ff."

3.2.4.8 map is a command to transfer characteristics of one score onto a second score. Thus, for example, we can cause the sequence of notes making up one score to assume the pitches of the sequence of notes of another score. The score whose characteristics are being copied is referred to as the *pattern* score. The score which is assuming these new characteristics is called the *transformed* score. The process of having one score assume characteristics of another we call *mapping*.

Various parameters can be mapped from one score to another: pitch (*i.e.* frequency), note durations, entry delays (thus rhythm), amplitude, and timbre. In the mapping process, the number of notes in the transformed score does not change, and only those characteristics being assumed from the pattern score are changed. The user specifies which characteristics are to be mapped. If there are fewer notes in the transformed score than the pattern score, then the pattern simply does not complete, since no new notes are generated. On the other hand if the transformed score is longer than the pattern, then the time pattern will repeat until the end of the transformed score is reached. Usage of the commands is as follows:

```
map <pattern score> <transformed score> <flag(s)> [":" newscore]
```

Note that the order of the pattern and transformed scores is important in this command. If the name of the new score is not specified, the new score will overwrite the original version of the transformed score. The characteristic flags are one or more arguments specifying what characteristics of the pattern score are to be mapped. Each flag consists of a minus sign ("-") followed by a single character. The flags and their meanings are as follows:

CHAPTER 3

FLAG	MEANING
-p	map pitch
-r	map rhythm (entry delay)
-d	map duration
-t	map time (rhythm & duration)
-a	map amplitude
-o	map objects

An example of the use of the command is as follows:

```
map mel beat -p : complete
```

which maps the pitch structure of score "mel" onto score "beat" and stores the newly created score under the name of "complete."

3.2.4.9 mix is used to mix several scores together. It assumes that all the scores start at the same time. (However, the command *sdelay* can be used to get around this limitation.) The scores to be mixed appear as arguments to the program. The orchestration of the scores being mixed is maintained. An example of the use of this program is:

```
mix s1 s2 s3
```

where "s1" to "s3" are the names of the scores to be mixed. The new mixed score in this case is stored in the score "m.out". There is no practical limit on the number of scores that can be mixed together. The only real constraint is to remember that the synthesizer is currently only capable of playing a maximum of 16 simultaneous voices.

Carrying on from the previous example, the user may specify the name of the new score, as in the following example:

```
mix s1 s2 s3 : poly
```

where "poly" is the name of the new score. In summary, the usage of the mix command is:

```
mix <score1> <score2> ... <scoren> [":" newscore]
```

Cautionary note: If there previously existed a score *m.out* before calling this command, it will be overwritten (i.e. lost) if the new score is not explicitly named. Finally, the two score editors *sced* and *scriva* both provide more flexible mixing facilities.

3.2.4.10 mode is a command to enable you to change the *mode* of a score. That is, it enables you to change the mode of a score from the major to the harmonic or melodic minor, for example. (A brief disclaimer: *mode* has no musical intelligence, so it works more like a musical "sieve".) In fact, the command lets you change the mode of any score to any of the common modes (such as the twelve "church" modes), or to any mode of your own invention. The notes of the score being transformed need not even "fit" with the frequencies of the notes of the piano; they will be treated as if they had the frequency of the closest pitch. Thus, one of the spin-off uses of this command is to convert a score whose notes fall anywhere in the frequency domain to one whose notes fall on one of the 12 pitch classes of the diatonic scale.

The usage of the command is as follows:

```
mode <score> <mode> <final> [":" newscore]
```

MUSIC COMMANDS

where the *score* argument indicates the score to be transformed, and the optional *newscore* argument indicates the name of the new transformed score if it is to be different than that indicated by the *score* argument.

The *mode* argument indicates the mode to which the indicated score is to be transformed. You can specify the mode in a variety of ways. These are as follows:

- By a Roman numeral (I through XII), indicating the Authentic (odd) and Plagal (even) church modes.
- By an Arabic numeral (1-7) indicating the degree in a major scale on which the equivalent 7-note mode would begin.
- "-m", indicating that the mode is the melodic minor (ascending).
- "-h", indicating that the mode is the harmonic minor.
- "-d", indicating that you wish to define your own mode. On using this option, the program will prompt you how to specify the mode.

Finally, the argument called *final* must be specified in order to indicate the *final* (or tonic) of the score to be transformed. This is simply indicated by typing the pitch class. Examples of legal *final* specifications are: c#, ab (Note lower case "B" is used for flat), d, etc.

3.2.4.11 *pscore* prints the contents of the given score on the terminal. For each note it gives the frequency, amplitude, duration, time to next note and object. An example of its usage is:

```
pscore <scorename>
```

where *scorename* is the name of a valid score. If the score is more than a couple of notes long, chances are its listing will not fit on the screen at once. Therefore, it is often useful to print the score out page-by-page. This is done as in the following example:

```
pscore fred | p
```

or

```
pscore fred ^ p
```

where "| p" and "^ p" both indicate paging. (Note that the '^' character can always be used as a synonym for '|'.) To get the next page, just press "RETURN". To stop the printing, type the RUBOUT character.

More than one score may be printed out with this command. Usage is as above except that any number of score names may be given. An example of this type of usage would be:

```
pscore fred charley joe
```

which would print the scores "fred", "charley" and "joe" in the order given.

If you want a printout of the score on the line-printer rather than on your terminal, usage is:

```
pscore <scorename> | lpr
```

Note that in this case, it is important that the "| lpr" be the last arguments. If you are printing multiple scores out on the line printer, it may be desirable to start each score

CHAPTER 3.

on a fresh page. For this purpose, the "-f" flag is used. It will cause a formfeed on the printer before printing out the scores following the flag in the command line. The form feeding can be disabled (the default state) by including a second "-f" flag in the command line after the names of all the scores that are to be separated. Usage in this case would be as follows:

```
pscore -f jazz madrigal -f fugue | lpr
```

This would cause the scores jazz and madrigal to be printed on fresh pages, but with the score fugue following immediately without a formfeed.

A feature of interest to programmers only is the verbose mode of output. When this mode is invoked by the appearance of a "-v" flag in the command line, all scores up to the next occurrence of a "-v" in the command line or the end of the command line itself will be printed out along with the contents of their symbol tables. The "-v" flag may appear any number of times in a command line and each occurrence has the effect of switching the state of verbosity. For example, the following command will print out the score frank with its symbol table and the score fred without it.

```
pscore -v frank -v fred
```

Note that the same effect (albeit with the scores in the other order) could have been achieved by the command:

```
pscore fred -v frank
```

When in verbose mode the score printout of object names is accompanied by the object's symbol table index.

3.2.4.12 rand is a command to randomize, within user defined limits, a specified parameter of all the notes in a score. This command allows a user to specify a "tendency" within which the computer will select, according to random chance, values for a specified note characteristic. In the simplest case an attribute may be randomized relative to its current value. Usage is then:

```
rand <score> <flag> <range>
```

where <score> is the name of an existing score and <flag> indicates which attribute is to be randomized. The flag may be one of the following:

- f (frequency)
- v (volume)
- d (duration)
- r (rhythm or entry delay)
- t (timing: duration and delay).

The <range> argument gives the range of values above or below the current value in which the attribute may fall after randomization. An example which illustrates this type of usage is:

```
rand fred -v 20
```

which results in a new version of the score "fred" where the respective dynamic levels of all the notes have been re-set to some new value that differs from the old by no more than than twenty units (or about one dynamic marking). The original score may be preserved by giving a new name to the randomized version through the colon (:)

argument as follows:

```
rand fred -d 1/4 : newfred
```

where "fred" remains unchanged and "newfred" is the name under which the randomized version of "fred" is saved. Note that the range is given as a beat ratio that limits the note durations in "newfred" to differ no more than a quarter note's duration from the corresponding durations in "fred".

The randomization can be made to occur about some value other than the current value of an attribute. In this case the usage is:

```
rand <score>.<flag> [center] <range> [":" newscore]
```

where "center" is the value about which the randomization is to occur. An example of this type of usage is:

```
rand fred -f 440 220 : joe
```

with the result that the new score "joe" has the same orchestration, volume and time relationships as "fred", but is composed of notes with random frequencies lying somewhere in the range of 220 Hz (440 minus 220) to 660 Hz (440 plus 220).

In the above examples, the center and range were given as constants (i.e. a specific numerical value: 20, 1/4, etc.) that did not vary over the course of the score, but they may also be given in terms of time-varying *functions* (defined using *funced* or *objed*). This way a dynamic, rather than constant, value may serve as the center or range (or both). The function is first scaled such that the function's duration matches the score's duration and then the function's values, which range from 0 to 1, are adjusted to the range of values which a particular attribute may assume.

When time-valued characteristics are being randomized, the center and/or range values may be given either as beat ratios (1/4, 1, 1/8) or as functions. In applying functions to time attributes, the interpretation of function contours by the program is basically logarithmic. A function value of .5 corresponds to a whole note, and each increase (or decrease) of .0625 of the function value corresponds to a twofold increase (or decrease) of the time value. A function value of zero, however, is taken literally, rather than being considered equal to 1/256. Thus, any of the possible values that a duration or entry delay may assume can be represented by a function.

Since zero is a legal value for the range, the randomization may be confined to portions of a score. This is achieved by defining a function whose value is zero at points where no randomization is desired and which assumes some other value where randomization is to occur. Using this function as the range argument will then cause randomization to occur only where the function value is non-zero.

Finally, the above examples would produce scores where the randomization is said to be "linear", that is, the values obtained are spread evenly over the allowed range (as is defined by the center and range). An alternative exists whereby the user may specify that the random values are to be distributed "normally". This means that the values would "cluster" about the center in the form of the familiar bell-shaped curve. Usage is then as follows:

```
rand <score> <flag> [distribution] [center] <range> [":" newscore]
```

where "distribution" is one of the flags -l (linear) or -n (normal). If normal distribution is desired, strictly speaking, the center and range arguments are actually interpreted to be the mean and variance, respectively, of the distribution. If unspecified, the

CHAPTER 3

distribution is assumed linear.

3.2.4.13 relpos is a utility program which allows the composer to determine when, relative to the duration of an entire score, a particular event occurs. It is of particular use when defining functions which are to be used to affect some aspect of a score (such as in controlling dynamics using *setvol*). That is, it helps determine the relative position of reference points in the score which may be of use in designing such functions.

The user must specify to the program the event whose position is to be determined, and the name of the score in which it occurs. The event is identified by its note number (i.e. 1 for the first, 2 for the second, ...). The program prints out a fraction which indicates how much of the score has elapsed before the indicated note begins. The program also provides the ability to determine the proportion of the score which remains following the start of the indicated note (that is, the inverse of the fraction normally output by the command). This is accomplished by preceding the note number by a minus (-) sign. Finding the relative position of the 8th note of "minuet" would, therefore, be as follows:

```
relpos 8 fred
```

while finding the portion of the score following the same note would be obtained by typing

```
relpos -8 fred
```

3.2.4.14 retro is a command which enables you to produce a retrograde version of a score. Its usage is as follows:

```
retro <oldscore> [":" newscore]
```

Where "oldscore" is the name of a valid score which you want in retrograde. If a second name is specified, the new retrograde score will be given the name "newscore". Otherwise the original score is made retrograde, retaining its original name.

3.2.4.15 rotate permits the values of one or more note parameter to be shifted, or rotated, with respect to the rest of the score. For example, the frequencies could be rotated by one, causing the second note to assume the pitch of the first, the third that of the second, etc., with the first assuming the frequency of the last. Thus, the pitch sequence "C, D, E, F" would become "F, C, D, E".

Parameters can be rotated by more than one step, and can shift in either direction. Thus, rotating the sequence "C, D, E, F" by 3 would result in "D, E, F, C". The same result could have been achieved in this example by a rotation of -1. In all of the examples seen thus far, only the pitch/frequency parameter was affected. If the original duration of a note was a quarter note, so it remains.

Rotation can be applied to any and all parameters. Rotating all parameters by -1, for example, would have the same effect as detaching the first note, and splicing it onto the end of the score. One can, however, rotate parameters together, such as duration and entry-delay, or amplitude, duration, and orchestration.

The simple usage of the command is to type its name:

```
rotate
```

The program will then ask you, in a conversational manner, for the features which you want to rotate. Alternatively, all (or some) of the details can be specified in the command line, thereby avoiding an otherwise verbose dialogue. The detailed usage is:

```
rotate [score] [parm list] [degree] [":" new score]
```

where degree is the magnitude of the shift (either positive or negative); and "parm list" is the list of parameters to be rotated. This list may contain any or all of the following values:

FLAG	MEANING
-freq	frequency
-vol	volume
-obj	object (orchestration)
-chan	output channel
-dur	note duration
-rhy	entry delays (rhythm)
-time	same as -dur and -rhy combined

When used in the command line, the "flag" for each parameter must be preceded by a minus sign (-), as shown. The default assumed is that all parameters are to be rotated.

3.2.4.16 scorch enables you to "scorchestrate" a score with another "sub-score". That is to say, instead of orchestrating the notes of a score with an object, you do so with a score. For example, if we have a "master" score consisting of a 3-note motif as well as a "sub" score consisting of a major triad (any spelling and root), then, scorchestrating the master score with the sub score will result in a new score which is a sequence of 3 triads whose rhythm and root notes are the same as the 3 notes of the master score. Usage of the command is as follows:

```
scorch <master-score> <sub-score> [":" newscore]
```

where the arguments are explained above. If no target score is specified, the master-score is modified to the new version. NOTE: unlike most commands, the ordering of the arguments to the scorch command is important; namely, the master-score's name must precede the sub-score's in the argument list. This is because both arguments are scores, and the program cannot read your mind as to which is which. The placement of the (optional) new score argument is open, as long as it is preceded by a colon.

For those still confused as to what scorchestration really is, see Appendix D where it is explained in the context of *scsd*.

3.2.4.17 sdelay is a "kludge" program. It is used to place a rest at the start of a score. Therefore, when that score is mixed with another (using the *mix* command), its entrance will be delayed by the duration of the rest at the beginning. (This type of mixing is usually better performed using *scsd* rather than *sdelay* and *mix*.) Usage:

```
sdelay <oldscore> <delay> [":" newscore]
```

where *delay* is a fraction specifying the duration of the rest. An example of the command's usage, defining a delay of a thirty-second note is:

```
sdelay original 1/32 : modified
```

where "original" and "modified" are the names used for the original and new scores, respectively. Note that durations are based on a 1/4 note as having a duration of one "beat", and the number of beats-per-second determined by the metronome marking. Finally, note that the same effect can be gained with more control in both *scsd* and *scriva*.

CHAPTER 3

3.2.4.18 setchan enables you to set the output channel for an entire score. If you have, for example, a score made up of four "parts" or "sub-scores", you can use "setchan" to assign each "part" to an independant channel before merging the "parts" into the master score. Usage is as follows:

```
setchan <scorename> <channel>
```

Where "score" is a valid score name, and "channel" is an integer (1 to 4) indicating one of the 4 channels of the synthesizer.

3.2.4.19 setvol is a command to (re-)set or adjust the volume of the notes in a score. There are two ways in which this command can be used. (1) Score note volumes can be set to new, absolute, values specified by the user. This we refer to as *absolute* mode. (2) Score note volumes can be adjusted up or down a specified amount from their current values. This we call *relative* mode. As will be seen below, the command allows the user to specify values -- in both relative and absolute modes -- either as a constant, which has the same effect on the score from beginning to end, or in terms of a function whose value changes through the course of the score. Thus, in the former case, all volumes of a score can be set to some uniform value (absolute mode), or turned "up" or "down" by some uniform amount (relative mode). In the latter case, note volumes in a score can be set to follow some contour such as a crescendo-decrescendo (absolute mode), or in relative mode, we can adjust the volume of different parts of the score "up" or "down" by differing amounts. Each alternative is explained with examples below.

In setting volumes to a constant value in absolute mode, the usage of the *setvol* command is as follows:

```
setvol <scorename> <volume> [":" newscore]
```

where "scorename" is the name of a valid score whose volume is to be adjusted, and the "volume" argument is an integer in the range of 0 (min) to 255 (max), specifying the volume to which the notes are to be set. (Note: a value of 0 is inaudible, 190 is about mf, and a change of about 20 is equivalent to a change of one dynamic marking.) In the following example:

```
setvol jazz 255 : tooloud
```

we have made a "clone" of the score "jazz" whose notes are all set to the maximum volume (255).

In order to effect a relative change, that is uniformly turn the volume of a score "up" or "down" by a specified amount, the command is as follows:

```
setvol <scorename> <offset> [":" newscore]
```

The "offset" argument specifies the amount and direction that each note's volume is to be adjusted. It is specified as a signed integer (such as "+20" or "-10"), within the range of 0 - 255. The offset is simply added to the volume setting of each note in the score. The result is that the relationship of the amplitudes among the notes of the score remains the same; the volume is simply altered up or down. The following example

```
setvol jazz -20
```

reduces the note volumes of the score "jazz" by about one dynamic marking.

To summarize to this point, if the numerical argument to the *setvol* command is preceded by a plus (+) or minus (-) sign, it is interpreted as an offset in relative mode. Otherwise, it is considered an "absolute" amplitude value.

Returning to absolute mode, if we want to set the note volumes to follow some contour, we can use a time-varying *function*. A function can be used to define the amplitude "envelope" of a score, just as it can be used to define the envelope of a single note. Any function (such as those defined using *funcned* or *objed*) will do. Usage in this mode is:

```
setvol <scorename> <function-name> [":" newscore]
```

How then are the amplitude values derived from the function? First, the function is "stretched" so that its base exactly fits the overall duration of the score. Then, for the start of each note in the score we see where we are with respect to the overall score duration, and find out what the instantaneous value is at the corresponding point in the function. Thus, if we are at the middle of the score, we find the "y" value for the function half-way along its base. The range of function values are considered as being proportional to the range of possible amplitude values (0 - 255). Thus, if the instantaneous value of the function is .5, the note volume becomes 127 (255 times .5). Similarly, a function value of 1 results in a volume setting of 255. Thus, in the following example:

```
setvol jazz invec
```

if "jazz" is a score, and "invec" is a function having the shape of an inverted "V" (start at 0, reach 1. at midpoint, drop to 0 by end), then the score will fade in from nothing, reach maximum volume by the middle, then fade out by the end. Note that in this example the first and last notes will be inaudible (their volumes being 0), so to get more of a crescendo effect, the start and end points should be about .5, rather than 0.

Functions can also be used in relative mode, to specify varying degrees of volume adjustment during the course of a score. In this case, the "y" value of the function at the point corresponding to the start of a note is used to determine the relative *offset*, rather than the absolute volume as was previously seen. In this case, we have a slight problem, in that the function must be able to express a change either up or down. In relative mode, therefore, we must treat the function as having a positive side and a negative side. The situation is just like that seen with the function affecting pitch in objects, which can specify a glissando both up and down from the notated pitch. Using functions in relative mode in *setvol*, we interpret the middle "y" value (.5) as being zero, or no offset. The range of the upper half of the function (.5 to 1.) is interpreted as covering the range of allowable positive offsets (0 to 255). The range of the lower half of the function (.5 to 0.) is interpreted as covering the range of allowable negative offsets (0 to -255). Thus, a function remaining at .5 means no change. The further the function value rises above .5, the more the volume is increased. The further the function descends below .5, the more the volume is turned down:

We know if a function is to be interpreted in relative or absolute terms by whether it is preceded by a '+' or '-' sign or not. (Normally, use '+' in relative mode. The effect of the '-' is to use the inversion of the function.) The following is an example of using a function in relative mode:

```
setvol jazz +curve
```

Finally, there are two additional points to make concerning the command. First, it can be used to avoid the distortion due to overloading. That is, if the sound distorts, turn the volume (of the notes) down. Second, the command *rand* is rather useful as a complement to *setvol*, as a means of introducing random variation into the score dynamics.

CHAPTER 3

3.2.4.20 `splice` is used to "splice" several scores together in sequence. That is, they are strung together one after the other. The scores to be spliced are passed to the program as its arguments. The user may also (optionally) specify the name of the new score being created. (If the name of the new score is left unspecified, the new score is named "m.out" by default.) In splicing scores together, all aspects of the component scores (such as orchestration) remain intact. An example of the use of this command is:

```
splice s1 s2 s3
```

where "s1" to "s3" are the scores to be spliced. In this case, the new score is unspecified, and is named, therefore, "m.out". The next example illustrates how the composer may specify the name of the new "composite" score being created:

```
splice s1 s1 s1::s2
```

In this case, we are creating a new score "s2" which is made up of three repetitions of the score "s1". Notice that it is perfectly legal to splice a score onto itself in order to obtain a repeat (often saving tedious duplication of data specification). Second, remember that the name of the new score must be preceded by a colon (":"). We can summarize the usage of the splice command as follows:

```
splice <score1> <score2> ... <scoren> [":" newscore]
```

There is no real limit on the number of scores which may be spliced together. One cautionary note, however; in creating a new score, any previously existing score of that name is overwritten, and consequently lost. Protection: if you want to save the old version of the file, choose another name, or rename the old file (for example m.out) using the UNIX mv command (see Chapter Four).

3.2.4.21 `transp` is a command to transpose a score. In its simplest form, the user need only indicate which score is to be transposed, and to what pitch. The degree of the transposition can be specified in two ways: by giving the pitch on which the transposed score is to start, or, by specifying how far (up or down) the score is to be transposed. For fairly obvious reasons we call these *absolute* and *relative* transposition, respectively.

In absolute transposition, the new start note can be specified either as a pitch (in the form c3, a4#, d6b, etc.) or as a frequency (eg., 440, 325.5, etc.). Usage in this case is:

```
transp <score> <startfreq>
```

Where "score" is a valid score and "startfreq" is the start note of the new score, specified as described as above.

In relative mode, the user can specify how far a score is to be transposed up or down in terms of either semi-tones or Herz. When specifying a relative change in semi-tones, the value specifying the degree of transposition is made up of three components: a sign (+ or -) indicating direction, a number indicating the number of semitones, and the letter 's' indicating "semi-tone". An example of transposing the score "fugue" up a major third (4 semi-tones) would be:

```
transp fugue +4s
```

Notice that there are no spaces between the sign, number, and 's'. The number which indicates the degree of transposition need not be an integer. Thus, the effect of the following example would be to transpose the score "lower" down a quarter-tone.

transp lower -.5s

If the 's' is omitted in relative mode, the number indicates the number of Herz that the score is to be shifted up or down. Using this feature, the composer should keep in mind that the score will no longer be "in tune". (Why? Because frequency intervals can correspond to different pitch intervals.) That may, however, be just what you want!

As with other commands, the original version of the score may be kept intact by the provision of an additional argument which specifies the name of the new, transposed score. Usage then has the form:

transp <oldscore> <where-to> [":" newscore]

Where "oldscore" is the unaltered original score; "newscore" is the name of a new version of "oldscore", and "where-to" indicates where "oldscore" is transposed to (in either relative or absolute terms):

3.2.4.22 **tscale** is a command which enables you to "compress" (i.e. diminish) or "expand" (i.e. augment) the time-scale of a composition. In its simplest form, **tscale** causes all time values of the score to be scaled. Therefore, note durations are adjusted, as is the score duration (and hence, tempo). Usage in this case is:

tscale <score> <factor>

Where "score" is the name of a valid score, and "factor" is the factor (a value such as ".5", "2.25", "3", etc.) by which time is to be scaled. In the above example, the actual durations of "score" are altered. If you want to preserve the old version of score as well as create a new scaled version, you can use the following construct:

tscale <oldscore> <factor> [":" newscore]

Where "oldscore" is the original version of the score (which is preserved), and "newscore" is the name for the new (scaled) score to be created.

A particular feature of "tscale" is that it allows you to scale note durations independently of score duration (and vice versa). For example, it enables you to change articulation (staccato or legato), without affecting tempo (i.e. by scaling note durations only). Usage in this case is:

tscale [flag] <score> <factor>

or

tscale [flag] <oldscore> <factor> [":" newscore]

Where "(old/new)score" and "factor" are as above, but we have a new argument "flag". The "flag" is simply one of the two following arguments: "-s", meaning that you want to affect only score duration, and "-n", meaning you want to affect only note durations. Example usages are:

tscale duet 4.

tscale 4 duet

tscale -s duet 4

tscale duet stacduet -n .5

CHAPTER 3

tscale duet 6.1 -n

With regard to scores, *tscale* can be used to change tempo; however, remember that we are modifying the time scale of the score and therefore the "entry delay" between notes; that is, the period of time between the start of one note, and the start of the following one. Thus, to double the tempo of a score (twice as fast = half as long), you scale score duration by a half.

Finally, note that the scaling factor need not be a positive number. A scaling factor of -1, for example, has the same effect as the command *retro*.

3.3 Performance

3.3.1 *conduct* is a program which enables the user to "interpret," or "conduct," scores which have been previously composed. Usage is:

```
conduct
```

You will then be asked to identify where you are working (upstairs or downstairs). Your response ensures that the proper transducers are connected to the machine. Once this is done, the system will prompt you to switch to the conducting terminal, or "work station". The first question which is asked at the *conduct* terminal is

```
Sym Cnt:
```

Normally, you can ignore this question and just type RETURN. (If, however, you are having troubles making your score fit into the system, see the documentation for the command *cksize* to find an alternative way to respond.) The conducting station then prompts you to type in the name of any scores to be conducted. When all scores to be conducted have been specified, push the "RETURN" button on the terminal, without typing anything else on the line. You will then be able to begin conducting. Appendix E of this document gives a detailed description of the *conduct* system.

Also see the command *makeff* to speed-up the specification of scores to the *conduct* system.

NOTE: periodically the screen cursor will not track the tablet puck. In such cases try the following. First, push the button on the terminal keyboard labelled "HOME". This will usually do the trick. Failing that, push the little button on the underside of the upper right hand side of the tablet.

3.3.2 *lsisplay* is a program for playing scores. It has exactly the same usage as *play*. The difference between the two programs is that *lsisplay* performs the score completely on the a private computer. As a result, it takes about 30 seconds longer from the time that the command is typed until the music is hear. The benefit is that the timing is unaffected by other users on the system, as is often the case with *play*.

3.3.3 *play* is used to play scores. The main argument which must be specified is the name of the score to be played. In this case, the usage is:

```
play <scorename>
```

Where <scorename> is simply the name of a previously defined score. If more than one score name is specified, the scores are spliced together and played sequentially in the order given. An example of such usage would be:

```
play start middle last
```

where "start", "middle", and "last" are three previously composed scores.

MUSIC COMMANDS

There are several optional arguments which may be specified to *play*. One allows control over the tempo of the performance, another automatically starts the audio tape-recorder in order to record the performance, another makes sure that all waveforms needed by the score are loaded into the synthesizer, and the last allows the scores to be "mixed" before playing. These optional arguments may be used together or alone.

The optional tempo argument is expressed as a metronome marking, just as in regular music notation. It is written as a positive whole number, such as "60" or "120", which indicates the number of quarter notes per minute. An example of the use of the tempo option is as follows:

```
play 80 concerto
```

When left unspecified, the tempo assumed is "60". When more than one score is being played, a different tempo argument may be specified for each. The rule is that a metronome marking remains in effect until a new one is encountered (just as in regular musical practice). An example of this type of usage would be:

```
play normal 20 slowpart1 slowpart2 200 fastpart
```

where the score "normal" would be played at mm 60, "slowpart1" and "slowpart2" at mm 20, and "fastpart" at mm 200.

When two or more scores are given to *play*, they need not be spliced and played in sequence. If a sequence of score names appears enclosed in curly brackets ("{" and "}") , then those scores will be *mixed* together, all starting at a common point in time. This is illustrated in the following example:

```
play 120 {soprano alto tenor bass}
```

As a result, scores can be composed in layers but auditioned together. In addition, a combination of mixing and splicing can be specified for the scores listed. This is shown below:

```
play intro {sop alt ten bass} coda
```

In this case, the mixed scores will follow the score "Intro", and "coda" will follow the longest of the mixed scores. Note that the effect of the mixing and splicing is *not* permanent, and no new score is created. This you must do using *mix*, *splice*, or *sced*, for example.

The waveforms required by your score are not always in the synthesizer. You can always put them in using the command *wfload*, but *play* provides a more convenient way. If an additional argument '-w' is specified, then all required waveforms are loaded before the performance begins. An example of this usage is:

```
play -w 120 mywave
```

In order to have the performance recorded, a special argument, "-r", is specified, as in the following example:

```
play -r concerto
```

See also the command *lsiplay*.

CHAPTER 3

3.4 Functions

3.4.1 Definition

3.4.1.1 `funced` enables the user to edit (define, delete, modify, etc.) stored functions.
Usage:

`funced`

Note: `funced` is an example of a command for which there are two versions: one using graphics, the other conventional alpha-numeric techniques. Therefore, if you are logged on at a graphics terminal you will get the former; otherwise, the latter. If you want to over-ride this feature (i.e. use the alpha-numeric version on the graphics terminal, or *vice versa*), prefix the command with "t" (typed) or "g" (graphics), respectively.

In using `tfunced`, the range of a function (i.e. the domain of Y values) can be selected to suit the convenience of the user, depending upon the function's intended use. The domains (modes) currently available and their respective possible Y values are:

MODE	CODE	RANGE
normal	n	0. to +1.
bipolar	b	-1. to +1.
chromatic	c	-12. to +12. (semitones)
volume	v	0. to 255.

The current mode is always indicated by the prompt character ('n' for normal, for example). To change the mode, enter the appropriate code (which is the first letter of the desired mode). Regardless of the mode that a function was originally defined in, it can be listed in any mode by changing into that mode before typing 'p' for 'print'. As an example, the following sequence of commands would define a function that would play a musical event a perfect fifth (7 semitones) above its original value:

```
Type command ('h' for help)
n: c
c: a
Type number of segments: 1
Type initial Y value (range: -12. to 12.): +7
seg 1:
    rel dur: 1
    Y Value: from 7.00 to: 7
c: w newfunc
c: b
b: Function name: newfunc
    Number of Segments: 1
    Segment data:
        1. Rel. Dur: 1.00 Y Value: from 0.58 to 0.58
b: q
Type x to exit
%
```

3.4.2 Auxiliary Commands

3.4.2.1 `fnv` is a command to invert, or flip a function "upside-down": It is of use, for example, in cases where you want the loudness of one object to fall when that of another rises, and *vice versa*. Usage is:

```
fnv <function> [":" newfunction]
```

where "function" is the name of the function to be inverted, and "newfunction" is the optional name of the inverted result.

3.4.2.2 fretro is a command to cause a function to be reversed in time. It is one way, for example, to generate envelopes which will give the same type of effect as playing an audio tape backwards. Usage is:

```
fretro <function> [":" newfunction]
```

where "function" is the name of the function to be reversed, and "newfunction" is the optional name of the reversed function.

3.4.2.3 pfunc is a command to list the data of a function on an ordinary terminal. The command permits the function to be printed in units which are appropriate to the user's needs. To do so, an optional flag can be specified which indicates the mode in which the function is to be listed. Usage:

```
pfunc <function name> [mode]
```

The legal mode arguments are as follows:

FLAG	MEANING
-n	normal (0 to 1)
-b	bipolar (-1 to 1)
-c	chromatic (semitones)
-v	volume units (0 - 255)

When left unspecified, functions values are expressed within the interval 0 to 1. Note that the mode flag does not cause any change in the function itself, just how it is notated. As an example, the following would list the function "gliss" in terms of pitch change by semitone.

```
pfunc gliss -c
```

3.5 Waveforms

3.5.1 Definition

3.5.1.1 fconv: The normal way to define a waveform is by defining its spectrum; or by drawing its waveshape, for which the programs *waved*, *wavesum*, and *objed* are provided. The *fconv* command provides an alternate facility of drawing waveforms -- albeit in a round-about manner. What the command does is convert a *function* file into a *waveform* file having the same shape. Thus, the waveform is drawn using the function editor (see the command *funced*), and then converted into a waveform using *fconv*. The resulting waveform can then be loaded into the synthesizer using the command *wfload*. Usage:

```
fconv <function name> [":" wavename]
```

If a name for the newly created waveform is not given, the waveform assumes the name of the function, and the original function is lost.

3.5.1.2 waved is an editor to enable the definition of waveforms. Waveforms may be specified by interactively defining their spectrum, or by drawing their waveshape. Figure 10 illustrates the definition of a waveform by specifying the relative amplitudes of

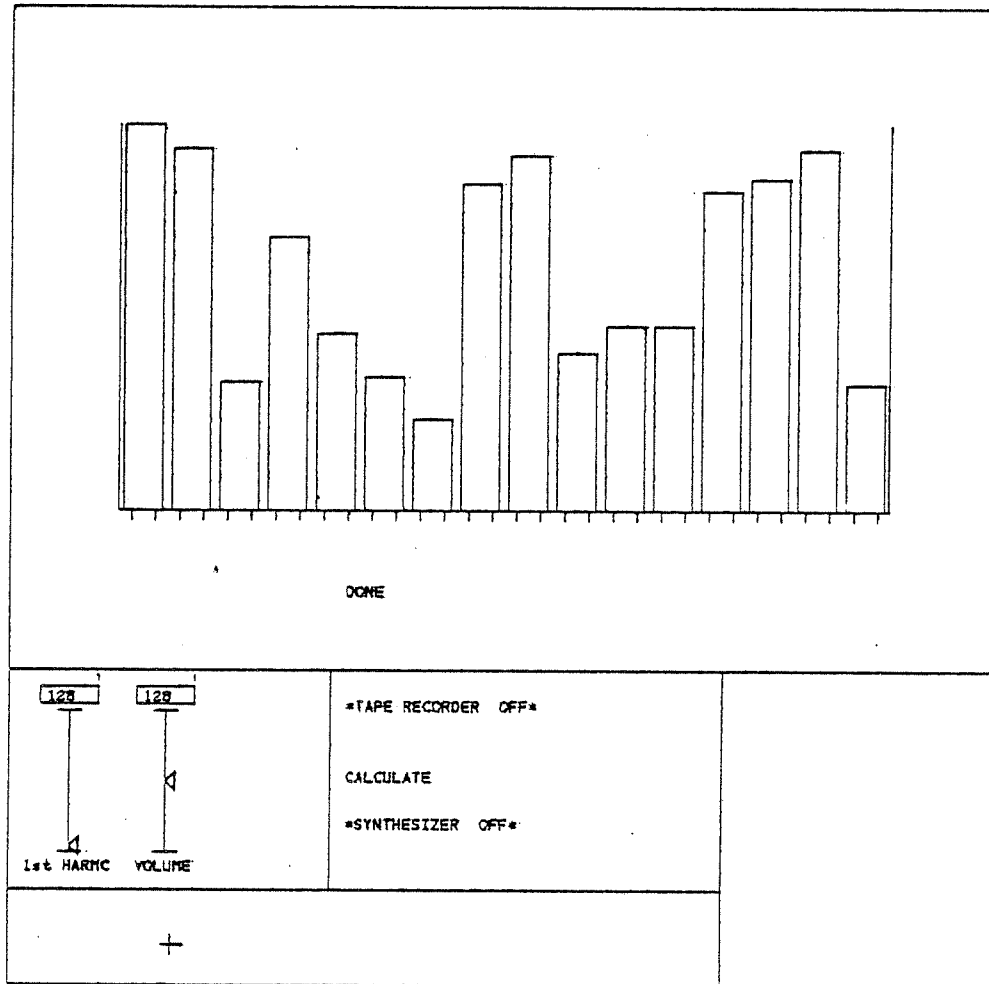


Figure 10. Defining a Waveform by its Spectral Content

its first 16 partials. The waveform defined may be auditioned in `waved`, loaded into the synthesizer, and saved for future use. Usage:

`waved`

3.5.1.3 `wavemix`: is a command which enables a waveform to be created by mixing together two or more previously defined waveforms. The user has the option of hearing the results during mixing. The command's arguments are the names of the waveforms to be mixed. For each, a number can be specified which gives the "weight" of that wave in the mix. Any positive number can be used. If no number is given, one (1) is assumed. The weight, if specified, must follow the waveform name. Usage is:

`wavemix <wavename [weight]>... <":" newname> [-1]`

If the '-1' flag is included, mixing occurs interactively using the synthesizer. For example, the following would mix the waveforms "fred" and "matilda" in a 2:1 balance:

`wavemix fred 2 matilda : together`

The weight of matilda is implicitly one, and the result will be saved under the name

"together". If the new waveform name is omitted, a prompt is given.

Wavemix can be called with no arguments in the command line. In such cases, the waves to be mixed and their weights can be specified interactively. Changes can be made and the composite waveform auditioned before it is saved. In this mode, wave mixing proceeds in a manner similar to *ed* and *sced*. Typing 'a' enters 'append' mode, where new waves can be added or the weights of existing waves changed. All user commands are prompted for interactively. Other commands allow changing the frequency of the test note and getting a listing of current waves and their weights.

3.5.1.4 *wavesum* is a command to generate waveforms by algebraically summing a set of weighted partials or harmonics. This is to say that the user may create a new waveform by specifying the relative amplitude of each harmonic. The program may be used in two different modes: (1) in the manner of most of the other music commands, with all the information necessary for the synthesis of the new waveform given in the same line as the command, or (2) in an interactive mode. The latter means that the user will give information in response to prompts given by the program. The two modes may also be mixed, where the user may give some information in the command line, then give the remaining information interactively. An example of the first type of usage is:

```
wavesum 1 .5 .4 .6 .1 : strange
```

which results in a new waveform "strange" that was the result of the summing of 5 partials. The relative amplitudes of the partials are given in order of ascending partial number as decimal numbers in the range of 0 to 1. In the above example, partial 1 has a relative amplitude of 1, partial 2 one of .5, partial 3 one of .4, partial 4 one of .6 and partial 5 one of .1. If the new waveform is not explicitly named through the colon (:) argument, it is saved under the name "wave.out". If a file named "wave.out" was already in existence, it is lost and the new waveform takes its place.

The same result as the above example could be obtained in interactive mode. The program's prompts and the user responses are given below just as they would have appeared on the terminal's screen. (In the example, anything typed after a "%" or ":" character is typed by the user, everything else is output from the computer.)

```
% wavesum
Type waveform name: strange

Enter highest partial number: 5

h for help
Partial 1
                rel. amp.: 1
Partial 2
                rel. amp.: .5
Partial 3
                rel. amp.: .4
Partial 4
                rel. amp.: .6
Partial 5
                rel. amp.: .1
%
```

The final "%" is the signal that the program has finished and that the computer is again ready to accept commands. An advantage of the interactive mode of input is the "h" or help command, which gives the user an idea of what to do. Typing an 'h' instead of a

CHAPTER 3

relative amplitude value elicits the following response from the program:

```
rel. amp.: h
```

```
Relative amplitudes are values from 0. to 1.  
Use 0 to indicate absence of a particular partial
```

after which the program again prompts for the relative amplitude of the current partial. As is indicated by the help response, zero is used to allow for the absence of a particular partial. This is necessary, as the program expects the relative amplitudes of *all* partials to be given explicitly. When in interactive mode, if a relative amplitude value outside the legal range is given, the program ignores it and prompts for a legal value.

The waveform used in the above examples is the sine wave; however, the user may specify an alternative "source" waveform in the command line (including one previously defined by *wavesum!*). Usage is then as follows:

```
wavesum [source wave] [rel amp [rel amp]] [: target]
```

where [source wave] is an optional waveform to be used instead of the sine wave. The source wave may be any of: the standard waveforms (those loaded by default by *wfload*), the current waveforms (those currently in the synthesizer buffers) or any waveform file in the user's directory.

In the above examples, it is assumed that all the partials start in phase, that is, each component waveform is assumed to have begun at the same point in time as any other. It is possible to "stagger" the waveforms by specifying their starting *phase* as well as their relative amplitudes. By phase we mean the point along the full cycle of a given partial at which that partial is starting. By convention, a phase is given in terms of degrees, with 360 degrees being one full cycle of the wave, 180 degrees being half of a cycle and so on. Phase is specified in the command line by following the relative amplitude with the phase, as an integer number from 0 to 360, followed immediately by a comma (with no space between the phase and the comma). The phase is optional and thus may be omitted for any partial (the default phase being zero degrees). The following example illustrates this type of usage, with partials 1 and 4 starting 90 degrees out of phase.

```
wavesum .5 90, .8 .6 .3 90, : staggered
```

Note that a comma follows the phase of partial 4 even though it is the last partial. The comma indicates to the program that the preceding number is a phase and does *not* act as a separator of arguments or as punctuation. In order to have the program interactively prompt for the phase of each partial, the *-p* flag must be given in the command line. Thus if the user desired to have the program prompt for complete information about the waveform (target file name, number of partials, relative amplitudes and phases) he would give the command as follows:

```
wavesum -p
```

The program would then prompt for all of the above information as in the following example:

```
% wavesum -p:  
Type waveform name: single  
  
Enter highest partial number: 1
```

h for help

Partial 1

rel. amp.: 1

phase: 30

%

The help command may also be given in response to a prompt for the phase of a partial, in which case the program would respond by giving the expected value for a phase.

By way of conclusion, the user is encouraged to first experiment with the waveform synthesis facilities of *objed* and only then attempt to make serious use of *wavesum*. Each of the two programs have their advantages and disadvantages: *wavesum* may be used at any terminal, while *objed* is only available at the Graphic Wonder terminal and does not allow phase specification. However, *wavesum* lacks *objed*'s ability for real-time playing of the waveform being synthesized.

3.5.2 Auxiliary Commands:

3.5.2.1 pwave is a command which enables the user to have the numerical data of any waveform listed on his terminal. Usage is as follows:

pwave <wavename>

3.5.2.2 wflist is a simple command which enables the user to list the names of all waveforms currently loaded in the synthesizer. Usage is:

wflist

The waveform names are listed on the user's terminal.

3.5.2.3 wload is a command to enable waveforms to be loaded into the SSSP synthesizer. The waveforms loaded may either be user specified, or come from the standard library. In the former case, the user must specify the name of the waveform and the buffer (number 1 - 8) in which it should be placed. The waveform previously held in that buffer is overwritten. In this case, use of the command can be summarized as follows:

wload <buffer#> <waveform name>

The second use of the command is to restore the "official" or "library" waveforms to the synthesizer. This would be done in order to restore things to normal when many waveforms have been changed (using *waved*, for, example). Another reason to use *wload* in this way would be in the event that the synthesizer has been powered down, thereby causing all the waveforms to be lost. (This case is easily detected by everything you play sounding like noise - although with some of the music people write ...) Usage in the "restoration" case is simply:

wload

While the above should be an adequate description for most users, *wload* is a more powerful command than has been indicated. The following, therefore, goes into more detail (detail which may be superfluous for most users).. To begin with, the formal definition of the command's usage is as follows:

wload [waveform source] [1] [2] [3].... [8]

The waveform(s) to be loaded can originate from three possible sources. The desired source is indicated by the user *via* the *waveform source* argument. The three forms of

CHAPTER 3

this argument are:

"-s" which indicates that the waveform(s) to be loaded originate from the standard library. This is the default mode, and the source when the *waveform source* argument is omitted. This mode is particularly useful in (re)initializing the synthesizer to the "normal" state. The optional numeric arguments (1-8) are provided to enable the user to selectively specify *which* of the standard waveforms are to be loaded. Thus, one can reinitialize some of the waveforms, while leaving others in their current state; however, the default case (no numeric arguments provided) results in *all* waveforms being loaded.

"-c" which indicates that the waveform(s) to be loaded are from among the "current" waveforms; that is, those waveforms which are supposedly already loaded in the synthesizer's buffers. The value of this option is in restoring waveforms when the synthesizer's waveform buffers have been inadvertently erased (due to the device being powered down, for example). As with the "-s" option, waves can be selectively loaded through the use of the numeric arguments. Omitting the numeric arguments causes all current waveforms to be loaded.

user defined waveform: in this case, the source argument is the name of a valid *waveform* file in a user's current directory (such as one created using the command *waved*). In this case, the named waveform will be loaded into the synthesizer buffer indicated by the *first* numeric argument (1-8). In this mode, this argument *must* be provided, in order that the command know the desired destination of the waveform. Any subsequent numeric arguments will be ignored.

3.5.2.4 *wfsave* enables the user to save a personal copy of any of the waveforms currently loaded in the synthesizer's memory. This is of particular use in cases where a previous user has left a useful waveform in one of the synthesizer's buffers. In order to determine what waveforms are loaded in what buffers, the user need only use the command *wflist* or the object editor (*objed*). Usage of the command is as follows:

```
wfsave <buffer#>
```

3.6 Miscellaneous Music Commands

3.6.0.1 *cleanup* is a command which should be ignored by novice users. It is a utility to cleanup files in a directory which are not used by a particular score. The program would be used when a directory is intended to hold the files utilized by one score, and one score only. Typically, in the course of composition files accumulate which in the end are not used. For house-keeping purposes, we want to get rid of the unused files, but not those needed. To do so, call the command as follows:

```
cleanup
```

It will then ask for the name of the score. Once given, the program reads into memory all files used by that score. It then prompts you to delete (using "rm") all files in the directory. Once done, those used by the score are written out fresh.

3.6.1 *pitch* is a simple command to give the *pitch* of a given *frequency* or the *frequency* of a given *pitch*. Usage:

```
pitch <pitch or frequency>
```

With the command, frequency is specified in c.p.s. (hz), and pitch as "a4", "c5#", etc.

3.5.2 `swit` is a utility not generally required by the average user. It allows the user to specify that a particular *work station*, -- that is a slider box, terminal, tablet, clavier combination -- be connected (or "switched") to the LSI/11. The program is called while working on the 11/45, before moving to a work station to work with an LSI-based program (such as *conduct*). This function is normally built into most SSSP programs, so need seldom be used by the user. Usage is:

```
swit <location>
```

where "location" indicates the desired work station. Currently valid arguments are "u" for upstairs (the main lab), and "d" for downstairs (room 105).

3.5.3 `radio` is used to control the radio (FM, of course) that is hooked up to the distribution network. This program takes one argument which may be the volume (between 0 and 63), "on" which turns the radio on to the default volume of 32, and "off" which turns the radio off. The argument may also specify what station you want to listen to. Stations currently available are: cbc, cjrt, chum, and q107. Examples of usage are:

```
radio on
```

```
radio off
```

```
radio cbc
```

```
radio 32
```

4. UNIX COMMANDS

4.1 General

In the SSSP system, *objects*, *scores*, *functions*, and *waveforms* are all stored as what we call *files*. When you are working on the system, all the files which you create are placed in your personal *directory*, which can be thought of as a drawer in a filing cabinet. Many of the commands described briefly below enable you to do useful things with the files (scores, objects, etc.) in your directory. These include commands such as copy (`cp`), remove (`rm`).

There are a few general comments which can be made with respect to the commands given below. First, UNIX allows you to abbreviate file names. To do so, the character '*' is used. It matches any sequence of characters. Therefore, if the command

```
rm <filename>
```

causes a file to be deleted, then

```
rm *
```

means remove all files, since '*' matches all file names. On the other hand, 'n*e' would have deleted all files with names beginning with 'n' and ending with 'e'. Another point is that the character '^' can be used as a synonym for '|'. Finally, it is possible to keep material for one composition in a separate directory from that of another. Users may make their own directories, and move files from one to another. Understandable information on all of these UNIX related operations is available in the document *UNIX for Beginners* (Kernighan, 1975a). (Copies are available in the CSRG main office.)

CHAPTER 4

4.2 Commands

4.2.1 `cp` is a function to create a new file which is a copy of an old one. The original version is unaffected. Usage:

```
cp <oldfile> <newfile>
```

4.2.2 `file` is a useful command which enables you to find out the type (e.g. score, object, function, etc.) of any file in your directory. Usage is as follows:

```
file <filename>
```

Note that you may ask for the type of more than one file at a time, as is shown in the following example:

```
file fred joe charles
```

If you want to get the type for all files in your directory, simply type as follows:

```
file *
```

4.2.3 `gwsnap` is a command which enables you to get a "snap-shot" or "plot" of the image currently on the graphic-wonder display. The copy is plotted on the *versatec* printer/plotter. Usage is:

```
gwsnap | opr -vp -t
```

Note: First, the order of the arguments is important for this command. Second, you normally would want to invoke this command from a terminal other than the graphic-wonder, so that the command line is not also reproduced. Finally, the process is very expensive in terms of system load and paper costs. The command, therefore, should be used with great reservation.

4.2.4 `ls` is a command to enable you to list all of the files in your directory. Usage:

```
ls
```

If you want the files listed in columns (so that they will all fit on the screen, for example), simply type as follows:

```
ls | c
```

4.2.5 `mail` is a command to enable you to send and receive "mail" to and from other users of the system. When mail has been sent, the receiver is notified that there is a message. It is also a handy means of sending reminders to yourself. Sending mail to the user "music" is the official way to report problems which are encountered in working with the system. Mail may be saved, and is stored in a special file in your home directory called *mbox*.

To send mail:

```
mail <username>  
your message (one or more lines)  
[ctrl d]
```

To read mail when prompted at logon time

```
mail
```

or

```
mail | p
```

To read old mail..

```
cat mbox
```

or

```
cat mbox | p
```

In the examples, including "| p" in the command stream means page-by-page output. To get the next page push the RETURN key. To abort the printing, hit RUBOUT.

4.2.6 **mv** is a command to enable the contents of a file to be "moved" from one file to another. Essentially, this constitutes re-naming a file. Note the difference from cp, since the old file is deleted. Usage:

```
mv <oldfile> <newfile>
```

4.2.7 **rm** is a command which enables you to "remove" (i.e. delete) a file from your directory. Usage:

```
rm <filename>
```

4.2.8 **who** is a command which enables you to discover who else is currently working on the computer. Just type the command and the names of all people currently logged on will appear on your terminal.

4.2.9 **write** is similar to *mail* in that it allows messages to be sent to other users. It differs in two ways. First, the receiver of the message must be logged on to the system at the time of writing. Second, the receiver may write to you in response so as to enable you to carry on a "telephone call" like dialogue. Instead of talking, you communicate by typing messages to one another. The messages are not saved. Usage is the same as mail.

5. REFERENCES

- Buxton, W. (1977). A Composer's Introduction to Computer Music. *Interface* 6: 57-72.
- (1978). Design Issues in the Foundation of a Computer-Based Tool for Music Composition. *Technical Report CSRG-97*. Toronto: University of Toronto.
- Buxton, W. & Fedorkow, G. (1978). The Structured Sound Synthesis Project (SSSP): an introduction. *Technical Report CSRG-92*. Toronto: University of Toronto.
- Buxton, W., Fogels, E. A., Fedorkow, G., Sasaki, L., & Smith, K. C. (1978). An Introduction to the SSSP Digital Synthesizer. *Computer Music Journal* 2.1: 28-33.
- Buxton, W., Reeves, S., Patel, S., & O'Dell, T. (1979). *SSSP Programmer's Manual*. Unpublished manuscript, University of Toronto.
- Kernighan, B. W. (1975a). *UNIX for Beginners*. Unpublished manuscript, Bell Laboratories, Murray Hill, N. J.
- (1975b). *A Tutorial Introduction to the UNIX Text Editor*. Unpublished manuscript, Bell Laboratories, Murray Hill, N. J.
- Ritchie, D. & Thompson, K. (1974). The UNIX Time-Sharing System. *Communications of the ACM* 17: 365-375.

APPENDIX A - A Tutorial on Editing Objects

1. INTRODUCTION

One musically potent feature of the SSSP system enables composers to "mix" their own palette of timbres. Rather than orchestrating notes with pre-set instruments, the composer has the option (as opposed to obligation) of defining a personal lexicon of timbres. In the SSSP system, these user-defined timbres are called *objects*. Like a conventional instrument, each object is identified by a name (such as "flute", "gong", or "fred"), and is characterized by a distinctive timbre. Notes of different pitches, durations, and loudness may all be orchestrated by the same object. Nevertheless, each instance of a particular object will maintain its composer-defined characteristics.

We are accustomed to thinking of orchestral instruments as belonging to particular families, such as "brass", "woodwind", or "percussion". In each case, family membership is determined by the way in which sound is generated. With the SSSP system, a similar situation exists. Objects are characterized by family according to the sound synthesis technique which they employ. There are five synthesis techniques available, according to which objects can be defined: fixed waveform, frequency modulation (FM), vosim, waveshaping, and additive synthesis.

There are two main graphics-based programs for working with objects. The usage of each is very similar. The first is called *objed*, which is an acronym for "object editor". It allows objects of four of the five families to be created, compared, and modified. For pragmatic and historical (rather than musical) reasons, there is a separate program, *bank*, for working with objects that use additive synthesis. Both programs are discussed in this tutorial, starting with *objed*.

While the remainder of this document attempts to provide a self-teaching guide to these programs, it is -- of necessity -- incomplete. In many cases, experimentation and careful consideration will provide the answers to specific questions. In working through the tutorial, do all the suggested exercises. If there are still problems, an experienced user is your best source of information.

2. ON ENTERING OBJED

On initial entry, *objed* assumes the state shown in Figure 1. Notice how the display is divided into five regions. Each serves a particular function. The main region, which occupies the upper two thirds of the screen, is where the actual object data being edited is displayed. In the example, this is a simple fixed waveform object. Its components, all of which are graphically displayed, are: a waveform, a time varying function controlling pitch, and an envelope controlling the contour of the note's volume.

To the left, below the main region, is an area containing data pertaining to the pitch, volume, and duration at which the object being edited can be auditioned. Remember, however, that these values are not part of the object. They are simply conveniences for exploring its behavior in different contexts.

The central panel in the lower part of the screen contains various options which allow the user to change the state of the editing environment. Options available include changing the object type and changing the way in which the edited object can be auditioned.

The panel to the lower right is dedicated to the saving and retrieving of objects with a minimum of effort. Finally, the elongated region along the bottom left edge is a "window" which permits the composer to access the outside world without leaving *objed*.

To work with the program, the user need only remember a simple strategy which forms the basis for all interactions: when you want to change something, just point at the diagram or word which represents it on the display, and depress the selection ("Z")

APPENDIX A

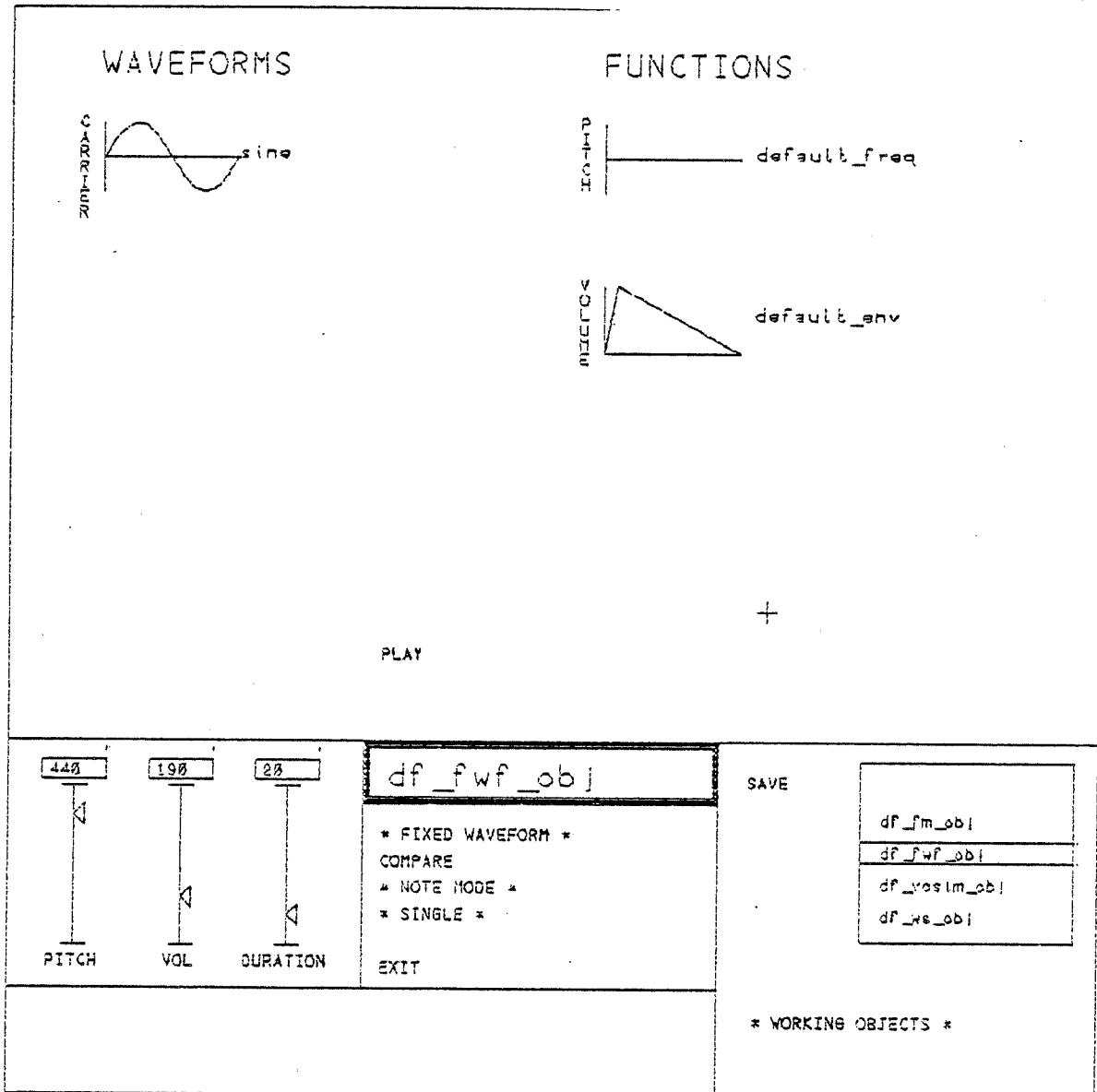


Figure 1. Objed on Initial Entry

button on the tablet's cursor. Any consequent options will then be presented, and the same method of interaction is applied to them.

We shall now cover the aspects of object editing, beginning with fixed waveform objects. This lets us begin with the simplest case, while building skills which can be used with more complex timbres.

3. AUDITIONING AN OBJECT

It is often useful to begin by auditioning the displayed object. This establishes a frame of reference for future changes. The object is played by activating the word (or *light button*) PLAY seen in the lower part of the work area. On so doing, the sound is immediately heard. Remember, though, that the object itself has no specific pitch, duration, or maximum amplitude associated with it. These must be provided externally. When PLAY was activated, the sound had a pitch of A4, dynamics of about *mf*,

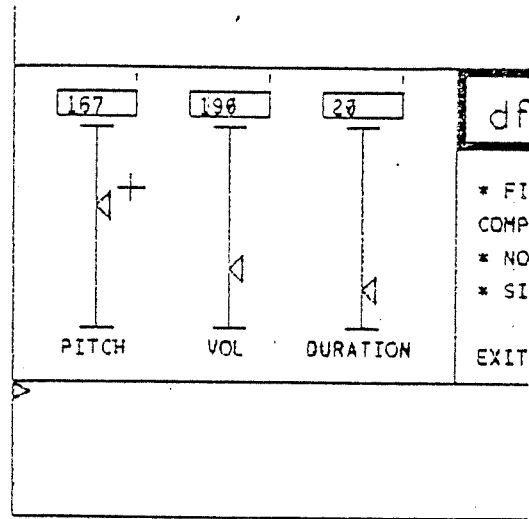


Figure 2. Graphical Pots for Performance Parameters

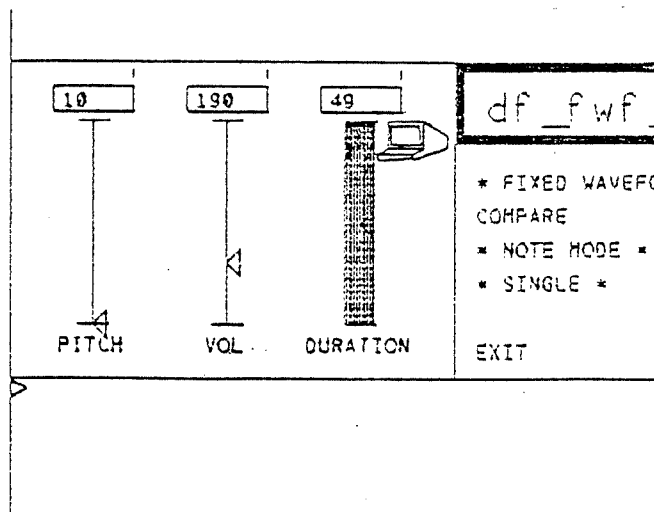


Figure 3. Setting a Graphical Pot by Typing

and a duration of 1/4 note: at mm. = 60. Obviously it is desirable to be able to change these values so as to learn more about the object's behavior. This is the function of the lower-left sub-panel seen in Figure 2. Here are seen three "graphical potentiometers", one for each performance parameter. The value for each parameter is displayed numerically in the boxes above, and its relative value is seen by the position of the potentiometer's "handle". The value of any pot can be changed by "dragging" its handle up or down using the cursor. Alternatively, the user may point at the box above the pot, activate the Z-button, and type in a numerical value. In this case, the tracking symbol becomes an icon of a terminal (as seen in Figure 3), as a prompt that something must be typed.

4. WAVEFORM SELECTION

Let us now consider changing one of the attributes of the object itself. As a first example, let us alter the waveform associated with the object. Pointing at the picture

APPENDIX A

of the current waveform and depressing the Z-button will cause the panel seen in Figure 4 to appear.

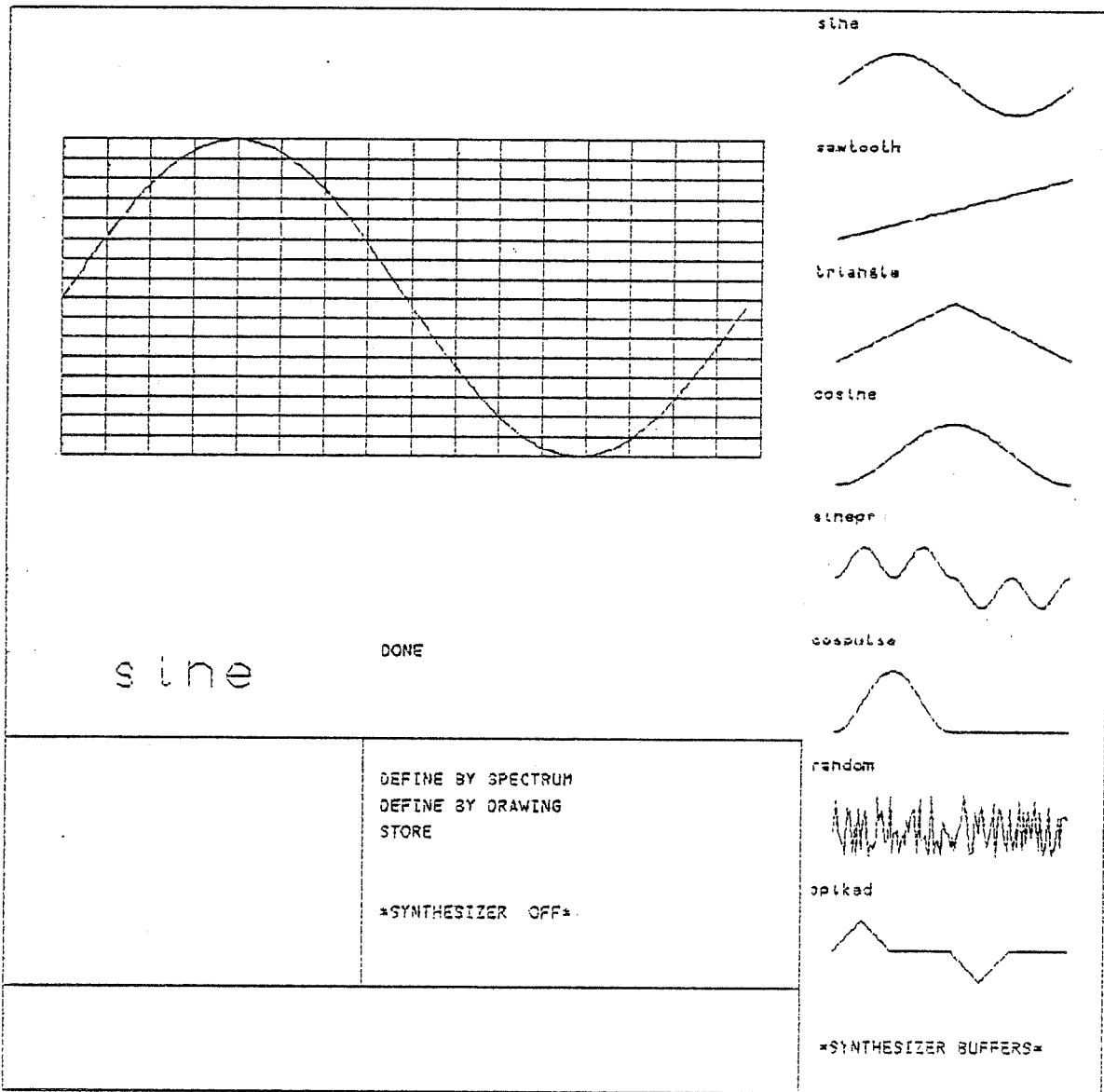


Figure 4. Waveform Selection

A menu consisting of the eight waveforms currently loaded in the synthesizer appears down the right margin of this panel.¹ Selecting one of these waveforms with the cursor will cause it to be copied into the grid in the main panel of the screen. We can then activate the light-button DONE which will return us to the original panel of Figure 1. The one difference will be that the selected waveform will be displayed in the place of the original one, "sine". This brings up an important point: the current state of the object being edited is *always* clearly displayed, thereby eliminating any mental burden of remembering its current attributes.

1. The waveforms shown in the figure are the "standard" ones. As we shall see, these can be altered by the user. If the previous user has left non-standard waveforms in the synthesizer and you wish to restore things to normal, use the command *wload* (Chapter 3.5 of the *Music Software User's Manual*).

EXERCISE 1: Enter *objed*. Play the default object at different pitches, volumes, and durations. Change these parameters by both "dragging" the potentiometers and by typing. Now select different waveforms and hear how the timbre is affected by the change. Be sure that you are comfortable using the cursor, and with switching back and forth between the main and waveform panels.

5. CREATING NEW WAVEFORMS BY SPECTRUM

If we want to define a new waveform, rather than use one of those in the synthesizer, we can do so by specifying the new waveform's spectrum. Again, we switch to the waveform panel. By activating the light-button **DEFINE BY SPECTRUM** (Figure 4), a third panel shown in Figure 5 appears.

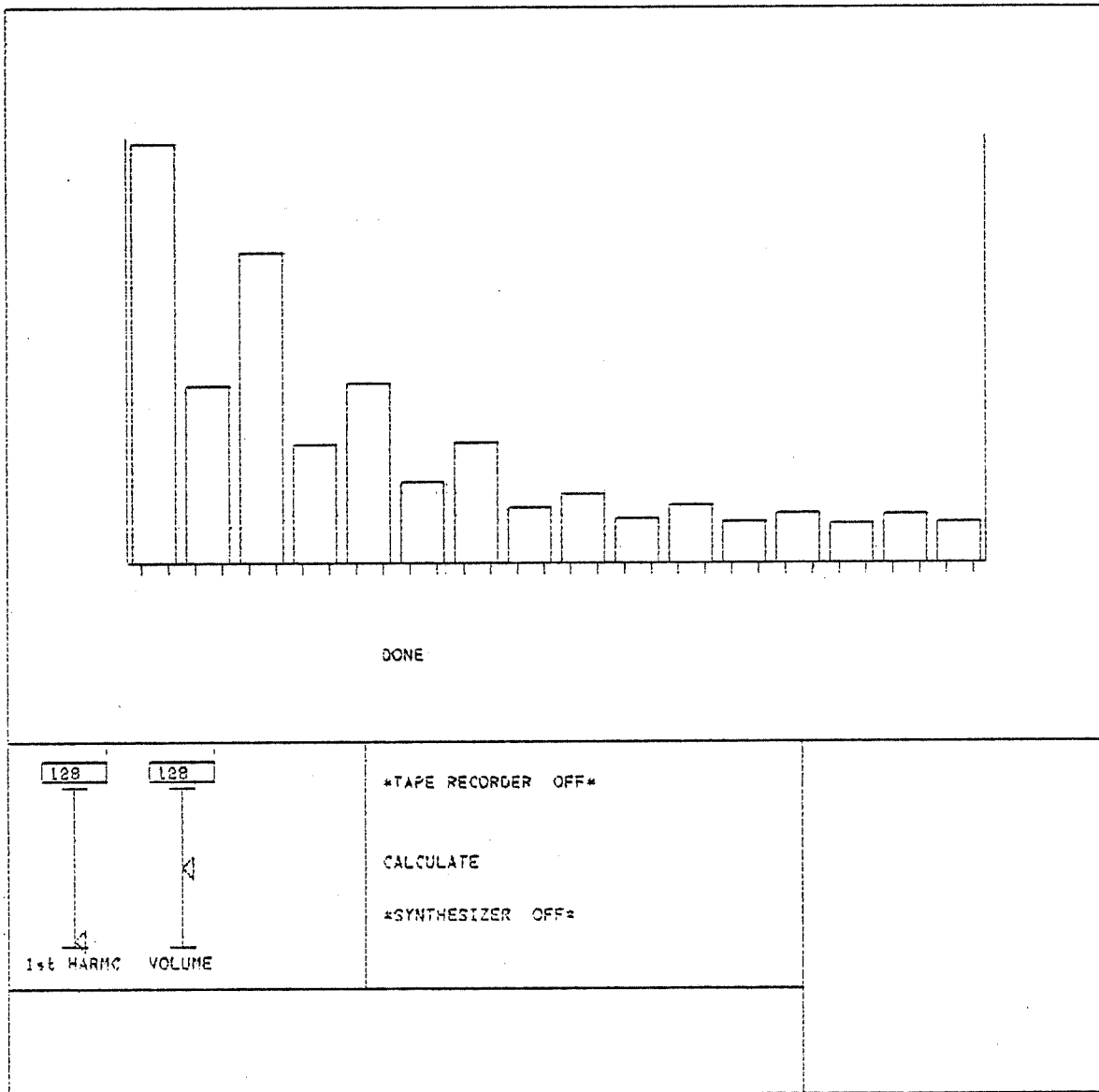


Figure 5. Defining a Waveform by Spectral Content .

What is seen in the work area is a bar-graph, where the height of the bars represent the relative amplitude of the harmonics of a sound. Harmonics one through sixteen appear

APPENDIX A

from left to right. Any bar will jump to the height of the cursor when it passes over the bar with the Z-button depressed. Thus, the amplitude of any harmonic can be dragged up or down, just like the graphic potentiometers already seen. The sound that results from these changes can be heard as they are being made. Activate the light-button * SYNTHESIZER OFF * and a 100 Hz steady state tone will be heard. The spectrum of this tone will be altered as you adjust harmonic levels. If you desire, you can also adjust the pitch and overall amplitude of this steady-state tone using the graphic potentiometers in the lower left corner of the display.

Once the sound has been adjusted to your satisfaction, the synthesizer can be turned off the same way that it was turned on. In so doing, notice that the button currently reads * SYNTHESIZER ON *; thereby indicating the current state of the system. We see then, that this light button functions like a switch, allowing the synthesizer to be turned on and off. This is an important point. It is a convention of *objed* that *all* light buttons enclosed in asterices (*) function as rotary switches. There may be more than 2 "positions", but repeated probing will always cause you to cycle back to where you started. We shall see much more of this later.

Once the waveform spectrum has been defined and the synthesizer turned off, activate the button labelled CALCULATE. This will then cause two choices, FOURIER and CHEBY-CHEV, to be presented. Activate FOURIER (ignore the other buttons for now), and wait. The system will calculate your new waveform. When done, it will be displayed on the grid. (The waveform seen in Figure 6, for example, results from the spectrum seen previously.)

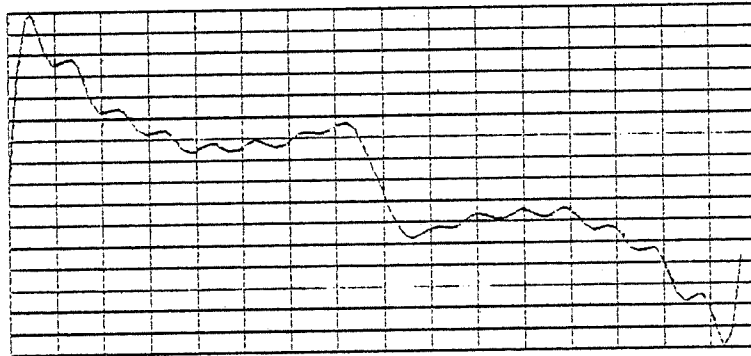


Figure 6. Waveform Defined by Spectral Content

You are finished now, so activate the DONE button, which will return you to the original waveform panel.

There are two final points to consider. First, since you have created a new waveform, you should give it a new name. (Notice that the new waveform still has the name of its predecessor.) The name is changed in the same way as everything else: point at it and depress the Z-button. You will then be prompted to type in a new name of your own choice.

The second point is that the new waveform must be loaded into the synthesizer. Only eight waveforms fit in the synthesizer at once. These are the eight waveforms shown on the right side of the menu. To put the new waveform in, one of the current ones must be overwritten. The program does not know which one you want to discard, so you must do this yourself. Activate the button STORE which is found in the lower centre panel. You are then prompted to indicate where the waveform is to go. Do this in the same way that you previously selected one of the waveforms in the right column. Having done so, the new waveform is loaded into the synthesizer and appears as one of the

eight waveforms displayed in the column. You are now finished, so activate DONE and return to the main panel.

EXERCISE 2: Define a new waveform by spectrum. Turn the synthesizer on and off. Become comfortable using smooth hand gestures to sketch the contour of the harmonics. Calculate, name, and save your waveform. Verify that it sounds the same when played from the main panel as it did when you defined it. Did you remember that the pitch had to be adjusted for a fair comparison?

6. FUNCTION DEFINITION

Change is one of the foundations of musical interest. In defining timbres, it is important to be able to specify how parameters such as amplitude, pitch and spectrum vary through the course of a sound. This is done by specifying a curve which defines how the parameter rises and falls through the sound's duration. Such a contour is called a *function*.

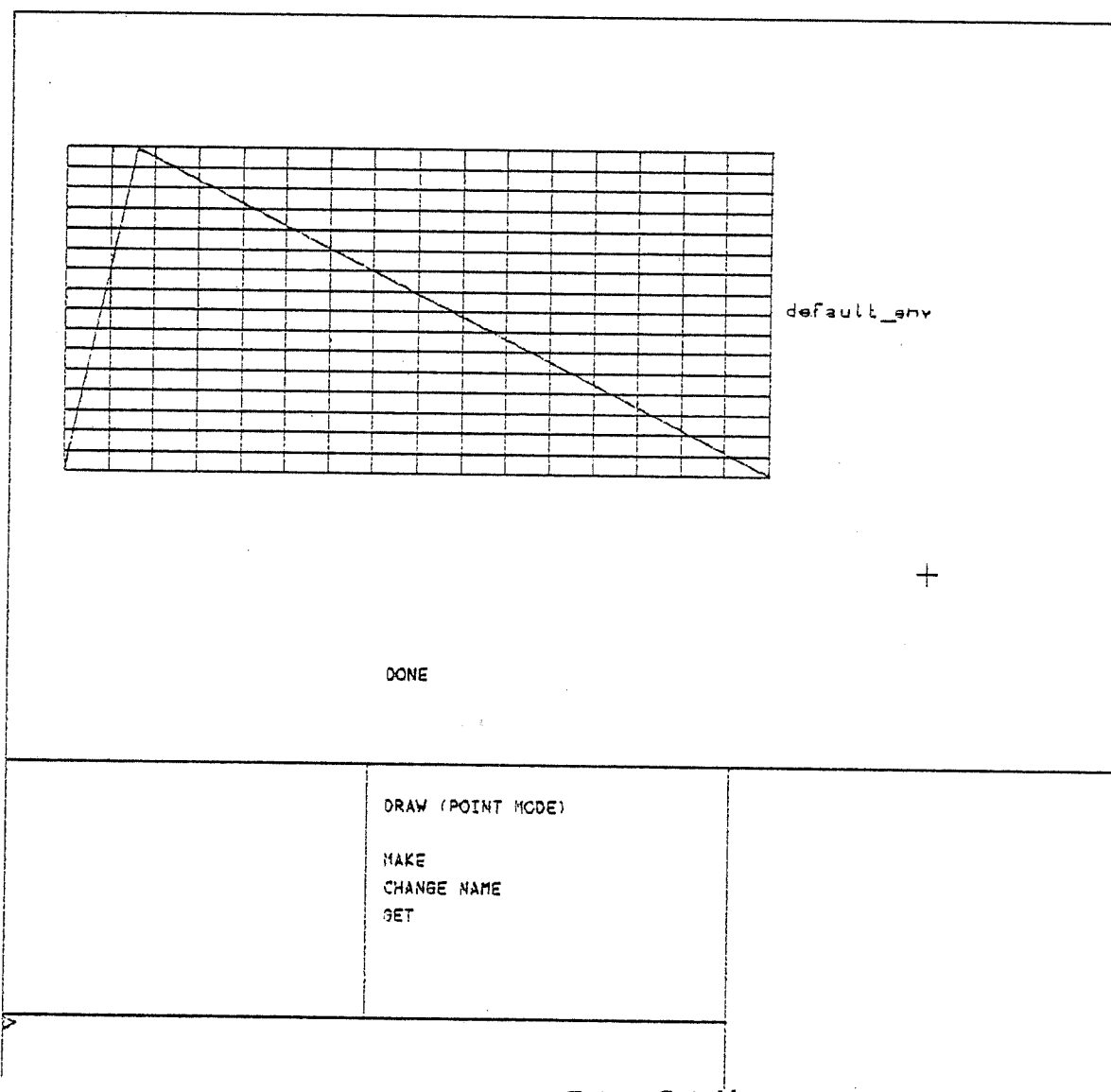


Figure 7. Function Editor Sub-Menu.

APPENDIX A

Fixed-waveform objects permit two such functions to be specified. One is used to control changes in pitch or portamento. The other controls the sound's amplitude "envelope". They are shown in the upper right side of the work area. The default functions assumed result in no change in pitch, and a simple attack-decay amplitude envelope. Each may be redefined with as complex a function as desired. Let us see how this can be accomplished.

Select the amplitude envelope with the cursor. The function editing sub-menu will then be displayed as seen in Figure 7. The function displayed in the work area is the one which is currently affecting the parameter that you selected (in this case the object's envelope). The function represents loudness (vertically) through time (horizontally). How long and how loud depends on the volume and duration of the note which is affected by the function. Think of the highest point of the grid as 100% of the note's amplitude, and the base of the grid as 0. Similarly, think of the left margin as the start of the note, and the right margin as the end.

Notice that the function has a name, "default_env". Now we have two choices at this point. Either we can just redefine the shape of this function, or we can make a completely new one. The difference is subtle but important. It is rooted in the role of names in the system. The first point to understand is that the same function can be used by many different objects. Among other things, this saves you from having to redefine the same envelope for several similar timbres. The second thing to recognize is that functions (all files for that matter) are referred to by name, *not* content. What is the result of this? If you change the shape of a function without changing its name, the change will carry through to every object which makes use of that function. That may be what you want. If it is not, however, what you should do is create a new function with a new name. The change will then be localized to the current object. If this is all rather foggy, don't be too concerned. The examples and exercises which follow will help to clarify things.

Let's assume that we want to create a new function. The way that this is done is to "clone" a new function from the current one. This is accomplished by activating the button MAKE (in the lower middle column), and typing in the new name in response to the prompt. You can now redefine the function's shape. Activate the DEFINE button. The tracker will become an icon of a quill, indicating that you are expected to draw. Draw the function from left-to-right, following the instructions appearing in the work area. You draw using what are called "rubber-band lines". The first thing you do is "anchor" the end of the rubber-band at the starting point of the function. Do this by positioning the cursor and depressing and releasing button-1. Then, depress *and hold down* the Z-button. The free end of the rubber-band will attach itself to the tracker, stretching from the starting point. Try this, moving your hand around the tablet till it feels relaxed. The objective is to stretch the rubber-band to the next point where you would like it anchored. Once there, release the Z-button. The line is anchored and you can proceed to define the next segment, again by depressing the Z-button and dragging the line from the last anchor point. Continue until the function is defined to the right margin of the grid. When done, depress button-3 to indicate that you are finished. The quill cursor will disappear, and your function will be redrawn as the computer understood it.²

Your function is now completely defined. If you are happy with it you may return to the main panel (activate DONE), and hear the result. Alternatively, you may redefine the function until you are satisfied with it.

2. The function should look the same when redrawn as it did when you drew it. If it did not, here are a few hints as to what might be wrong. First, you must draw left-to right. Second, you must start at the left margin and finish at the right one. Third, you should hit button-1 at the start only once. Fourth, the entire function should be drawn within the grid. If things didn't work the first time, try again. If things still don't work, ask someone for help.

You may redefine the function controlling variations of pitch through time in the same manner as just seen for amplitude envelopes. There is one important point to note, however. There is a difference in how the two are interpreted. The amplitude envelope describes variations between 0 and the maximum amplitude. The pitch function describes deviations from the specified pitch of the note. These may be in either direction, up or down. Thus, the horizontal line going through the *middle* of the grid is interpreted as meaning "no pitch deviation", the upper border represents deviation of one octave upwards, and the lower border one octave down from notated pitch. Everything else is linear in pitch so, for example, a quarter of the way up the grid implies a change of a tritone.

EXERCISE 3: Define a new envelope in the manner described and listen to its effect. Now draw the retrograde of the same envelope and hear the difference. Set the duration to be fairly long and then define an envelope that changes in steps, like a staircase. Do the same thing for the pitch function. Make the function have four steps: at the notated pitch, an octave lower, an octave higher, and back to the original pitch. Make a simple melody using a simple object. Try to synchronize changes in the two functions. Do not progress until you are quite comfortable changing and defining functions and waveforms. Remember that you only have to change the function's name when you want to make a new entity, as opposed to changing the shape of an existing one.

7. NAMING, SAVING AND RETRIEVING OBJECTS

Before an object can be used outside of the context of the editing environment, it must be named, and then saved. An object's current name appears in the heavily outlined box at the top of the environment control sub-panel (Figure 8).

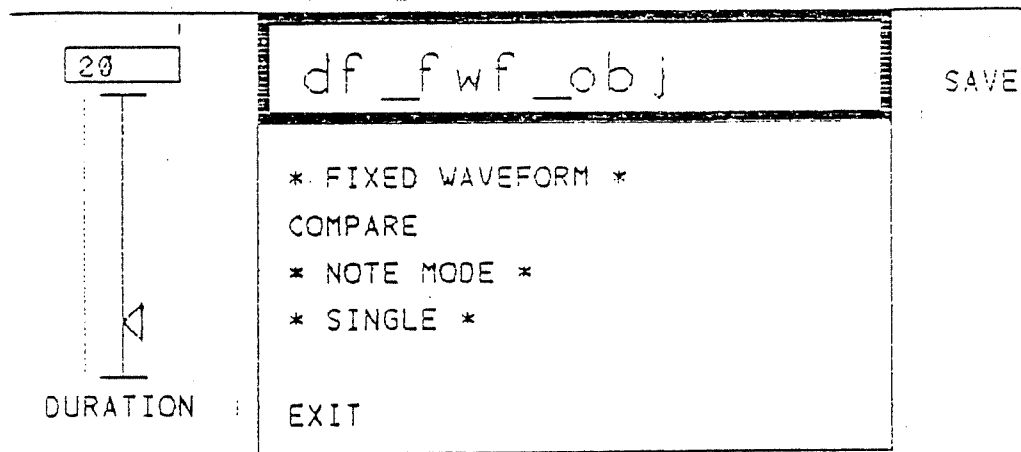


Figure 8. Editing Environment Control Sub-Panel

Names can be changed the same way as anything else: you point at it, activate the Z-button, and type in the new name in response to the prompt. Saving an object for future use is equally straightforward: the light-button SAVE seen in the bottom right-hand sub-panel is activated:

The same sub-panel also provides the mechanism for retrieving previously defined objects. The technique employed makes use of what we call a *directory window*. The directory window is the rectangle seen in the region. It functions like a window which has selective vision, allowing us to peer over the names of objects which are available

APPENDIX A

for retrieval. The names in the window are the names of these objects; and the name of the object which we are currently looking at in detail is the object whose name appears between the horizontal lines. This is the object being edited. A different object can be selected by pointing at its name and depressing the Z-button. This will cause that object to be displayed in the work area for purposes of editing or audition. Alternatively, slider 2 can be used to "scroll" through the list of object names. The object whose name appears between the horizontal lines at the moment when the slider is released is read in.

An important option in this process has to do with the domain over which the window looks. That is, the window will either let us view objects which have been defined and stored on disk or those which are in primary memory having been worked on in the current session. (The difference will be made clear in Exercise 4, below.) Thus, while only one object can be edited at a time, several "working" objects can be kept in primary memory during a work session. Nothing need be saved unless it is desired for a future session.

EXERCISE 4: Using what you have learned about envelopes and waveforms, define, name, and save three distinctive objects. Point to their names in the directory window and verify that they are retrieved and displayed in the work area. Switch from *WORKING OBJECTS* to *SAVED OBJECTS*. Notice that the only names now seen in the window are the ones which you saved. The additional objects which were seen in the *WORKING OBJECTS* window are system generated "default" objects and are not stored in your directory. (Only one of these default objects is of the type "Fixed Waveform", which accounts for previously unseen information being displayed if they are selected.)

If you do not save an object, it will be lost on termination of your work session. What if you save an object called "sax", for example, and then make additional changes to it. Do so. Then select one of your other three objects (just to get away from "sax" for a moment). Now, from among the *working objects* re-select "sax" and notice that it is the version most recently defined in the work session which is displayed. Now select "sax" from among the *saved* objects and notice that the most recently saved version is returned. Further experimentation will show that this "backup" version has become the current "working" version of "sax".

One final point worth noting is that all functions and waves associated with an object are saved with it. They need not be explicitly saved individually. Now is a good time to experiment with the effect of redefining the shape of a function which is used by more than one object.

3. FM OBJECTS

You should now be comfortable with the basics required to work creatively with *objed*. We can now begin to apply what has been learned to objects which are timbrally more complex and interesting. A different object definition mode can be set by activating the light-button FIXED WAVEFORM seen in the lower central panel (Figure 8). As a result, the panel will switch to present the set of object type options. Selecting "Frequency Modulation" will result in the display appearing as seen in Figure 9. Note that the panel is the same as that seen in Figure 1, except that there are five additional elements in the working area. There is now a waveform displayed for both the carrier and modulating waves. In addition, graphical potentiometers are provided to set both the maximum index of modulation and the c:m ratio. Finally, there is a third time-varying function which controls the evolution of the index of modulation.

It is not our purpose here to give a tutorial on FM synthesis. This can be obtained in a SSSP handout, and more detailed information is available from Chowning (1973). We can point out, however, that working with FM objects requires no techniques not already seen with Fixed Waveform objects.

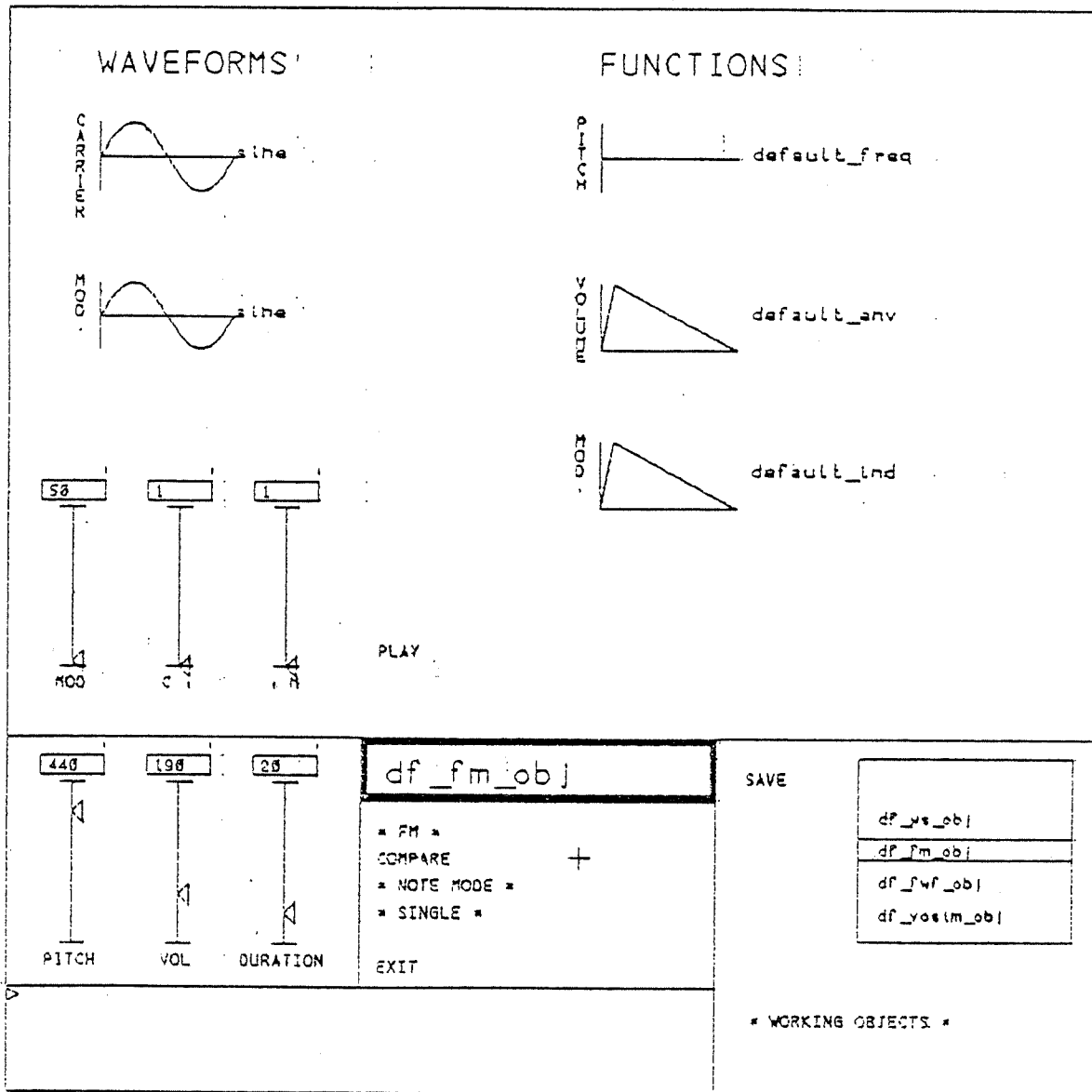


Figure 9. Editing an FM Object

9. MORE ON AUDITIONING MODES

During an editing session, it is often desirable to be able to audition the object repeatedly. One way to do this is to activate the * SINGLE * button at the bottom of the environment control panel (Figure 8). The button will be renamed CYCLE, indicating that when PLAY is activated the sound will play repeatedly until killed by the user.

The problem still exists, however, that all of our interactions with the object's data take the two-part form: change a value, listen to the effect. What would often be more useful, when setting the index of maximum modulation for example, would be to emulate the operation of an analogue synthesizer. That is, it would be useful to turn the sound on in its steady state, and hear the effect of adjusting parameters while they are being changed. (This is the same thing which we have seen when defining waveforms by spectrum.) This can be accomplished by activating the CYCLE button, which then switches to STEADY. Activating PLAY will cause the object to sound, and the effect of

APPENDIX A

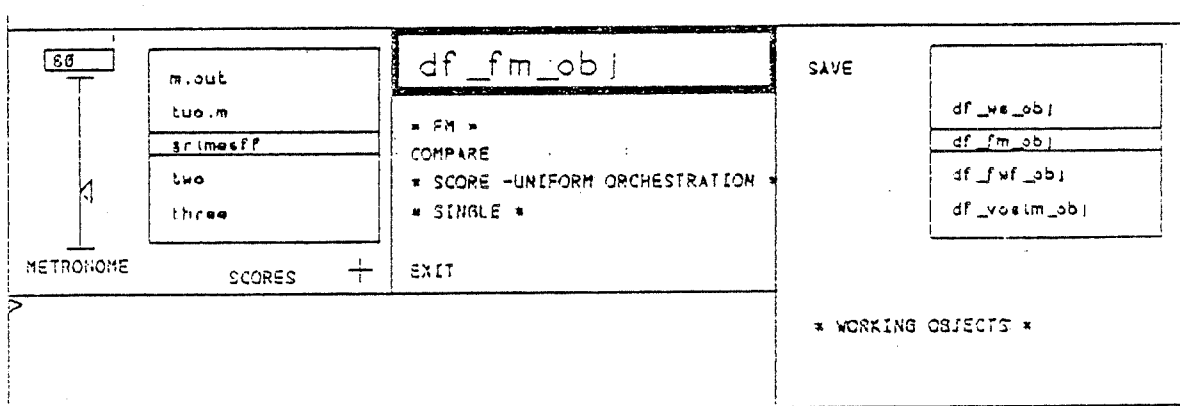


Figure 10. Auditioning Objects in the Context of a Score

adjusting any of the graphic potentiometers (including those affecting pitch and volume) will be heard immediately. When desired, we can return to the original SINGLE mode by activating the button * STEADY *.

In spite of the flexibility just seen in auditioning an object, there is still one fundamental problem. The effect of hearing an object in isolation is often (usually?) very different than hearing it in some musical context. So far, we have only been able to hear an object as a single note. This can be altered by activating the * NOTE MODE * switch (Figure 8). The result will be that the button is renamed SCORE - UNIFORM ORCHESTRATION, and the sub-panel controlling note parameters is switched, as seen in Figure 10. The box that appears is a "window" which looks out on all of the scores in the composer's directory. It is similar to that already seen in retrieving objects. The name of each score is listed in this window, and if there are more names than will fit, slider 1 can be used to "scroll" through them. The point to note is this: when the PLAY button is activated, the score whose name appears between the horizontal lines of the window is performed. More important to our purposes, the score is temporarily orchestrated with the object currently being edited! Furthermore, the metronome marking controlling the tempo of the performance can be controlled by adjusting the graphical potentiometer beside the window. Finally, we can "rotate" the mode switch one position further and change it to read SCORE - ORIGINAL ORCHESTRATION. In this case, when the score is played it retains its original orchestration. However, all notes orchestrated with the object whose name corresponds to that of the object being edited will be performed with the working copy of that object.

10. COMPARING OBJECTS

It is often useful to be able to audition two or more objects in rapid succession for purposes of comparison. There are problems in doing so with the techniques seen thus far. First, it takes a couple seconds for an object's data to be drawn on the display when it is retrieved. Second, the hand motion involved in travelling between the directory window and the PLAY button slows things down. To overcome these problems, another auditioning mode is provided in the editing environment region. This is the COMPARE button (Figure 8).

On activating COMPARE, almost everything on the display is erased. One main exception is the object directory window. In this mode, pointing at an object's name in the window will cause it to be heard immediately *without any graphical display of the data.*

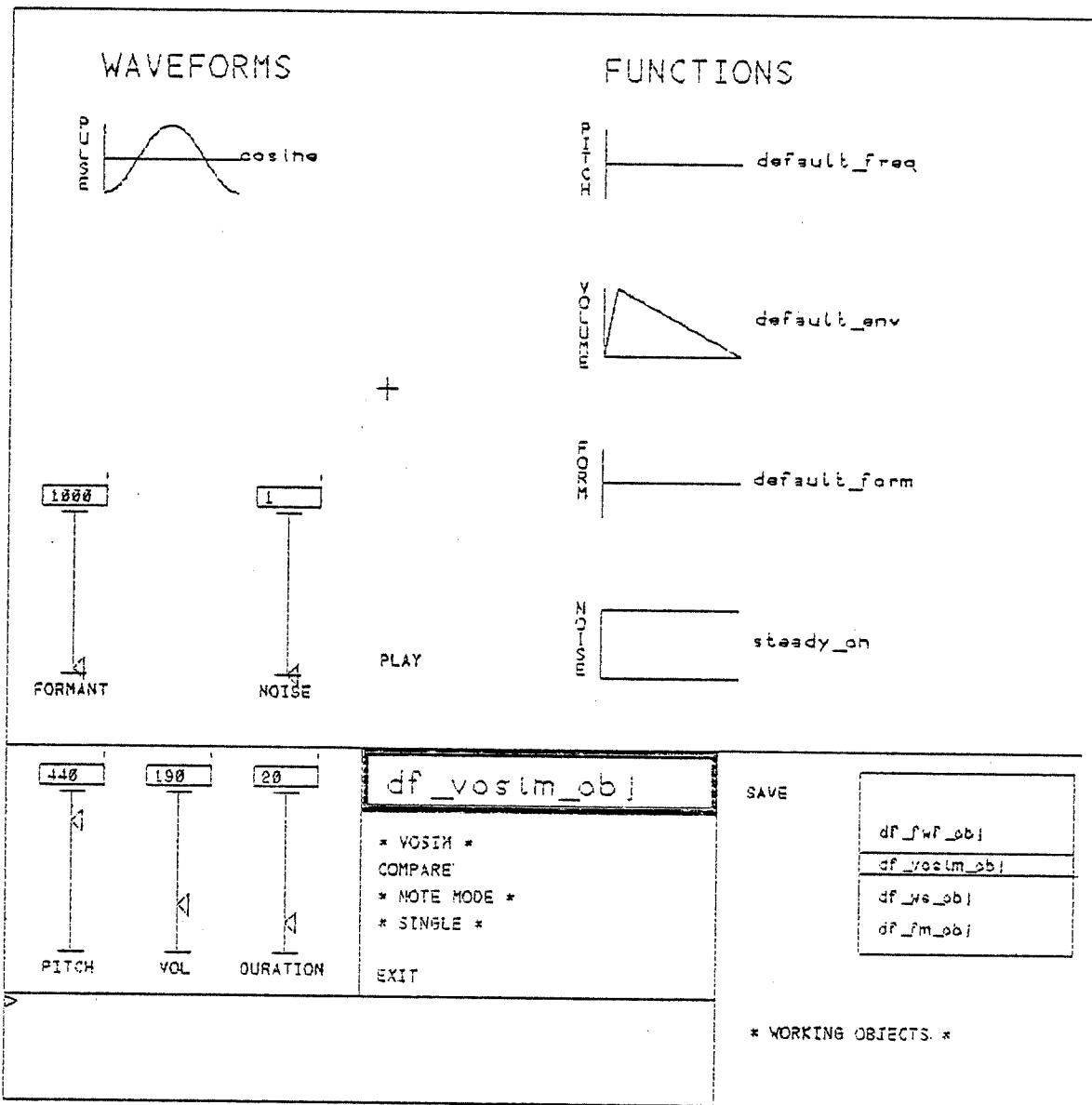


Figure 11. Editing Vosim Objects.

There are a few points to note about COMPARE. First, for fast comparison the objects in question must be in working memory. That is, they must have been read in to the program during the current session and, therefore, appear in the * WORKING OBJECTS * window. Second, when finished with COMPARE, the program is restored, displaying the data of the last object auditioned.

11. VOSIM AND WAVESHAPING

The mode can be set to edit vosim and waveshaping objects in the same way as FM was selected. Again, neither mode requires any working techniques not already encountered. The vosim panel is seen in Figure 11 and the waveshaping panel in Figure 12. Kaegi and Tempelaars (1978) is a source for information on vosim. Arfib (1978) and Le Brun (1978) are sources for detailed information on waveshaping. Roads (1979) gives a more readable tutorial on the subject.

APPENDIX A

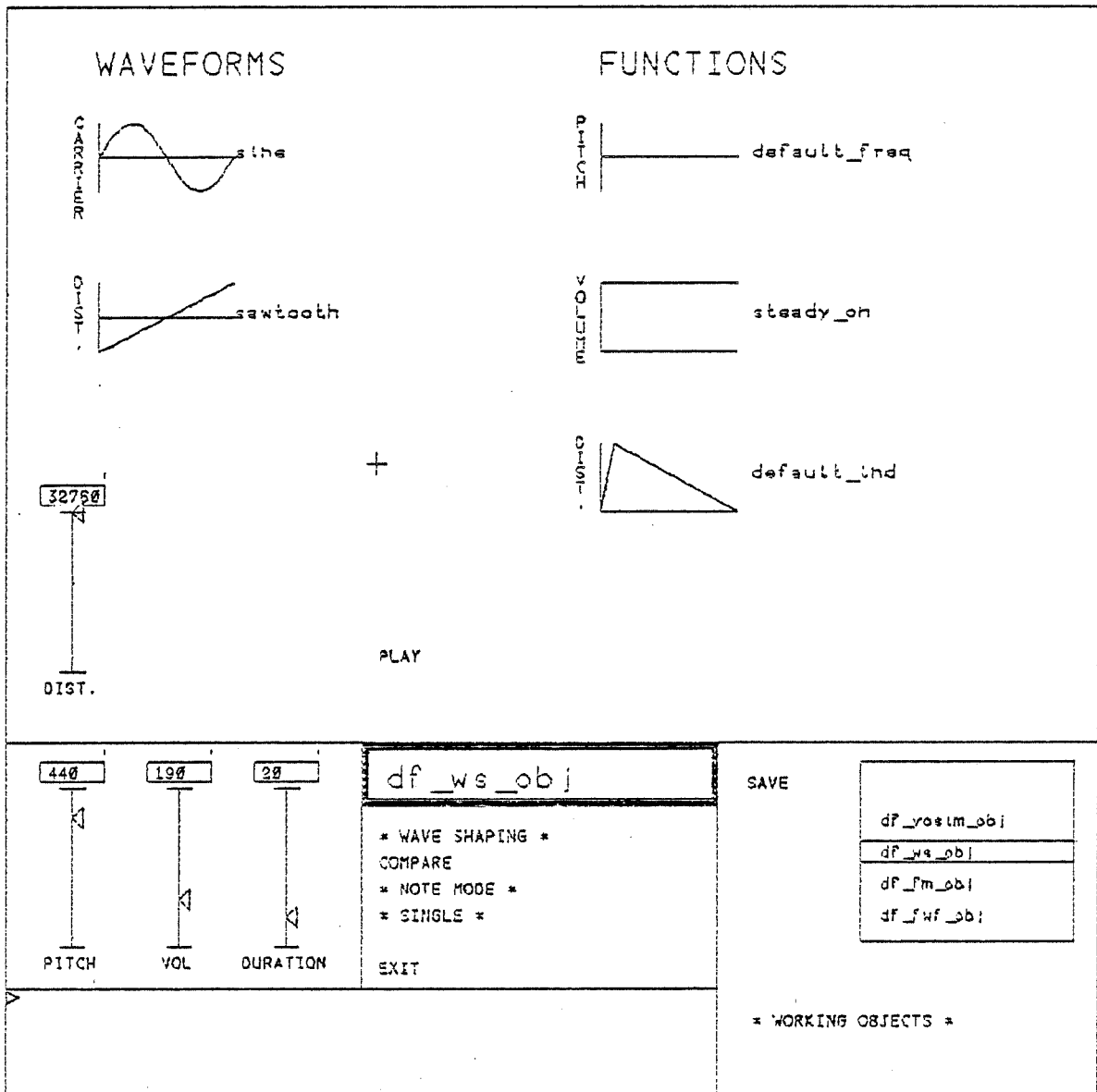


Figure 12. Editing Waveshaping Objects

12. ADDITIVE SYNTHESIS AND BANK

In order to edit objects according to the additive synthesis model a different program must be used. It is called *bank* and its usage is very similar to *objed*. Here we shall only point out differences and functions which have not yet been seen. A view of the *bank* editing environment is seen in Figure 13. The program allows the specification of a function to control the amplitude of each of up to the first 16 harmonics of a tone. The spectrum defined by the object in the figure has four sounding partials. To (re)define the function controlling any partial, point at it and activate the Z-button. The means which is then used to specify the function depends on the "tool" selected in the lower centre panel. The default mode uses rubber-band lines as already seen in *objed*. Typing, drawing free-hand curves, and retrieving previously stored functions are available alternatives. For example, Figure 14 illustrates how pre-defined

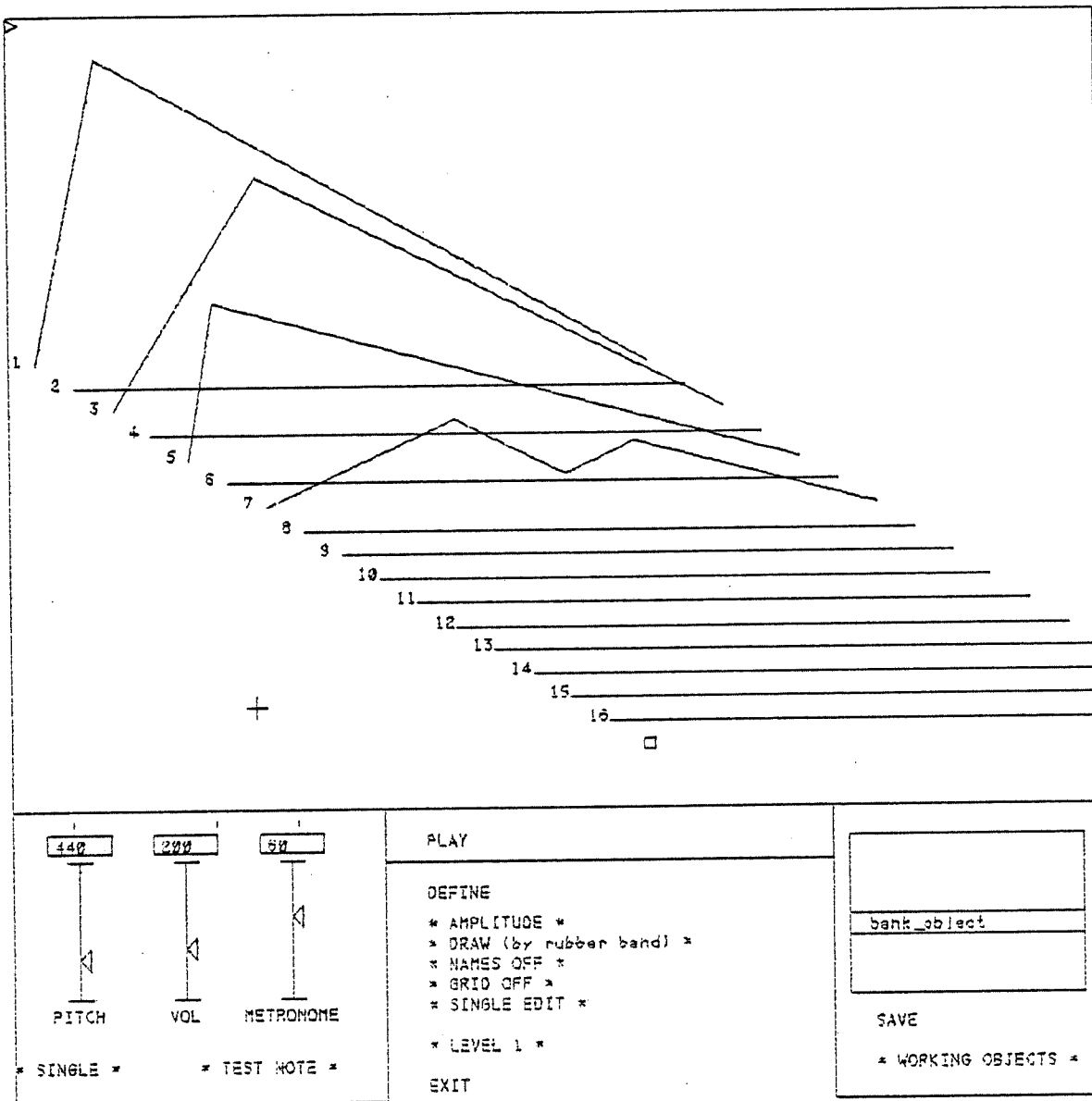


Figure 13. The BANK Editing Environment

functions are accessed. Here the window shows both the names *and* the shapes of the functions in the user's directory. The approach is a cross between the technique previously seen in selecting waveforms, and the use of directory windows.

Notice that the lower-left panel is slightly different from *objed*, but the meaning of the repositioned buttons should be obvious.

One nice thing about the program is that it allows you to view the functions from various perspectives. Notice that near the 16th harmonic that there is a little box. Point at this, depress the Z-button, and *while holding it down*, drag the "Z" axis of the functions to a new position. Using this means, the spectrum seen two figures previously can be viewed synchronized in time (as seen in Figure 15) and overlaid for tracing (Figure 16). Notice in the last figure how a grid can be requested as an aid to drawing functions. Regardless of how the functions are oriented on the page, the grid will

APPENDIX A

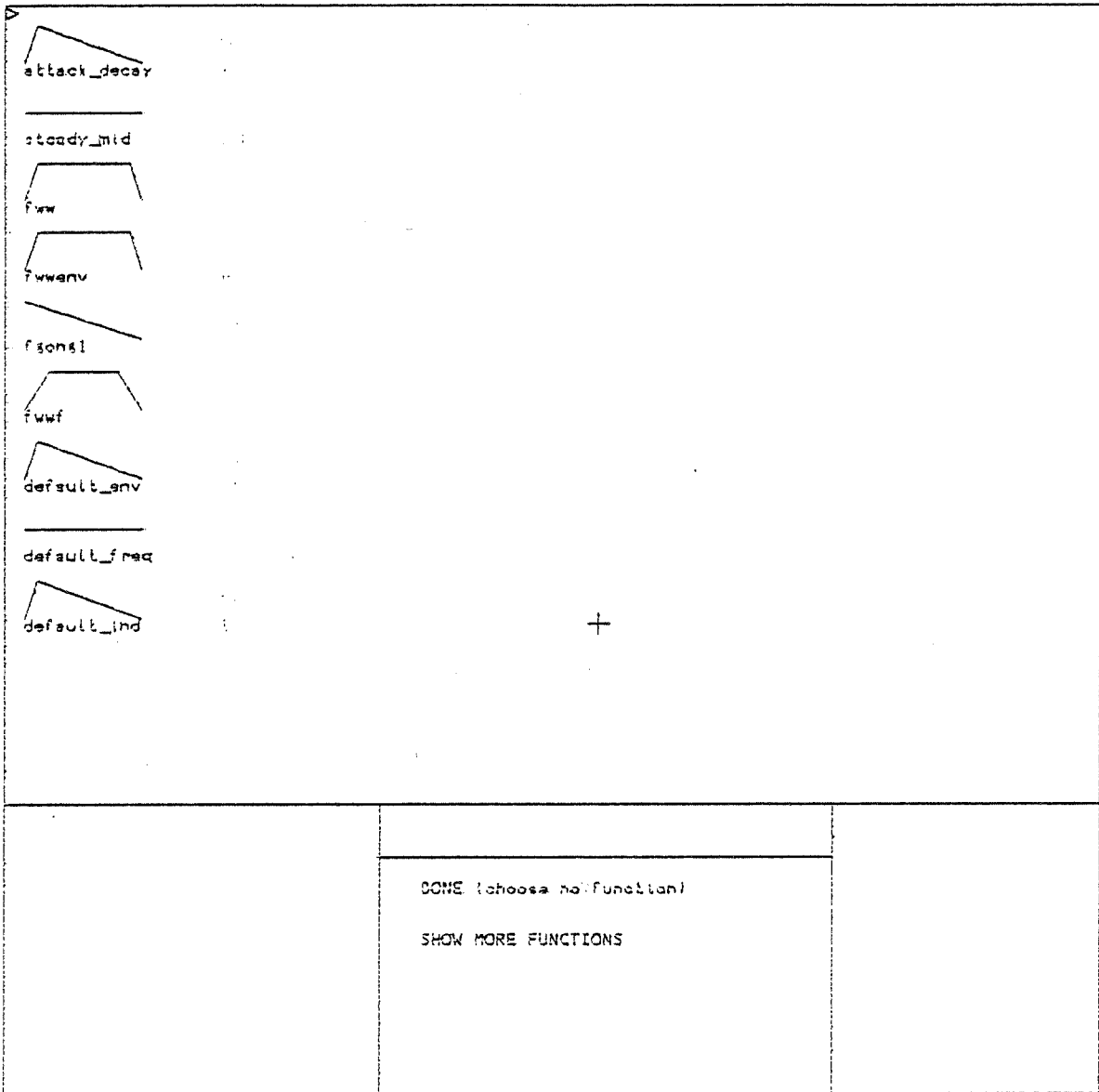


Figure 14. Selecting From Predefined Functions

always be positioned at the origin of the function being edited. The grid is requested using the switch in the lower central region of the display.

For each harmonic, *bank* also permits a function to be defined which controls variation of pitch over time. To do so the user must activate the * AMPLITUDE * switch in the lower middle region. The frequency functions can then be edited in the same manner as those controlling amplitude. Activating the same switch (which now reads *FREQUENCY *) a second time will permit the amplitude and frequency functions to be viewed and edited simultaneously. This is seen in Figure 17.

In working with frequency functions it is often useful to specify the same function for more than one partial. To do so, activate the button * SINGLE EDIT * which switches you to * GROUP EDIT * mode. Then point at each function which you want included in the group. Depressing the Z-button over each will cause an "arrowhead" marker to

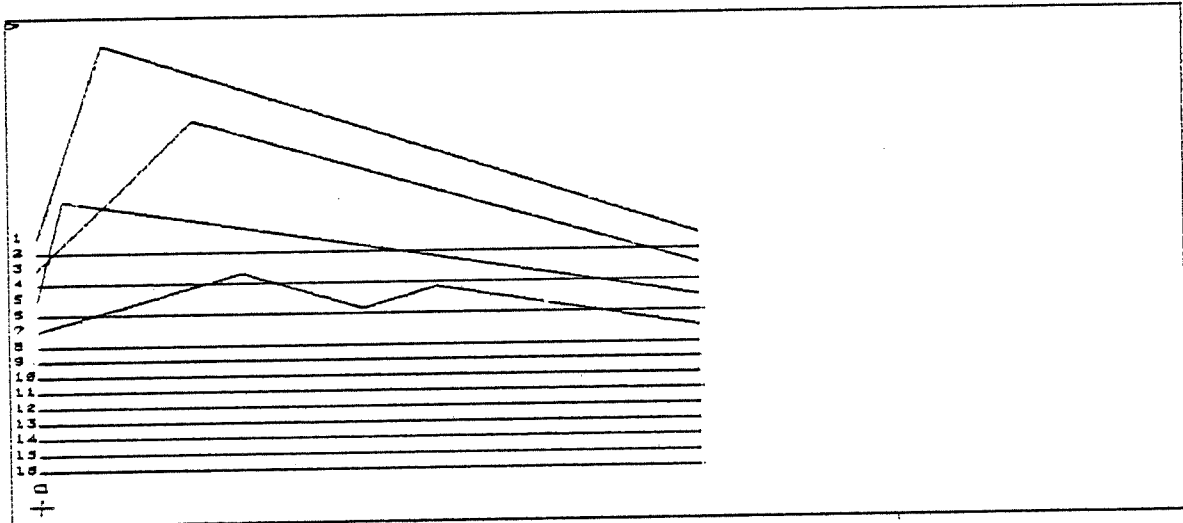


Figure 15. Functions Synchronized in Time

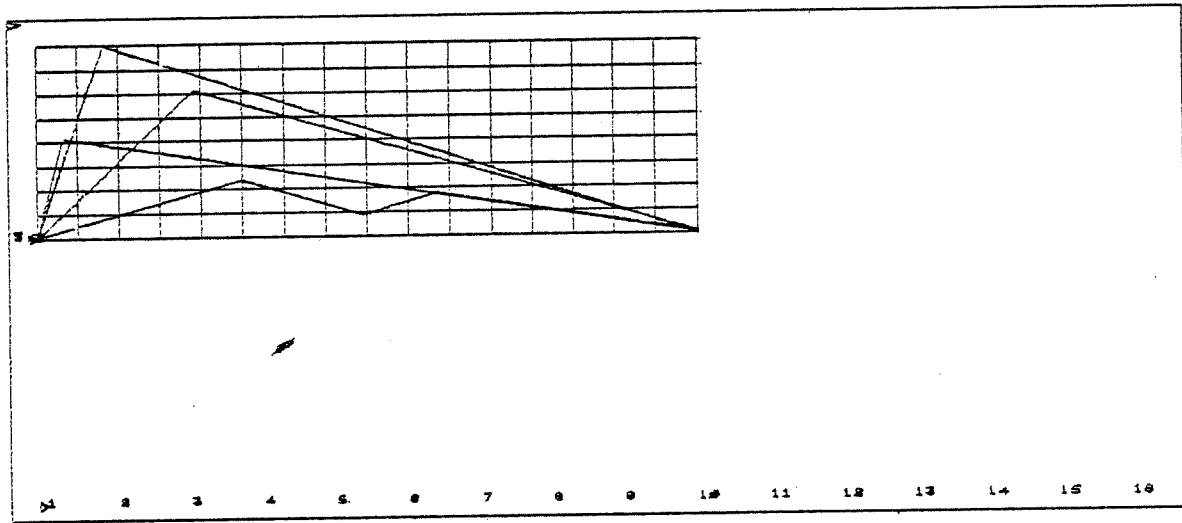


Figure 16. Overlaid Functions With Grid

appear over the function. When all functions have been "pointed out", activating the DEFINE button will allow the function to be defined in the usual way.

Finally, remember that each function specified is a file. This can be verified by activating the * NAMES:OFF * switch. The result is that the name of each function is printed (as seen in the last figure). Note that names for new functions are generated automatically by the system.

APPENDIX A

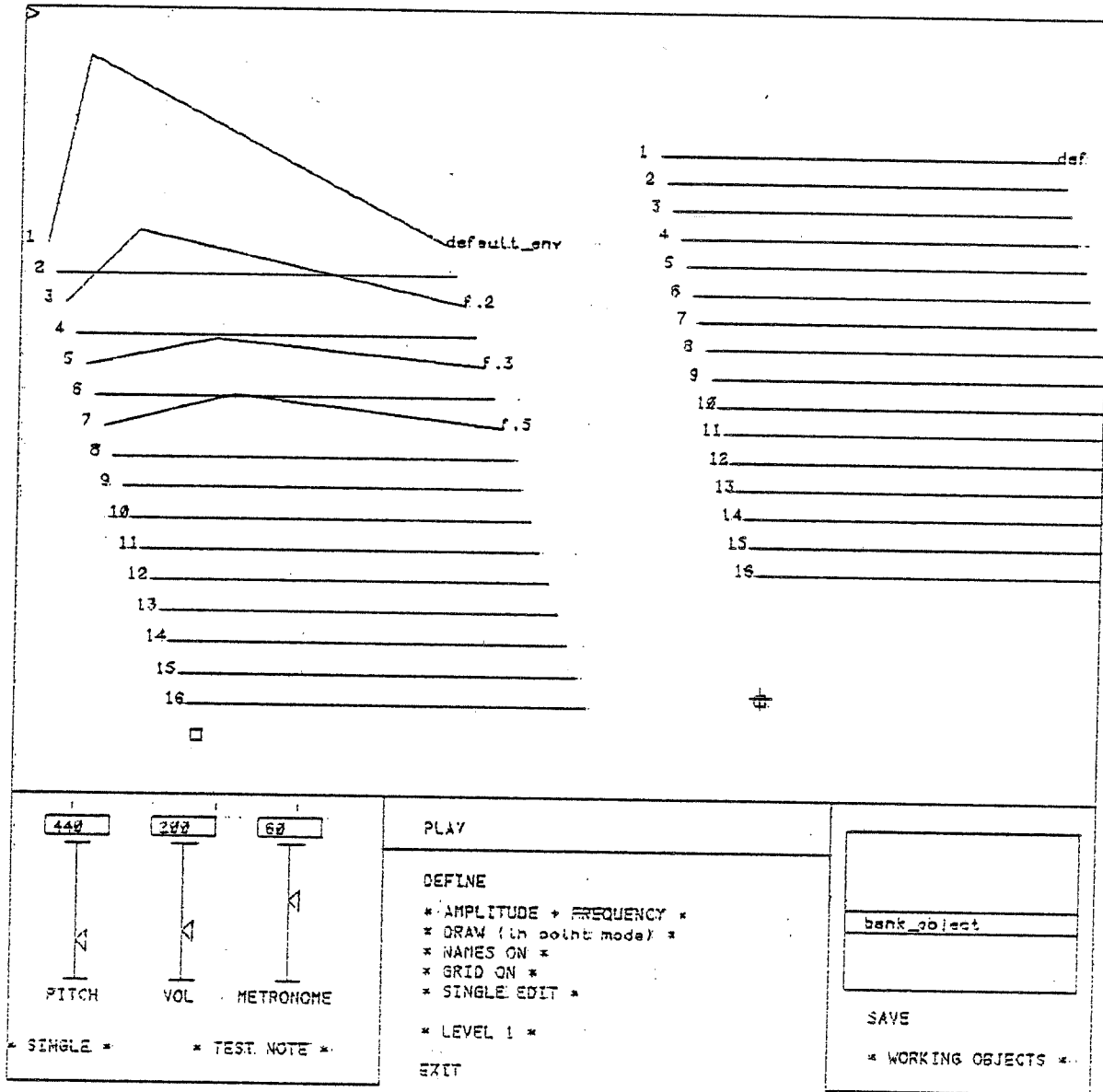


Figure 17. Editing Frequency and Amplitude Functions

13. REFERENCES

Arfib, D. (1978). Digital Synthesis of Complex Spectra by Means of Multiplication of Non Linear Distorted: Sine Waves. *AES Preprint No. 1319 (U-2)*.

Chowning, J. (1973). The Synthesis of Complex Audio Spectra by Means of Frequency Modulation, *Journal of the Audio Engineering Society* 21: 526-534.

Grey, J. (1975). *Exploration of Musical Timbre*. Stanford Univ. Dept. of Music Tech. Rep. STAN-M-2.

----- (1977). Multidimensional Perceptual Scaling of Musical Timbre. *Journal of the Acoustical Society of America* 61: 1270 - 1277.

A Tutorial on Editing Objects

Kaegi, W. and Tempelaars, S. (1978). VOSIM-A New Sound Synthesis System. *Journal of the Audio Engineering Society* 26: 418-424.

Le Brun, M. (1978). Digital Waveshaping Synthesis. *Journal of the Audio Engineering Society* 27: 250-266.

Moorer, J. A. (1977). Signal Processing Aspects of Computer Music - A Survey. *Proceedings of the IEEE*, 65: 1108-1137.

Roads, C. (1979). A Tutorial on Non-linear Distortion or Waveshaping Synthesis. *Computer Music Journal* 3.2: 29 - 34.

APPENDIX B

APPENDIX B - A Tutorial Introduction to SCRIVA

1. INTRODUCTION

Scriva is a computer program which allows graphically displayed scores to be edited by the user. By editing, we mean that the program permits the specification, modification, viewing, and auditioning of score material. Work may be saved from session to session, and may be combined with material composed using other means.

A key feature of *scriva* is that it allows the same score material to be notated in different ways. Some examples are seen in Figure 18, which shows eight different notations of the same score material. This variety helps the user in performing many different tasks by allowing different attributes of the music be made prominent in the notation. For example, if you want to specify pitches outside the chromatic scale, you can use "roll" rather than common music notation (CMN). If you are orchestrating a *klangfarbenmelodie*, then you may want to use the notation which highlights timbre. It only takes about one second to re-notate any score. (In what follows, you will see that some transactions -- such as adding notes -- are undertaken differently, depending on the notation being used.)

It is important that the user realize right from the start that *scriva* is an experimental program. It was designed to test certain basic concepts, rather than provide a comprehensive compositional tool. As such, it has some very pronounced limitations. When the system is busy, response time is very poor. Only note durations which are multiples of a 32nd note can currently be specified when using CMN. As the music becomes more complex, the program's ability to notate it in readable CMN breaks down. Finally, there are rather severe restrictions on the size of score which can be edited using the program (circa 200 notes). *Scriva* does, however, serve as a reliable and useful tool, and for each of the above restrictions, there exist alternative solutions within the SSSP system.

This document is, by necessity, only introductory in nature. It assumes that the user has read (and understood) Chapters One and Two of the *SSSP User's Manual*, and is therefore familiar with the basic concepts of naming files, conventions for typing, and interaction based on graphics. It presents the basic concepts of *scriva*, describes how to carry out key operations, and serves as a reference for explanations of the various options available (see the final section).

The program is best understood by a combination of this document and practical experience. Try out the examples described! Additional information is available in the article "The Evolution of the SSSP Score Editing Tools" found in the *Computer Music Journal*, Volume 3, Number 4. Beyond this, an experienced user is your best source of information.

2. GENERAL

The usage of *Scriva* is as follows:

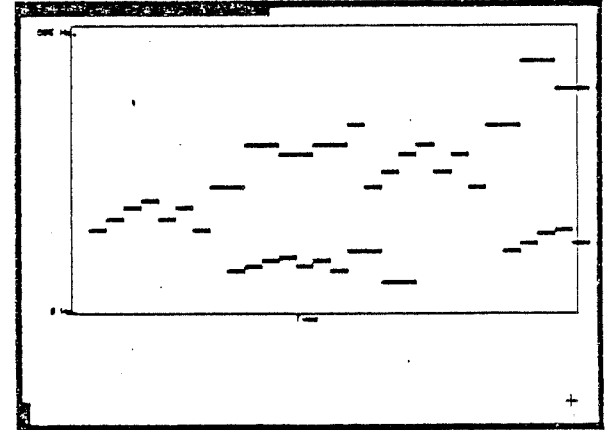
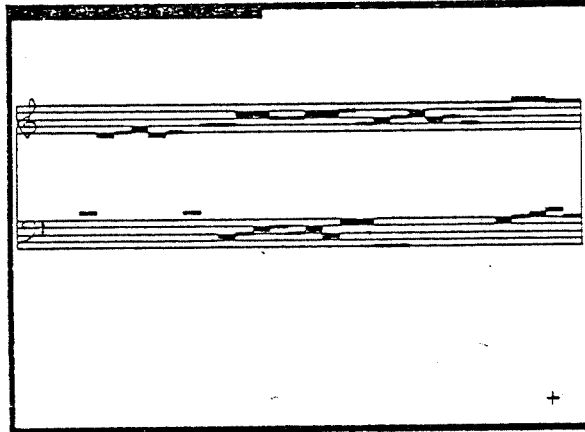
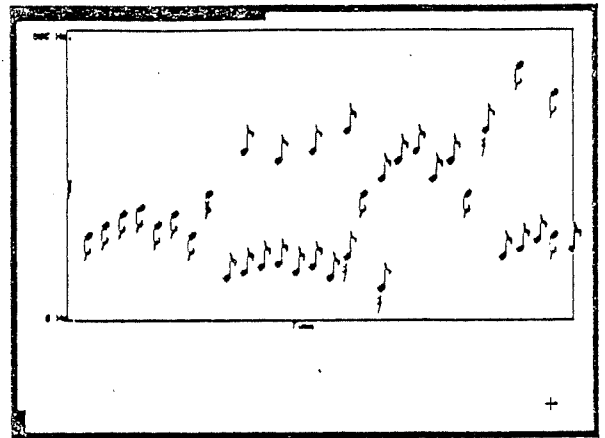
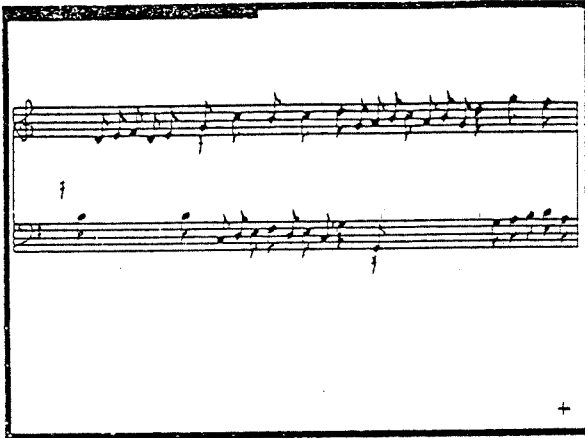
```
scriva [scorename]
```

If the optional score name is specified, then the program assumes that that is the name of the score to be edited. If it is unspecified, it assumes (by default) that the name of the score is "m.out". In either case, if there already exists a score of that name, it will bring it into the workspace. Otherwise, it will present you with a "fresh slate". If we assume the latter case, typing

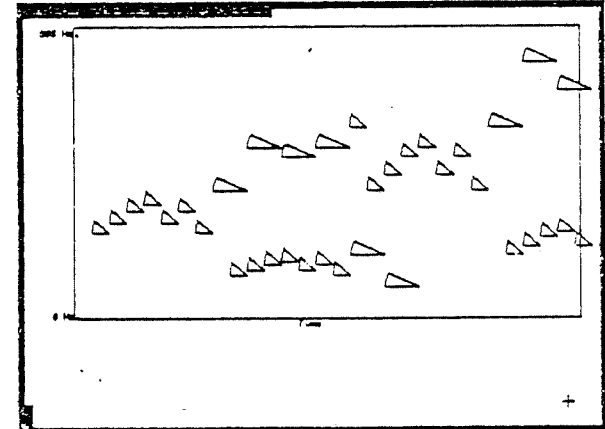
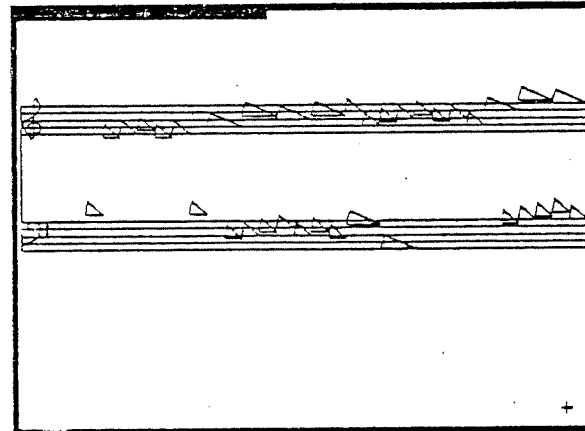
```
scriva
```

will cause what is seen in Figure 19 to appear on the screen.³

CMN



AMP



OBJ

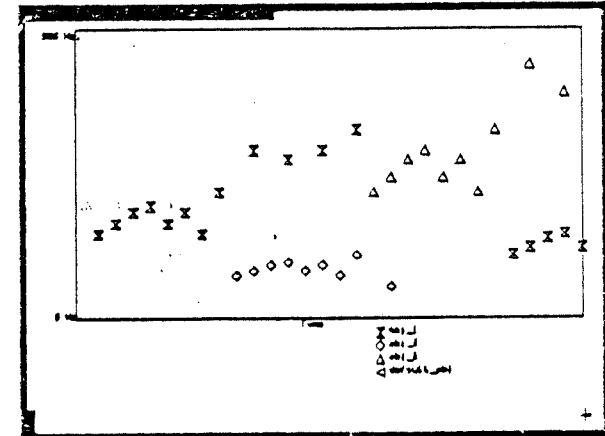
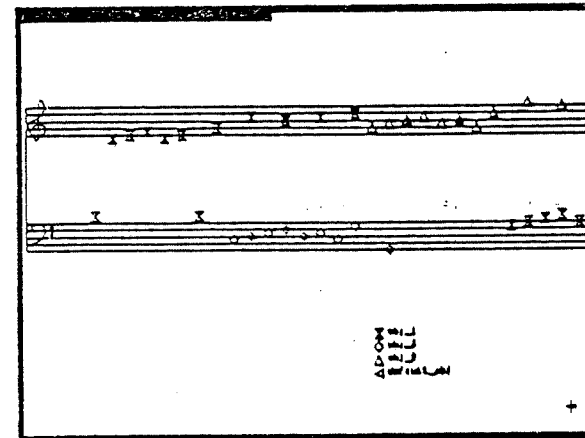


Figure 1. Notational Flexibility in SCRIVA

3. It should be noted that *scriva* can be called from any terminal, but it "set up shop" on the graphic wonder. In so doing, it ties up two terminals. Normally, therefore, it should only be called from the g.w.

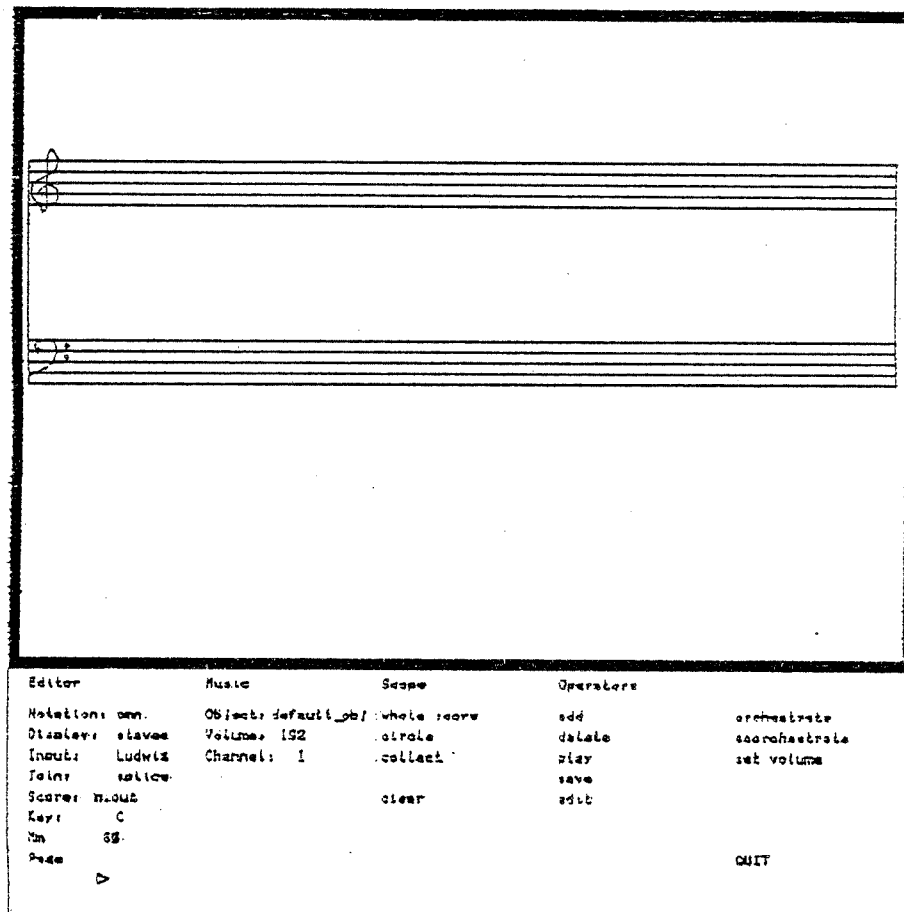


Figure 2. SCRIVA Upon Initial Entry

Before going into further detail, we should examine the general layout of this display. First, note that the screen is divided into two sections. The top section is the "working space" where the music being edited appears. The lower section is the control area, where most of our editing "tools" appear. Close examination of this lower section (Figure 20) will show that it is divided into five columns under four headings. The first column, under the heading "Editor", contains fields concerning the basic editing environment. Control over notation and score name, for example, are in this region. The second column, under the heading "Music", concerns various attributes of any notes which are added to the score. The commands falling under the third heading, "Scope", have to do with enabling you to address yourself to specific "chunks" of the score which you would like to affect (for purposes of playing, or orchestration, for example). Finally, the last two columns (under the heading "Operators") are the primary editing functions, or "commands" available in the program. They allow you to undertake the following transactions:

Editor	Music	Scope	Operators	
Notation: cmn	Object: default_obj	whole score	add	orchestrate
Display: staves	Volume: 100	circle	delete	scorechestrate
Input: Ludwig	Channel: 1	collect	play	set volume
Join: splice			save	
Score: m.out		clear	edit	
Key: C				
Mm: 66				
Page:				QUIT

Figure 3. SCRIVA Command Window Layout

- build up a score by adding notes
- delete notes from a score
- play score material
- save score material
- orchestrate notes of the score
- modify the volume of notes in a score

Functionally, this is all that *scriva* can do. All else (such as everything in the upper half of the screen and in the other three columns) exists solely to support the user's ability to undertake these primary tasks. Manipulations and modifications to score data beyond these functions must be undertaken using tools which exist outside of *scriva*. (It will be seen, however, that external functions are easily invoked without having to exit from *scriva*. Hence, the power of the environment is significantly extended.)

We shall now provide a brief tutorial on how to undertake each of these primary functions, or transactions. Along the way, we will introduce the light-buttons in the other columns as needed. A column-by-column summary of the function of all light-buttons is given in the reference section concluding this tutorial.

Note that in the documentation, references to light-buttons are always followed by a Roman numeral in parentheses, as in *add(IV)*. This is a notational convenience to indicate to the reader the column in which that light button is found. If during the tutorial additional information on a light button is desired, simply look it up in the concluding reference section under the documentation for the indicated column.

3. ADDING NOTES

3.1 General

There are different ways of adding notes to the score being edited using *scriva*. Notes can be added one-by-one, or in groups from previously defined material. To begin adding notes, activate the *add(IV)* button. The technique which will be in effect is that opposite the *input(I)* button. The *Ludwig* input tool is initially opposite the *input(I)* button on entering the program; therefore we shall explain it first.

APPENDIX B

3.2 Ludwig

This input mode is in effect if the *add(IV)* button is activated with *Ludwig* opposite the *input(I)* button. When in effect, one or two vertical "ladders" indicate where new notes can be added. one or two vertical "ladders" (as seen in Figure 21a).

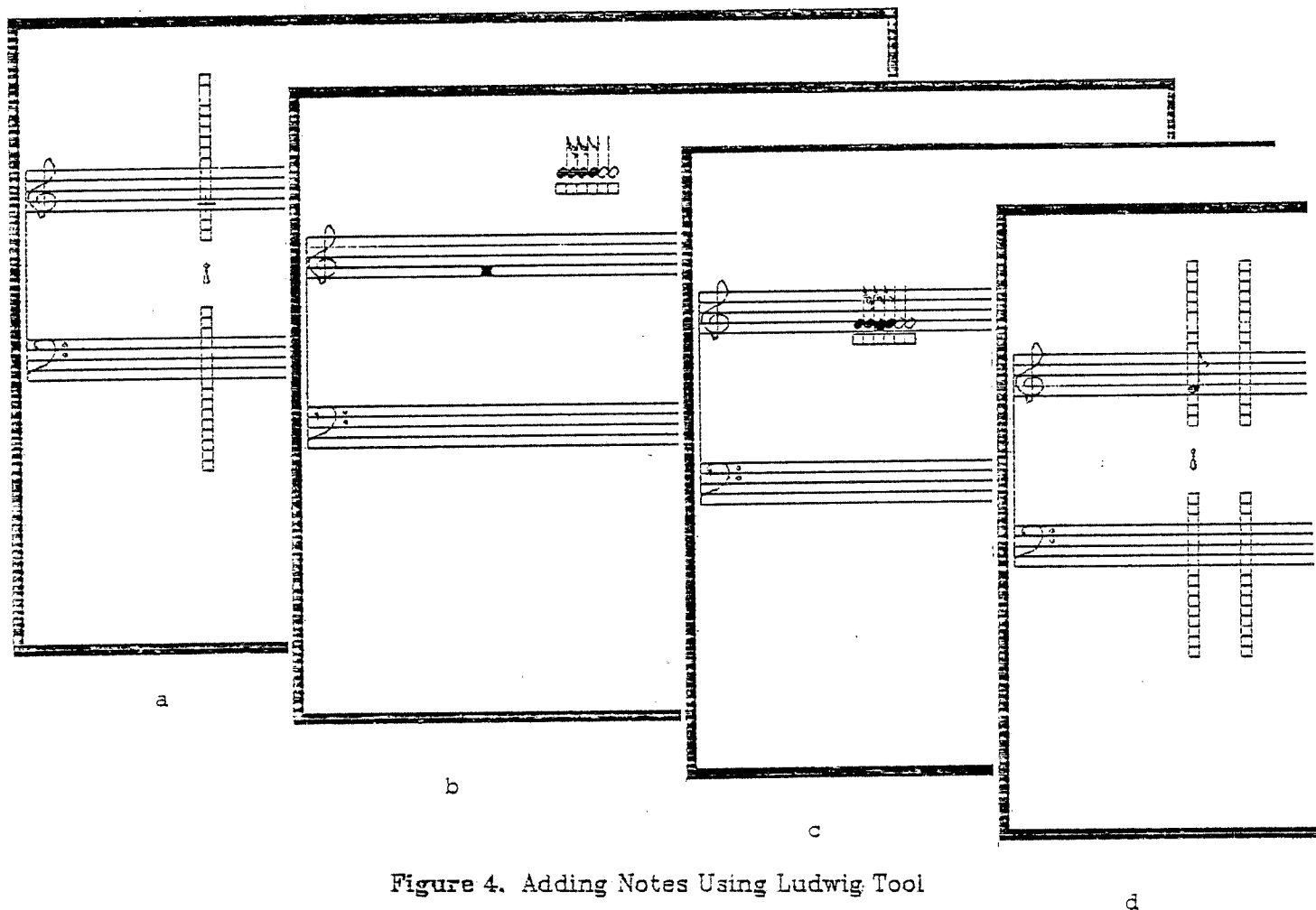


Figure 4. Adding Notes Using Ludwig Tool

To indicate the pitch and entry point of the note to be input, position the tracking cross over the appropriate vertical position within one of the ladders. In Figure 21a, for example, the position of the cursor indicates that a f_4 is to be input. Once positioned, if the Z-button of the cursor is depressed *and held down*, three things will happen. First, a stationary marker will appear at the spot pointed at. This indicates the pitch and entry point. Second, the tracking symbol becomes a set of notes of durations from a 32nd note to a whole note set above a row of boxes. Third, the ladders disappear. All of this is seen in Figure 21b, where we have moved the tracking symbol off to the side to be better able to see the stationary marker. If you are trying this for the first time, move the cursor about while holding down the Z-button in order to become relaxed with the motion. Now, to specify the duration of the new note, all we need do is position the tracking symbol note of the appropriate duration over the marker, and release the Z-button. If we select an 8th note as in Figure 21c, then on releasing the Z-button we will have the completed result notated as in Figure 21d. Once the note is entered, the ladders are repositioned so that the first one is synchronous with it, and the second one offset to the right by an amount corresponding to the note duration. We can now add additional notes which either start simultaneously with

the last note entered (thereby forming a chord), or follow it (thereby forming a melody). We just select the first or second ladder, respectively.

Note that in using the tool wasted motion is avoided by exploiting the redundancies of much music. The duration which automatically appears above the marker is always equal to the last duration entered. Verify this by adding notes of different durations.

To add rests, exactly the same technique is used, except the box *below* a note is placed on top of the marker, instead of the note itself. Thus, the box below the 8th note will cause an 8th rest to be entered. Note that for both notes and rests the method used is just the opposite of the menu techniques used thus far. Rather than having a moving pointer select from a stationary menu, we select from a moving menu using a stationary pointer. At any point, if you want to hear what you have written, activate the *play(IV)* button. You can then continue adding notes without having to first reactivate *add(IV)*. (The performance can be interrupted at any time by hitting the key labelled "RUBOUT".

To get note durations which are combinations of those on the tracking menu, add two notes of the same pitch one after the other and then activate the light button shaped like a neck-tie which is positioned between the left-hand ladders. This will cause the two notes to be "tied" together. Only durations which are multiples of a 32nd note are currently possible with the *Ludwig* tool.

To get accidentals, adjust slider 2 up or down. This will cause the last note entered to be dragged up or down semi-tone by semi-tone. If you want to enter notes in a particular key and have the accidentals set automatically, activate the *key(I)* button, and select the appropriate major key from the menu which appears in the bottom left-hand corner of the work area, and which is illustrated in Figure 22.

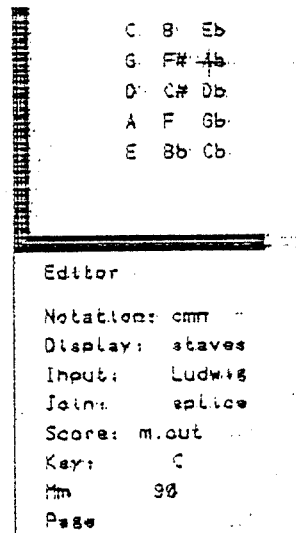


Figure 5. Key Menu

After hitting *key(I)* (or most any other light button, with the notable exception of *play(IV)* -- as already mentioned) the ladders will disappear. This indicates that you must reactivate *add(IV)* before new notes can be input. However, if some notes have already been defined, just reactivating *add(IV)* will produce an error message stating, "YOU MUST IDENTIFY A REFERENCE NOTE". This is because the program does not know where you want to begin adding your new notes. That is, it does not know where to

APPENDIX B

position the ladders. You must indicate this by pointing at the note closest to where you want to begin working, and depressing the Z-button on the cursor. If *scriva* understood you, it will draw a triangle beside the indicated note (as shown in Figure 23).

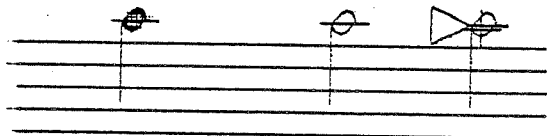


Figure 6. Identification of the 'Reference' or 'Current Note'

If you missed, try again. Once you have identified the "reference" note, if you want to add, you must *as the very next step* activate the *add(IV)* button. The ladders will appear, and you can begin adding as before.

To change the tempo at which the score is played, activate the *MM(I)* button, and type in the new mm marking (expressed in quarter notes per minute). Finally, it is important to note that the *Ludwig* tool only works when the music is notated on the piano staves.

Before progressing, practice adding notes with the *Ludwig* tool. In your experiments, try writing chords, melodic structures, rests, and notes of different durations (including tied notes). Constantly play what you have written (at different tempi) in order to verify that what you got was what you intended. Verify that changing key has no effect (except notational) on notes already written.

In the previous exercise, it may be desirable to periodically delete a note. You can do this by indicating it as if to make it the reference note (as seen in adding notes), and then activating *delete(IV)*. Deletion will be discussed in more detail shortly.

3.3 Roll

To keep new music fans happy, we will now look at a different way of adding (and notating) new notes. First, let us change the notation used. Activate the *notation(I)* button, and select *Roll* from the menu which appears in the lower left-hand corner of the work space, and which is illustrated in Figure 24.

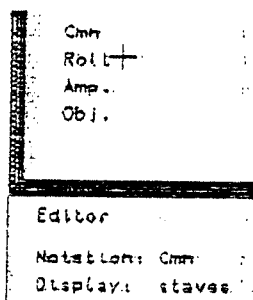


Figure 7. Notation Tool Select Menu

Notice that all material composed so far is re-notated in "piano-roll" notation. Activate the *play(IV)* button to verify that changing notation affects the music visually, not sonically. Try adding a few more notes using *Ludwig* just to verify that it still works. (Remember to first identify a reference note.) Now, activate the *display(I)* button. This, in effect, changes our manuscript paper from piano "staves" to frequency/time

space ("linear"). If we now try to add notes with *Ludwig* we will get an error, since it only works with the piano staves.

To add notes in our current state, we must select a different input tool. This we do by activating the input(I) button, and selecting *Roll* from the menu which appears in the bottom left-hand corner of the work space, and which is illustrated in Figure 25.

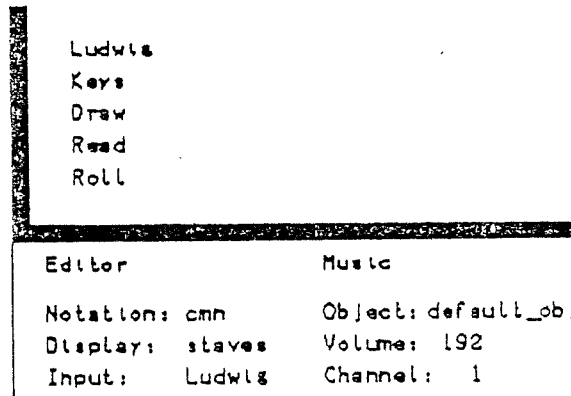


Figure 8. Input Tool Selection Menu

We can now activate the *add(IV)* button. All light buttons will now disappear and a new one, "DONE", will appear in the lower right of the screen. The roll mode of input is free-form, so no reference note need be specified, and no ladders appear. To input a note, position the tracking cross to that point in frequency (vertical position) and time (horizontal position) where the note should start. Depress the Z-button of the cursor, and *while holding it down*, drag it to the right. Notice that a "rubber-band" line is stretched horizontally from the point where the Z-button was depressed to the current position of the cursor. This line, which remains horizontal regardless of any vertical motion of the cursor, indicates the duration of the note being entered. When the duration is "stretched" to the desired length, release the Z-button, and the note is entered. You can add as many notes as you like in this way. (Also, notes can be drawn left-to-right or right-to-left.) When you are finished, or want to hear what you have done, activate the *Done* button which has temporarily appeared in the command area. The *play(IV)* and all other buttons will then reappear, and may be activated.

It is interesting to see what happens when you switch back to the piano staves by activating the *display(I)* button. All notes are notated as being on one of the 12 notes of the chromatic scale in spite of the fact that many were specified at frequencies which do not correspond with such pitches. If you play the material, however, you will notice that the sound of the material has not changed, just the notation. The program does its best to notate things to the closest pitch and duration, but does not change the actual data values to conform to notational limitations.

4. WHY DID IT SOUND LIKE THAT?

In adding notes, we specified pitches, starting points, and durations. However, when the material was played, it should have been noticed that a timbre, volume, and output channel had somehow been determined for each note. The attributes assumed for each of these parameters are controlled by the values associated with the light buttons in column two. Thus, if the name opposite *Object(II)* is "default", that is the name of the object associated with each note specified thus far. Similarly, if the value 190 is opposite the button *Volume(II)*, 190 is the volume of each note. However, if we activate *Volume(II)* and type the number 230 in response to the prompt of the terminal icon, all

APPENDIX B

subsequently added notes will have a volume of 230. Note that this change will have no effect on notes already specified. Verify this by experimentation. Change the volume (the range is 0 - 255), and then add some new notes. Play the composite score and note the change in dynamics. Add other notes using other volume settings. Notice that a change of volume of 20 is a change of about one dynamic marking.

You have already seen how the notation of the notes can be changed. Activate the *notation(I)* button and select *Amp* mode. Notes are now notated with the amplitude envelopes which are associated with them. Notice how envelope height is proportional to note volume.

Now read the description of *Object(II)* and *Channel(II)* (in Section 14.2 of this tutorial), and verify that you understand how they work. (Remember that you will not hear the score performed as orchestrated unless you have already created objects having names corresponding to those used in orchestrating. To create objects, you must leave *scriva*, and use the program *objed*.)

5. MIX AND SPLICE

Switch to the *Roll* notation and input tool (using the *notation(I)* and *input(I)* buttons). Add some notes in melodic sequence. Now add a new note somewhere in the middle of the sequence. Notice that in so doing, all notes to the right of the new one are offset in time. We have, in effect, opened up a space in the middle of the sequence into which the new note is fit. From analogy with the tape studio, we call this "splicing". This is not, however, how we always want to work. Think of the case, for example, where you first add a bass part, and then want to add the treble. In this case, you want to "mix" the new voice in with the old, without affecting the timing relationship among any of the existing notes.

Scriva makes both options available for all input tools, and the mode in effect is set by the *Join(I)* button. Activate the button (thereby switching from *splice* to *mix*). Add some notes and verify the difference in effect between the two modes. Switch back to *splice* mode (by reactivating the *Join(I)* button) and reconfirm the difference. Verify that the mode equally affects inputting notes in CMN using the *Ludwig* tool.

6. DELETING MATERIAL

As hard as it is to believe, we sometimes make mistakes, and therefore want to delete material from the score being edited! This is done by activating the *Delete(IV)* button. Before this has any effect, however, we must first indicate what note is to be affected. This is done by pointing, and depressing the cursor Z-button (just as was done to indicate the reference note with *add(IV)*). Activating *delete(IV)* will now cause the note to be removed from the score.

Indicating a note to be operated upon by pointing in this manner is a reoccurring action in *scriva*. It is referred to as the specification of *immediate scope*. Notice that if you change notation after defining immediate scope, the triangle will disappear and you will have to re-identify the note before it can be deleted. Immediate scope remains in effect for only one button activation.

There are two other points to notice before moving on. First, verify that what happens when a note is deleted in the middle of a sequence is affected by whether *join(I)* is set to *mix* or *splice*. Second, verify that the specification of immediate scope can be used in conjunction with the *play(IV)* button. The effect will be that only a single note will be played: the one you pointed at. We will see that immediate scope can be used with all of the operators in columns IV and V. But first, we should investigate the general concept of scope in more detail.

7. AN ASIDE: THE CONCEPT OF SCOPE

We are used to thinking about the scope of a discussion. By this we mean the topics which are to be encompassed by the conversation. Similarly, we can talk about the scope of an operator in *scriva*. By this, we mean the notes which are to be encompassed, or affected, by that operator. We have seen how the implicit scope of *play(IV)* is the entire score, but that an explicit scope of one note can be specified by defining immediate scope. We will now see how the scope of an operator such as *play(IV)* and *delete(IV)* can be set to include whatever notes we desire. This is accomplished using the buttons in column III.

There are three techniques for collecting notes into the *current scope*. First, one can explicitly specify that all notes of the score are to be included. This is done by activating *whole score(III)*. Second (as is illustrated in Figure 26), one can draw a circle around all of the desired notes after activating *circle(III)*.



Figure 9. Scope Specification by Circling

Finally, notes can be collected one-by-one by pointing at them after the *collect(III)* button has been activated. Read the documentation for each of these buttons (Section 14.3), and experiment with their use in conjunction with *play(IV)* and *delete(IV)*. Notice that unlike immediate scope, the current scope remains in effect until *clear(III)* is activated. Also, additional invocations of the three *scope* buttons have a cumulative effect, adding to the current scope rather than changing it. Note also that current scope is visually indicated by the notes being drawn at a brighter intensity. Finally, note immediate scope always has precedent over current scope.

8. SAVING MATERIAL

In editing, it is important to remember that you are working on a "scratch pad" version of the score. Normally, if you exit the program by activating the *quit(V)* button, all your work will be lost. Obviously it is desirable to be able to save work from day-to-day so as to be able to work on it in subsequent sessions. Scores are saved by activating *save(IV)*. It is important to know, however, that in order to retrieve anything, it must be filed under a specific *file name*. Therefore, when you save data using *save(IV)*, you must know the name under which it is saved. This is the score name which is opposite *score(I)*. This name can be changed by activating *score(I)*. As a result, the names of all previously defined scores (if any) will appear in a box in the lower left hand corner of the work space (as illustrated in Figure 27). As well, a terminal icon will appear as a prompt below the *score(I)* button. This is also seen in Figure 27. The score name can be set to that of an existing score by pointing at its name, or moving slider 1 until its name appears between the pair of horizontal lines in the centre of the box. Alternately, any name of less than 14 characters (whether previously used or not) may be typed in using the keyboard. Once the name is specified using one of these techniques, the menu of score names and the terminal prompt disappear, and the new

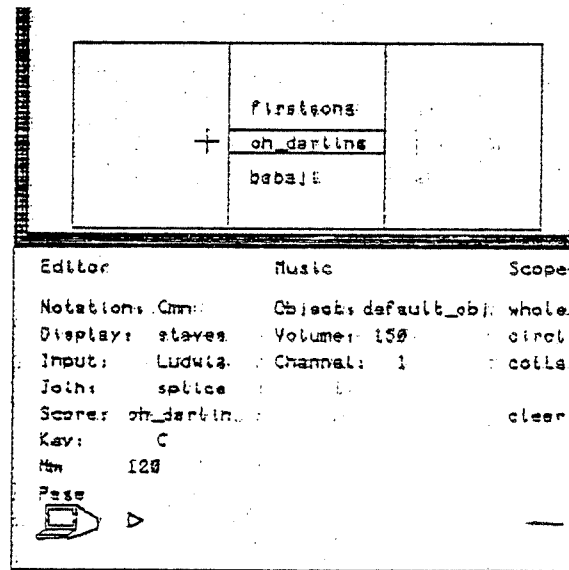


Figure 10. Window Showing Names of Saved Scores

name appears opposite *score(I)*.

Always beware that only one thing can be saved at a time under any particular name. If the name opposite *score(I)* is the same as that of some existing file, activating *save(IV)* will cause the previous contents of the file to be overwritten. (That is, the former contents of that file will be erased and replaced by the material being saved. Finally, it is useful to note that you can save parts of a score, since the notions of current and immediate scope also work for *save(IV)*. This is especially useful as a means of extracting fragments of a score so as to be able to use them in other contexts without having to re-specify them from scratch.

9. ADDING BY READING

Now that we can save material, it is important to know how to retrieve it. One way which has already been seen is to give the name of a score as an argument to *scriva* when it is originally called. Once inside the program, however, we must use another approach. First, consider the fact that in retrieving a score we are simply adding notes, but as a group rather than one-by-one as seen thus far. Therefore, the way to add notes from a previously defined file is to select the appropriate input tool by activating *input(I)*. The tool which we need is *Read*. Once it has been selected, we can add the notes by activating *add(IV)*. Just as before, however, it is important to specify a reference note so as to determine where to start adding. Also, note that the mode specified by *join(I)* still affects the input.

So, first set the join mode and select the read input tool. After determining the reference note, activate *add(IV)*. You will then be confronted with exactly what was seen when *score(I)* was activated. *Scriva* is asking you to indicate the name of the score to be retrieved. Do so in the same way as before. Once the name has been communicated, the score will be read in and appear on the display. Notice that this causes the name associated with your current score to be changed to that of the score read in. Beware when you save the score because you may write over the saved version of the read-in score!

Before going further, spend some time practicing saving and retrieving scores. Set *join(I)* to mix and try and write a simple canon by reading in the same score fragment

on top of itself, but with each entry staggered. Work with different notations. After saving material, set the current scope using *whole score(III)* then activate *delete(IV)*. Then build up a new score exclusively from stored fragments. Try thinking of the program as a multi-track tape recorder where each score file is one track, and retrieving scores on top of one-another in mix mode is like the mix-down. Do not progress until you are completely comfortable with the material covered thus far.

10. ORCHESTRATION

Through *object(II)* we have seen that associated with each note is the name of an "object" which determines its timbre. *Orchestrate(V)* permits the name of this object to be changed for any note, without having to redefine the whole note. On activating the button, a list of existing object names (if any) appears in the lower left-hand corner of the work space, and the terminal prompt appears opposite *object(II)*. By specifying an object name (by typing, or selecting from the menu - as with *object(II)* and *score(I)*) the notes encompassed by the scope in effect will be reorchestrated with that object. Experiment with *orchestrate(V)* using both immediate and current scope. Notice that with immediate scope, using the menu of object names and pointing is much like dipping a brush into a paint-pot on a palette and painting the notes with a particular colour. There is a problem, however: it is difficult to remember what "colour" is associated with any note. The solution to this is to activate *notation(I)* and select *Obj*. This is a notation which highlights orchestration, and which functions like a topographical map. Each note is drawn as a graphic icon which represents a particular timbre. The association between timbre and icon is shown in the "key" in the bottom right-hand side of the work space (as can be seen in the two lowest panels in Figure 1). Therefore, the object associated with any note is immediately visible, the notation being tailor made for the command. Finally, note that *object(II)* is set to the name of the last object used in orchestration and therefore affects notes subsequently specified one-by-one using *add(IV)*.

11. SETTING VOLUME

Like orchestration, the volume of any note can be changed without having to redefine the whole note. Activating *set volume(V)* causes all notes encompassed by the scope in effect to assume the amplitude currently displayed opposite *volume(II)*. Therefore, to update certain notes, collect them within the current scope, adjust *volume(II)* to the desired level, and activate *set volume(V)*. As an aid to this process, you can select a notation which highlights the volume attribute of the notes. Simply activate *notation(I)* and select *Amp*. The shapes drawn for each note follow the note's loudness contour (or "envelope"). The height of the envelope with respect to its base line is directly proportional to its volume. Two examples of this form of notation are seen in the third row of Figure 1. Note that the last volume used remains opposite *volume(II)* and therefore affects subsequent notes specified one-by-one using *add(IV)*.

12. NAVIGATION

By now you should have encountered the fact that more notes can be specified than can be viewed at one time. Think of the score as being laid out on a time line, and the rectangle of the working area as a "window" looking onto a portion of it. For any part of the score to be viewed, it must be positioned so that it falls within the sight-lines of the window. This is easy to do. The upper border of the work space serves a special function: that of a time-line representing the total duration of the score (regardless of how long). That portion of the entire score which is currently visible in the window is represented by the thick bar running along the border. Thus, in Figure 1 only the first half of the score is shown, since the thick bar covers only the left half of the time-line. To view a different part of the score, point to its relative position on the time-line and quickly depress and release the cursor Z-button. This will cause the indicated material

APPENDIX B

to come into view, and the position of the thick bar on the time-line to be updated. (Clearly the presence of rehearsal marks or other reference points along the time-line would improve our ability to navigate through the score. Due to the prototype nature of the program, however, these have not been implemented.)

Often the window of the work area does not allow enough of the score to be seen at one time, or conversely, there is too much. We can adjust in this situation by changing the effective size of the window; that is, by adjusting how much of the score is looked on through the window. This is done by pointing at the relative position on the time-line which should appear at the left-hand border of the window. On depressing the Z-button and moving the cursor to the right *while holding the Z-button down*, the user can indicate the relative amount of score which is to fall into the window. The position on the time-line at which the Z-button is released indicates the point in the score which will appear at the right-hand border of the window. When specifying the window width, a horizontal line stretches from the point where the Z-button was depressed to the current cursor position. Once a new window width has been defined, it can be moved around over different parts of the score just as before. The techniques of moving the window and changing its size are much like panning and zooming with a movie camera. The difference with the zoom, however, is that it only works in one dimension: time. Draw a line along the full length of the time-line, and the entire score will appear in the window. (However, with large scores, the display will "run out of ink" before all notes can be notated, and the error message "GPAC ERROR: DISPLAY FILE FULL" will appear. When this happens, you might switch to *Amp* or *Roll* notation, both of which use less "ink" per note.)

In the same way that we can pan and zoom with respect to time, we can also pan and zoom with respect to frequency. This can be done when *display(I)* is set to *Linear*. In this case, the left-hand border of the work area represents the frequency range (currently linear scale, unfortunately) of 0 - 50,000 Hz. The thick bar along this border represents that frequency range currently falling within the view of the window. The labels on the frequency axis of the display indicate the current maximum and minimum frequencies. Verify that the bar controlling the window's frequency dimension works in the same way as that controlling time. As a result, unlike a movie camera, we have independent control over zooming in "x" and in "y". Finally, verify (using *play(IV)*) that even though no notes are currently visible in the window, they are still "there" in the editor.

13. TEMPORARY ESCAPE

There are many things which *scriva* cannot do. Furthermore, if we tried to build them all into the program (assuming this was possible), we would end up with a monolithic monster. A better solution, which takes into account the pragmatics of the real world, is to make it as easy as possible for you to access "outside" programs which fill in the gaps of *scriva*. In the middle of editing a score you may, for example, want to define a new object, or transpose some score file. You may just want to know who else is on the system, or what is the time of day. All of these things can be done from within *scriva* in exactly the same way as they would be normally carried out. Just type the command name, following its usage as described in the *SSSP Users' Manual*. During the period when you are working with the temporary program, the tracking symbol of *scriva* will indicate that you have "broken off" temporarily by assuming the shape of a broken egg shell. When you have finished, *scriva* will pick up from where you left off, and the regular tracking cross will reappear.

Finally, since there is only space for two typed lines in the command area, the *page(I)* button is provided. Read its documentation to see how the whole screen can be made available for typing.

14. SUMMARY OF LIGHT BUTTON FUNCTIONS

14.1 Column 1: EDITOR

14.1.1 General

This column contains commands which affect the editing environment, rather than editing functions *per se*. Aspects such as notation and other user-selectable modes of operation are included.

14.1.2 Notation

This button permits selection of the type of notation, or symbols, used in displaying notes. The notation currently in effect is identified to the right of the button. Alternative forms of notation can be selected by activating the button. This will cause a list of the available alternatives to appear in the lower left-hand corner of the work area. The alternatives are:

- **CMN:** a loose form of common music notation. The notation knows nothing about meter, bar lines, or beaming. Only durations which are multiples of a 32nd note currently can be notated. Pitches falling between notes of the chromatic scale, or durations which cannot be notated exactly are notated to the closest value.⁴
- **ROLL:** is a form of "time-line", or "piano-roll" notation. Notes are written as horizontal lines whose vertical positions indicate pitch, and whose length and horizontal positions indicate duration and timing, respectively. Notes of any duration can be notated accurately.
- **AMP:** notation highlights note amplitude characteristics. Notes are drawn as contours whose base-lines indicate pitch and duration (as with *roll* notation). The contour shape is determined by the amplitude envelope of the *object* with which the note is orchestrated. Whereas in the global view the vertical domain is pitch, in the micro sense the height of the contours from their base is proportional to the amplitude of the individual note.
- **OBJ:** notation highlights the timbral characteristics of the score as determined by orchestration with *objects*. The approach taken has much in common with a topographical map. Each object is represented by a graphic symbol, the meaning of which is given in a "key" in the lower right-hand corner of the work area. Each note is represented by the graphic symbol associated with its object. Each symbol is positioned such that it is centred on the starting point in pitch and time for the note in question. There is no indication of note duration in this notation. Notes which have not yet been orchestrated are notated by the symbol of a simple note head (a symbol which does not appear in the key). The graphic symbols are generated automatically by the system.

14.1.3 Display

This button is a "switch" which permits the user to choose between two formats of "manuscript paper": piano staff layout, or frequency/time free-form layout. The display option is independent of notation, as is seen in the first figure of this tutorial. There, examples of the four different types of notation are shown: in the left column on the "staves" display, and in the right column, on the "linear" display.

14.1.4 Input

This button allows the user to select the technique (or "tool") which is to be used when adding notes to the score as a result of activating the *add(IV)* button. Notes may be added note-by-note or score-by-score. *Notes specified one-by-one implicitly assume*

4. It is important to distinguish between notation and what is being notated. The music is not restricted by the notation's inability to transcribe it exactly. The notation does the best that it can, but the integrity of the values are maintained and are performed as composed, rather than as notated.

APPENDIX B

as attributes the values displayed in column two. Thus, notes added after setting *volume(II)* to 100 will have their volume set to 100.

On activating the *input(I)* button the user must select one of the input tools made available in the menu appearing in the lower left hand corner of the work area. This menu is shown in Figure 8. The input tools available are:

- **Ludwig:** allows the inputting notes and rests one-by-one using CMN. The tool uses a graphics-based approach of pointing at the desired pitch, and using a small moving menu to specify duration. Notes and rests can be tied in order to obtain any duration which is a multiple of a 32nd note. (Currently restricted to use with *staves display(I)*.)
- **Keys:** allows notes to be specified one-by-one by playing them on the clavier keyboard. (Not currently implemented.)
- **Draw:** allows notes to be specified one-by-one by hand drawing special symbols which are understood, or recognized, by the program. (Not currently implemented.)
- **Read:** allows the notes of a previously composed score to be added as a single unit to the score currently being edited. Any score file, regardless of how it was composed, can be added in this way, starting at any point in the current score.
- **Roll:** allows notes to be specified one-by-one by pointing at where in frequency and time the note should start, and "dragging" a horizontal line to the right from that point to indicate the desired duration. (Cannot currently be used with *staves display(I)*.)

A detailed description of the usage of the *Ludwig*, *Roll*, and *Read* input tools is given in the main body of this *scriva* tutorial.

14.1.5 Join

If we were to use *add(IV)* to insert some new notes into the middle of a score, an important question arises. What happens to the timing relationship between the two sides of the point where we started adding the new material? If the material to the right of this point was "pushed back" in time in order to make room for the new, we could say that the new material was *spliced* into the score. On the other hand, the new material could be *mixed* in, thereby retaining all previous temporal relationships.

A similar question arises when using the *delete(IV)* button. If we delete a note which is surrounded on either side by other score material, what happens to the temporal relationship between the material on either side? If, on deletion, the remaining material shifts in time in order to fill the gap left by the deleted note, then we can think of the deleted note as being *spliced* out of the score. On the other hand, if the deleted note is replaced by a rest of equivalent duration, thereby preserving the temporal relationship of the notes on either side, then one can consider the note to be *mixed* out of the score.

Join(I) allows the user to select between these two alternatives in both adding and deleting notes. It is a switch which is followed on the right hand side by an indication of the current mode: *mix* or *splice*. Activating the button causes the mode to switch. The current mode affects both *add(IV)* and *delete(IV)*.

14.1.6 Score

The name to the right of this button indicates the name currently associated with the score being edited. It is the name under which material will be filed when saved using the button *save(IV)*. If you have not explicitly specified a name, the program will use the default name "m.out".

To change the name, you need only activate the button. In response, the program will (a) post a list of all currently defined score names in the lower left-hand corner of the work area, and (b), post a picture of a terminal opposite the *score(I)* button. You may then specify a name by one of two techniques. Either you may select one of the existing names in the list, or you can type a new name (as prompted by the terminal icon). Names are selected from the list either by pointing with the cursor, or by "scrolling" through the list (using slider 1) until the name desired is positioned between the central horizontal lines, and then press the Z-button.

14.1.7 Key

This button sets the "key signature" which determines the accidentals when adding notes using the CMN *input(I)* mode. Only accidentals for notes outside the specified key are posted. A key is selected by activating the button and selecting one of the 12 (major) keys which appear in the lower left-hand corner of the work space. Note that changing key has no effect on pitches already defined. Only the notation is changed with the addition of accidentals according to the new key. The key signature is not posted on the staff.

14.1.8 MM

This button is used to display and permit the changing of the "metronome marking". Thus, it determines the tempo at which the current score is performed when the *play(IV)* button is activated. The value is specified in terms of quarter-notes per minute, as in common musical practice. Its default value is 60.

14.1.9 Page

In working with *scriva*, at any time that the tracking symbol is the regular cross ("+"), you may type commands as if you were not inside *scriva*. That is, you need not exit the program if in the middle of a session you want to modify a score file (using *retro* or *invert*, for example), or send a message to someone using *mail*. There is a problem, however, because only two lines at the bottom of the screen are reserved for typed messages. Therefore, if a command like *poobj* is called, there is not enough room for the requested text. To overcome this problem, one can activate the *page(I)* switch. This enables the whole screen to be used for text. (The text is superimposed over any graphics, but in most cases is still legible.) Activating the switch a second time clears the text from the screen and sets the text region back to the bottom two lines.

14.2 Column 2: MUSIC

14.2.1 General

The *input(I)* modes *Ludwig* and *Roll* enable the specification of pitch, entry time, and duration note-by-note. Each note entered, however, must also have an *object*, *volume*, and *output channel* associated with it. The values assumed for these parameters are determined by the buttons in this column. Note that notes added using the *input(I)* mode *Reud* already have these parameters specified, and are therefore unaffected by this column.

14.2.2 Object

The name opposite this button determines the name of the object associated with notes added note-by-note. The object name can be changed by activating the button. In response, the program will (a) post a list of all currently defined object names (if any) in the lower left-hand corner of the work area, and (b), post a picture of a terminal opposite the *object(II)* button. You may then specify a name by one of two techniques. Either you may select one of the existing names in the list, or you can type a new name (as prompted by the terminal icon). Names are selected from the list either by pointing with the cursor, or by "scrolling" through the list (using slider 1) until the name desired is positioned between the central horizontal lines, and then depressing

APPENDIX B

the Z-button. This name change will have no effect on notes already specified. Existing notes may be reorchestrated using the *orchestrate(V)* button. To view the current orchestration see the *Obj* mode *notation(I)*.

14.2.3 Volume

The number opposite this button determines the volume assigned to all notes added note-by-note. The range of legal values is 0 through 255, with 255 being the maximum. The value 190 is about *mf*, and a change of 20 is the equivalent to a change of about one dynamic marking. The value can be changed by activating the button and typing in a new value (as prompted by the terminal icon which appears opposite the button). Changing the value associated with the button will not affect notes already defined. These can be modified using the *set volume(V)* button. To view the current dynamics, see the *Amp* mode of *notation(I)*.

14.2.4 Channel

The number opposite this button determines the output channel assigned to all notes added note-by-note. The channels are numbered 1 through 4 counting clockwise from front left. The value can be changed by activating the button and typing in a new value (as prompted by the terminal icon which appears opposite). Changing the value associated with the button will not affect notes already defined.

14.3 Column 3: SCOPE

14.3.1 General

In using any of the operators in columns IV and V, it is extremely useful to be able to indicate which notes of the score are to be affected. For example, in using the command *play(IV)*, we may want to audition the entire score, a single note, a chord, a motif, or some other user-specified structural entity. The same goes for *orchestrate(V)*, and any other operator. The buttons in this column give the user quite a bit of flexibility in this regard. They allow the user to indicate which notes fall within the *scope* of (that is, are affected by) a command. There are three ways of associating note to fall within what is called the *current scope*. These correspond to the first three buttons in the column. The current scope is retained from command to command, and is only modified when explicitly changed by the user. Notes encompassed by the current scope are visually identifiable due to their being displayed at a higher intensity than other notes. Gathering notes into the current scope is cumulative. That is, repeated invocations of the first three buttons build up the current scope, rather than redefining the current scope from scratch. To start fresh in defining the current scope, the button *clear(III)* must be activated.

Often we want to affect just one note with a command. We do not want to have to redefine the current scope to be that note, nor do we want that note to be added to the current scope. For such situations, there is also the notion of what we call *immediate scope*. Immediate scope implies that a single indicated note only is to be affected by the next operator invoked. That note is specified by depressing the Z-button of the cursor while pointing at it. The program responds by drawing a triangle in front of the note which it understood to be selected. (See Figure 8.) This note will be the sole operand of *the very next operator only*. After the next button is activated, the triangle disappears, and the current scope comes back into affect. Again, immediate scope takes precedence over the current scope, current scope is unaffected by immediate scope, and immediate scope is in effect for one button activation only.

14.3.2 Whole Score

Activating this button indicates that the entire score constitutes the current scope. It is worth noting that when there is no current scope defined, both *play(IV)* and *save(IV)* assume the whole score as scope, thereby eliminating the need to explicitly activate this button in some circumstances.

14.3.3 Circle

On activating this button, the tracking symbol becomes a drawing of a quill pen indicating that the user may draw a circle around those notes which are to be included in the current scope. More than one circle may be drawn. Circles may be inside one another, but may not overlap. If a circle is inside a first circle, the notes in the inner circle are excluded from the scope. However, if a third circle is inside this second circle, then the notes circled by the third circle are excluded from the exclusion, and therefore included. This embedding can go on for ever (almost).

To draw a circle, activate *circle(III)*, position the cursor, and draw by moving the cursor with the Z-button depressed.

To draw a second circle, release the Z-button, reposition the cursor, depress the Z-button again, and draw as before. When all finished, depress button 3 on the cursor. Finally, if you make a mistake and want to restart drawing from scratch, depress cursor button 2, and start again.

14.3.4 Collect

After activating this button, new notes may be added to the current scope by "collecting" them one-by-one. This is done by pointing at the notes and depressing the cursor Z-button. On collection, each note will be redrawn at a brighter intensity. When finished, activate the button *end: collect(III)* which has temporarily replaced *collect(III)*. Note that if *scriva* cannot identify which note you are pointing at, it will automatically take you out of collect mode.

14.3.5 Clear

Activating this button will clear all notes from the current scope. This prepares the way for a new current scope to be built up from scratch:

14.4 Columns 4 & 5: OPERATORS

14.4.1 General

The buttons in these columns are the primary editing commands. In using them, it is important to be aware of how the individual commands relate to the current scope (see documentation for column III).

14.4.2 Add

This button is activated in order to add notes to the current score. The method by which notes are to be added is set by the current state of the *input(I)* button. Whether notes are to be spliced or mixed into the current score is indicated by the *Join(I)* button. Notes added one-by-one assume the attributes indicated by *Object(II)*, *Volume(II)*, and *Channel(II)*. If no notes have been specified as of yet, then scope has no relevance; otherwise, before invoking the *add(IV)* button, the user must indicate at what point he desires to begin adding notes. This is done by pointing at a note, as done in specifying immediate scope (see general comments on column III). Failure to do so will result in the error message: "YOU MUST IDENTIFY A REFERENCE NOTE".

14.4.3 Delete

Activating this button will cause all notes encompassed by the current scope to be deleted. Whether the time relationship among any remaining notes is affected by the deletion is determined by the *Join(I)* mode (splice or mix) currently in effect. On completion, the current scope becomes empty.

14.4.4 Play

Activating this button causes all notes encompassed by the current scope to be played. Rests are substituted for any notes not played, so the temporal relationship

APPENDIX B

among played notes is preserved (except that the performance starts right in on the first note of the scope and stops after the last note of the scope, thereby eliminating possible delays on either end). If no current scope is currently defined, the entire score is played. The performance can be interrupted at any time by hitting the key labelled "RUBOUT".

14.4.5 Save

Activating this button causes all notes of the current scope to be saved in a file on disk. The name of the file is that currently associated with the *Score(I)* button. Beware that saving a score under a particular name will cause anything previously saved under that name to be overwritten, and therefore lost. If the current scope is empty, then the whole score is saved. If the current scope only encompasses some of the notes of the score, then only part of the score is saved. After saving the data, *scriva* always outputs one of the following messages: "YOU HAVE SAVED ONLY PART OF YOUR SCORE", or "WHOLE SCORE SAVED". The score being edited is unaffected by the *save(IV)* button.

14.4.6 Orchestrate

This button allows previously defined notes to be reorchestrated. Notes are orchestrated by associating with them the name of a particular *object* (see the command *objed* in the *SSSP User's Manual*). Note that notes can be orchestrated with a particular object name before that object has even been defined. In such cases, the note will be played using a default timbre until such time as the named object has been created.

On activating the button, a list of all current objects appears in the bottom left-hand corner of the work area. Also, an icon of a terminal appears opposite the *object(II)* button. If there are notes currently encompassed by the current scope, selecting one of the names from the list, or typing an object name will cause all notes in the current scope to be orchestrated by the object of that name. (Names are selected from the list using the cursor, or by scrolling the desired name -- using slider 1 -- until it appears between the central horizontal lines, and then pressing the Z-button.)

Immediate scope (see the general discussion of column III) can also be used in orchestration. After activating *orchestrate(IV)* point to a note, then select an object name. Point to the next note, then the desired object name, *etc.*

At any time, the current orchestration can be heard using *play(IV)*. Remember, however, that the current scope is still in effect, and affects the performance as well as orchestration. One can, however, also change the scope. Orchestration is often facilitated by selecting *Obj mode notation(I)*. Finally, be aware that *object(II)* is set to be equal to the last object used in orchestration; so all subsequently added notes will assume that timbre.

14.4.7 Scorchestrate

This button is currently unimplemented.

14.4.8 Set Volume

Activating this button causes all notes encompassed by the current (or immediate) scope to have their volumes set to the value currently associated with the button *Volume(II)*. In setting volume, the *Amp mode notation(I)* is often a useful aid.

14.4.9 QUIT

Activating this button will allow the user to exit from *scriva*. As a safety catch to prevent loss of material due to accidental selection of this button, there is a second step. The program asks you to depress cursor button 2. If this is done, the program is terminated. Otherwise, such as if the Z-button is depressed, the exit is aborted, and

one resumes working.

APPENDIX C

APPENDIX C - A Tutorial Introduction to SCED

1. INTRODUCTION

Sced is a "score editor", that is, an interactive program for creating and modifying music scores, *via* directions specified from a computer terminal.⁵ *Sced* is called an "alphanumeric" score editor since both the commands which it understands, and the way in which it notates music, are in the form of alphanumeric characters. This is in contrast to other score editors, such as *scriva*, which are graphics based. Both types of editors have their advantages, and you should learn to know when to use one or the other.

Always remember: a score is a score is a score, no matter how it was created. Therefore you can edit scores created with other editors or composing programs using *sced*, and *vice versa*.

This tutorial is meant to simplify learning *sced*. The recommended way to learn the program is to read this document, simultaneously using *sced* to follow the examples. *Sced* is based on the UNIX text editor *ed*, so any experience with *ed* can be directly applied, and *vice versa*. Beyond this, experienced users are the best source of additional information. A summary of the *sced* commands is given as the final section to this tutorial.

Do the exercises! They cover material not covered in the actual text.

2. DISCLAIMER

This is an introduction and a tutorial. For this reason, no attempt has been made to cover the full facilities offered by *scsd* (although the most useful and frequently used features are presented). Also, basic UNIX procedures are not explained here. The reader is referred to the appropriate section of the *SSSP User's Manual*. It is assumed that you know how to log on, and that you have an idea as to what a score file is, and that you have read the section in the *SSSP User's Manual* explaining conventions used in typing and documentation.

3. GETTING STARTED

We will assume that you have logged into UNIX and that it has just said "%". The easiest way to invoke *sced* is to simply type its name:

```
sced
```

The program will respond by typing:

```
Type "h" for help.  
0  
s*:
```

You can ignore all of this for the time being, except to note that by typing out the prompt "s*", *sced* is indicating its willingness for you to tell it what to do.

4. SPECIFYING NOTES - the Append Command 'a'

As our first problem, suppose that we want to compose or transcribe -- from scratch -- the notes making up a score. We shall soon see how these notes can be played, printed, modified, *etc.*

5. *Sced* is directly based on the UNIX text editor *ed*, in order to facilitate user cross-over from one to the other. One advantage of this similarity is the ability to share documentation. This tutorial is based, with permission, on that for *ed* (Kernighan, 1973).

When *sced* is first started, it is like starting with a blank sheet — there is no score information present. This must be supplied by the user. For the time being we will assume that the notes are to be typed in, one-by-one.

First a bit of terminology. In *sced* jargon, the score being worked on is said to be "kept in a buffer." Think of this buffer as a work space, a scratch pad, or simply as the information which you are going to be editing. In effect, the buffer is like a piece of manuscript paper on which we compose material, then modify some of it, and which is finally *filed* away until a later work session.

The user tells *sced* what to do by typing instructions called "commands." As has already been stated, *sced* indicates its willingness to receive a command by issuing the "s*:" prompt. Most of the commands which may be typed by the user consist of a single letter, which must be typed in lower case. (As we progress, we shall see the introduction of certain multi-letter commands, as well as the use of qualifying prefixes and suffixes which extend the power of most commands.)

The first command is *append*, written as the letter

a

all by itself. It means "append (or add) notes to the buffer as I type them in." You will know that the program has understood the "a" command if it responds with:

s:

which is a prompt indicating that you can "splice" notes into the buffer.

We now need to examine the format in which a note's data is typed in. Let us take the example of a simple C major scale, starting on middle C. (Non-traditionalists: don't panic — we'll get to how to write "funny stuff" shortly.) To enter the scale, we need only type "a" (for append), and then enter the note data as follows:⁸

```
s*: a
s: c4
s: d
s: e
s: f
s: g
s: a
s: b
s: c5
s: .
s*:
```

There are several points to note in this example. First, the append command (the first line) must appear on a line by itself. Second, the data for each note must also be on a separate line. Third, the only way to stop appending is to type a line that contains a period, only (the last line of the example). The "." is used to tell *sced* that we have finished appending. *Sced* will then respond by issuing the "s*:" prompt indicating its willingness to again accept a command from the user. Note also that the "a" in the seventh line of the example is unambiguous. It is the note "a4", not the command "a", since no command can be issued after the original append until the "." is encountered. *Sced* is clearly expecting note data, not commands. This is unambiguous to the user as

8. In the example, the "s*:" and "s:" prompts are typed by the computer. They have been included in this example for clarity. In all subsequent examples, the prompts will be omitted. The reader should keep in mind, however, that they are always printed in the actual program.

APPENDIX C

well. When expecting commands, *sced* outputs the "s*:" prompt, whereas when appending it outputs "s:", which indicates that notes are being "spliced" onto the buffer.

Finally, notice that only the Cs have explicit octave markings (c4 for the first, c5 for the last).⁷ The other notes could have just as correctly been written "d4", "e4", etc.; however, a convention of the program is that when aspects of a note are left unspecified, some value is implicitly assumed. In the case of octave, the octave assumed is that of the last explicitly specified octave. Therefore, all notes in the example (except for the last) are in the 4th octave. (This ability to have note data assume values implicitly is one of the most important properties of the program. It makes the program both easier to learn, and to use. We shall see more of this as we progress.)

5. LISTENING TO A SCORE - the Listen Command 'l'

After we are finished with the append command, the buffer contains the eight notes of the scale. The "a" command and the "." are not there, as they are not note data. We can now play what we have written by using another command "l" (lower case L), for *listen*, or play. The way this is accomplished is by typing:

```
*l
```

Notice that the "l" command is prefixed by an asterisk character, which must be typed by the user. This means "play everything". The asterisk is not to be confused with the prompt "s*:". (We shall go into the "l" command in more detail shortly.)

At the end of the performance, the "l" command prints out the last note played. You will notice that there is more information there than you specified -- information such as duration, volume, and orchestration. For the time being just recognise that this information gets defined nevertheless. This is the phenomenon which we have already seen with regards to octave. Stay with us, and we will soon get to how to control all of these other aspects of the score.

To splice more notes to the end of our "score", we must type the "a" command again and begin entering note data. Thus, we could add two new notes to our score and play the new result by typing:

```
a
f4#
c
.
*l
```

The new notes will be "spliced" on to the end of the score in the buffer. Note that we had to specify the 4th octave for the f#; otherwise the octave would have been the 5th, since that was the octave of the last note entered. Note also that sharps are specified by the number sign "#", and flats are specified by lower case B, "b". The octave can range from 0 through 10, and the proper ordering (according to the Acoustical Society of America Standard) is pitch-class, octave, accidental.

6. ERROR MESSAGES

If at any time you make an error in the commands you type to *sced*, it will tell you by typing:

```
?
```

7. Note that c4 is interpreted as C in the 4th octave, that is, middle C.

This is about as cryptic as can be, but with practice (and some information that you do not yet possess) you can usually figure out how you goofed. Inside of the append command, however, the error messages are usually more helpful, and the program attempts to tell you where you went wrong. We will see more of this later on.

7. LEAVING sced - the Quit Command 'q'

To terminate a session with *sced*, type the command

q

which stands for *quit*. The program will then respond with the message:

Type "x" to exit.

This is a safety catch to keep you from leaving the program prematurely, and inadvertently losing the work which you have done. (A means of saving work from session to session will be introduced shortly.) If you are ready to exit, simply type

x

The system will respond with "%", and your work will vanish. If you type anything else besides "x", *sced* will resume functioning, leaving your work intact. You can then issue any command such as "l", "a", or "q".

EXERCISE 1:

Enter *sced* and create a simple score using:

a
[some notes]

Play the score, add some more notes using "a", and play the new results.

8. PRINTING THE BUFFER CONTENTS - the Print Command 'p'

To *print* the contents of the buffer (or parts of it) on the terminal, we use the command:

p

Since the buffer often contains a large number of notes, and we may only want to print a few of them, we will explain the use of "p" so as to allow you to control what is to be printed. To do so, consider the notes of the score in the buffer to be numbered from one on, starting at the beginning. Then, all we have to do is tell "p" -- by referring to note numbers -- where to begin, and where to end. This we do by typing the start number, a comma, the end number, and the command "p". Thus, to print the first two notes of the buffer (that is, notes one and two), we would type:

1,2p

If we were editing our original file "junk", *sced* would respond with:

NOTE	PITCH	FREQ	DUR	OBJECT	VOL	DELAY	CHAN
1:	c4	261	1/4	default_obj	190	1/4	1
2:	d4	293	1/4	default_obj	190	1/4	1

APPENDIX C

(Again, as noted before, there is more information printed out than we explicitly specified. Don't worry. We will get to it soon. For the time being, consider it a carrot being dangled in front of you to motivate you to push through the tutorial.)

Suppose we want to print *all* the notes in the buffer. If there were 10 notes, we could use

```
1,10p
```

However, in general we do not know exactly how many notes there are. *Sced* provides a shorthand symbol meaning "note number of the last note in the buffer." This is the dollar sign "\$". Use it this way:

```
1,$p
```

This will always print out every note in the buffer, regardless of the number.

Since you often see what you want before printing is finished (for example if you are looking for a particular part in the score), and since in such cases it is tiresome waiting for the printing to finish, *sced* provides a mechanism to interrupt the printing, and get back to editing. This is accomplished by hitting the key on your terminal labelled "RUBOUT" ("DELETE" or "DEL" on some terminals).

Returning to "p", to print the last note of the buffer, we could use:

```
,$p
```

but *sced* allows us to abbreviate this to:

```
$p
```

This can be generalized to say that if only a single number is specified (rather than both a begin *and* an end number), then the command will operate on one note only - the note whose number was specified. Thus, we can print the fifth note by typing:

```
5p
```

or the second note by typing:

```
2p
```

In fact, *sced* will allow us to abbreviate even further: as long as we specify the numbers of the note(s) to be printed, we can omit the "p". Thus, each of the following pairs of examples are equivalent:

```
1,$p  
1,$
```

```
5p  
5
```

```
$p  
$
```

```
5,7p  
5,7
```


EXERCISE 2:

As before, create a short score using the append command, and experiment with the "p" command. You will find, for example, that you cannot print note zero ("0"), or a line beyond the end of the buffer. Also, attempts to print the buffer in reverse order such as:

```
3,1p
```

do not work.

9. PLAYING THE SCORE - More on the 'l' Command

We have already seen how to play the entire buffer using the *listen* command "l". However, the command has the same type of flexibility as the "p" command, and very similar usage. Thus, typing:

```
3,6l
```

will play notes three through six, and both

```
*l
```

and

```
1,$l
```

will cause the entire buffer to be played. (The asterisk is seen to be a generally applicable abbreviation for "1,\$". Try it in combination with "p" to verify that it works there, as well.)

Finally, typing:

```
2l
```

would cause just the second note in the buffer to be played. Note again that when finished, "l" causes the last note played to be printed out on the terminal.

10. SCOPE - Consolidating a Concept

We have now seen how the effect of two commands ("p" and "l") can be limited to a specified sub-set of the buffer. This facility is an extremely important feature of *sced*, in that it allows us to address ourselves in the editing process to that part of the score which is of current concern. Thus, for example, if we are orchestrating a certain chord, we can listen to or examine that chord, and that chord alone. We are not restricted to working on the entire score, or individual notes. We can address ourselves to the score "chunk-by-chunk", where *sced* provides us with a means of defining what constitutes a "chunk". We refer to this feature as controlling the *scope* of commands. It is a feature which can be applied to many other commands besides "p" and "l", and we shall increasingly see how it enables us to express our desires in a concise and direct way.

11. THE CURRENT NOTE - 'Dot' or '.'

Suppose that our buffer contains the eight notes of a C major scale (starting on c4), and that we have just typed:

```
1,3p
```

APPENDIX C

After *sced* has printed out the three notes, try typing just

p

This will print:

NOTE	PITCH	FREQ	DUR	OBJECT	VOL	DELAY	CHAN
3:	e4	329	1/4	default_obj	190	1/4	1

which is the data of the third note in the buffer. In fact, it is the last (most recent) note that we have done anything with. (We just printed it in the previous example!) We can repeat this "p" command without line numbers, and it will continue to print just note three. The reason is that *sced* keeps track of the last note that we did anything to (in this case note three). It does this so that we do not always have to type an explicit note number when executing a command. We refer to this "most recently used" note as the *current note*. As an abbreviation for this current note we use the shorthand symbol ".", so we usually refer to the current note as *dot*.

Dot is a note number in the same way that "\$" is; it means "the current note", or loosely, "the note which we most recently did something to." We can use it in several ways. One possibility is to say:

..\$p

which will print out all notes from the current note to the last note, inclusive. In our case, these would be notes three through eight, and the following example would have exactly the same effect:

3,\$p

We have said generally that dot is defined as the last note that we did something to. Well, this is true, but in working with *sced* it is important to be continually aware of what effect a certain command will have on dot. If dot is intended as a convenience to allow things to be understood implicitly, it is important that both you and the program agree on what has been left unsaid! This is especially true since some commands have a slightly different effect on dot than others. Most cases are easy to remember, and "make sense"; but just in case you are ever confused, or not sure, you can always find out the current value of dot by typing:

.=

which following the previous example would respond with

8

since the last note printed was the last note in the buffer: number eight. While we're at it, we can also explain how to ask *sced* what it using for the value of the "\$" abbreviation. We use exactly the same mechanism of typing the abbreviation followed by the equals sign:

\$\$=

Note that this is also a simple way of finding out how many notes are in the buffer, something that can be determined by alternatively typing:

=

Now the last example, typing the equal sign to find out how many notes are in the buffer, is an example of an operation which does not change dot. As we said earlier, though, this is easy to remember, since it makes sense: the "=" did not really affect any note, so the last note affected has not changed. However, suppose that the current value of dot is three. What happens if we type the "l" command like this:

```
l
```

without any explicit scope? If "l" behaved like "p", it would just play the third note; however, that is not the case. As a result of comments made by users of *sced*, when "l" is typed alone, the program plays the current note, but then keeps on playing until either the end of the score, or until the RUBOUT key is depressed. In either case, following playing, dot is set to the last note played. If you just want to play the current note, that is easy. Just type:

```
.l
```

(If however, you want the "l" command to behave the same way as "p", you can tell *sced* by typing the command "lm", which means "switch the listening mode". You can always switch back by typing "lm" a second time.)

Let's look at some other examples. One frequent use of dot would be in combination like:

```
.+3p
```

or equivalently

```
.+3
```

which means "print the third note after dot". Having done so, *sced* will advance dot three notes, to correspond to the last note printed.

We can also type:

```
.-1l
```

which would play the note previous to dot (and consequently change dot as well). Or, we could type:

```
+.1
```

or simply

```
+1
```

which would advance dot one line and print it out. This particular operation, which is often used in "stepping through" scores note-by-note can be alternatively effected by typing a blank line: that is, just push the "RETURN" button without typing anything. This second way of advancing dot by one has the added attribute that it does not print out the header line, which lists the meaning of the note fields. This speeds things up, and follows the general axiom that "*sed* shalt not run off at the mouth."

At this stage, we can introduce yet another abbreviation: "&". This is useful when we want to set the scope to be "dot plus the twenty notes following," twenty being about the number of lines that can comfortably be viewed on a display. Thus,

APPENDIX C

```
&
.,+20
.,+20p
&p
```

are all equivalent (with one exception: if there are not 20 notes following dot then using "&" will not cause an error; rather, it will assume the value ".,\$").

SUMMARY OF SCOPE SYMBOLS

```
.      - the current note
$      - the last note in buffer
*      - same as "1,$"
&      - essentially the same as ".,+20"
.=     - print note number of dot
$=     - print number of last note
=      - print number of notes in buffer (same as $=)
RETURN - advance dot 1 note and print (same as +1p)
```

EXERCISE 3:

Use a combination of the "a", "p", and "l" commands and explore what effect different operations have on dot. What happens to dot when you hit "RUBOUT" while you are printing or playing? Use "l=" to see if sed's opinion of where dot should be corresponds with your own.

Note what happens if you change dot and then go into append. What is the relationship between dot and append? What happens if you try either of the following:

```
3a
```

or

```
3.5a
```

Once you figure that out, try and put a new note in front of the first note in the buffer. Try this:

```
0a
[new note]
.
1p
```

Try again using the insert command "i". (You add note data in "i" in the same way as with "a".) Try this:

```
1i
[new note]
.
1.2p
```

What then is the difference between "a" and "i"? Think of your answer in terms of dot.

12. MORE ON APPEND - Details on Notes

In specifying notes thus far, we have only explicitly defined pitch values, leaving all other attributes to assume values implicitly assigned by *sced*. We can, however, use the append command "a" to be as explicit as we want as to the characteristics of each note. The aspects of timbre, duration, loudness, rhythm, and pitch/frequency can all be specified explicitly. Furthermore, we shall see in a later section how these attributes can be easily changed or modified.

The more complete description of the line specifying a note using "a" could be summarized as follows:

```
[pitch/freq], [dur], [object], [vol], [delay], [channel]
```

where each item in square brackets represents the field (or position) where a particular note attribute can be specified. The fields are summarized as follows:

SUMMARY OF NOTE FIELDS

pitch/freq	- the pitch, or frequency
dur	- the note duration
object	- the note timbre, or instrument
vol	- the note volume, or loudness
delay	- entry delay: the delay from the start of the current note until the start of the next note. Effectively controls rhythm.
channel	- the output channel to which the sound of the note will be routed

As has already been seen, each field does not have to be specified. If we want to specify a note fully, we could type:

```
a
a4, 1/4, default_obj, 190; 1/4, 1
```

We will explain momentarily the precise meaning of each of the values specified. For the moment, let it suffice that what we have specified is a quarter note sine wave at concert A pitch, at about *mf*. Now if we only cared about the pitch and the channel number, we could have just as easily specified the note as:

```
a4,,,,,1
```

The fields left unspecified assume implicit values as has been seen already, but note that the commas are important in informing the program that the "1" refers to channel number. Of course, trailing commas convey no information, so they can be omitted as was seen in our very first example with the C major scale, or in the following example:

```
c3,,trumpet
```

where all we wanted to specify was the pitch and the timbre ("trumpet"). So, we can see that it is important to memorize the order of the parameter fields, and to recognize that attributes can be left unspecified, that commas are important, but that

APPENDIX C

trailing commas can be omitted. Now let's look at each parameter field and see how values are specified by the user, and how unspecified values are derived.

12.1 Pitch/Frequency

The highness or lowness of a note may be expressed in terms of either frequency or pitch. In specifying frequency, values must be given as positive integers within the range of 0 and 25000. These values are interpreted as Hz. If pitch is specified, values are specified according to Acoustical Society of America Standards, as outlined previously. The range is from octave 0 through 10, inclusive.⁸ If frequency is left unspecified, then that of the last note input is assumed. If the field is left blank with no previous value having been specified in the work session, then the value "a4" is assumed. If octave is left unspecified, then the octave of the last specified note is assumed. Again, if no explicit octave has been specified in the session, the 4th octave is assumed.

12.2 Duration

Duration is expressed in terms of whole notes, and fractional values may be used. Thus, the following example demonstrates specifying a C major scale with each successive note being half the duration of its predecessor. The first note has a duration of two whole notes...

```
a
c4, 2
d, 1
e, 1/2
f, 1/4
g, 1/8
a, 1/16
b, 1/32
c5, 1/64
```

The longest note which can be specified has a duration of 255 whole notes. The shortest legal note (aside from 0) has a duration of 1/255 whole notes. More generally, specified as a fraction, the numerator must be in the range of 0 to 255, and the denominator in the range of 1 to 255. Specified as an integer number of whole notes, the duration must be specified within the range of 0 to 255. Also, fractional values are automatically reduced to their simplest form by *sced*. Thus a duration specified as "2/4" will be printed out as the equivalent "1/2". If duration is left unspecified, the last explicitly specified duration is used. If no explicit duration has yet been given in the work session, a duration of "1/4" is assumed.

12.3 Object

The object parameter controls the timbre (or instrument) with which a note is orchestrated. For this field, the user is expected to specify to *sced* the name of an "object" file, such as one which would be defined using the program *objed*. (Note that *sced* does not allow you to define objects. It just lets you orchestrate notes with objects which you have defined, or intend to define.) If the name specified is that of an existing object, then all subsequent calls to the "l" command will result in that note being performed with the timbre of that object. If at the time of specification there exists no object of the name given, then *sced* will perform that note with a default timbre, a sine

8. In this document we shall use the terms pitch and frequency interchangeably. The 'p' command lists both the pitch and frequency of each note, but beware in defining pitch by Hz. that the pitch printed is that which most closely corresponds to the frequency. The pitch indication may, therefore, be up to half a semitone out.

wave; however, the moment that an object of that name is subsequently defined, then the note will be performed by that object. Thus, you can orchestrate before you have even defined your "orchestra"!

Finally, if the object is not specified; then the last explicitly specified object will be assumed. If there has not yet been an object explicitly specified, then the object "default_obj" (a sine wave) will be assumed.

12.4 Volume

Volume is specified as a positive integer within the range of 0 to 255. A marking of zero is inaudible, and 255 is the maximum. A volume setting of 190 is about "normal" (*mf*), and a change of volume of 15 to 20 results in a change of about one dynamic marking. Left unspecified, the last explicitly given volume is assumed. If no volume has yet been explicitly specified in the session, a value of "190" is assumed.

12.5 Delay

We now get into an important, but (for some) difficult, distinction. That is the difference between *duration* and *entry delay*. Duration (as controlled by parameter two), is simply the specification of how long a note will last once it has started. Entry delay is the length of the period between the start of the current note, and the start of the one which follows. (The interval between attack points or "attack-point rhythm".) Both are specified in the same way: as whole notes, or fractions of whole notes. Up until now, we have not had any confusion between the two, since we have been treating them as the same thing. One consequence of that is that each note specified thus far has started right at the end of the preceding note. Entry delay and duration have been the same, and consequently all of the resulting material has been made up of monophonic melodies.

What happens, then, if we want to play a chord (let us say a fourth chord on c4) using *scd*? All we have to do is specify the duration of the chord, the pitches, and ensure that there is no delay between the start of the notes making up the chord. Typing the following will do the job just fine:

```
a
c4, 1/2...0
f,,,0
bb,,,0
eb5
.
*t
*p
```

Defining, playing, and printing the chord as shown demonstrates several important points. First, we really do get a four voice chord. Second, the duration specified for the first note ("1/2") was implicitly carried over to the other notes. Third, we are reminded that lower case "B" represents "flat" (and "#" for "sharp"). Fourth, we see that notes specified as flats are printed as their enharmonic "sharp" equivalents. Finally, we notice that *unlike all other parameters*, when delay is not explicitly defined (as in the final note), it *does not* assume the delay of the previous note. Rather, it assumes a value equal to the duration of the current note! This presents somewhat of a bias towards melodic writing, but it will be seen that there are other mechanisms in *scd* to counterbalance this bias.

Before leaving delay, let us work through one other example. Let us assume that the previously defined fourth chord is still in the buffer. Let us follow it with an arpeggiated version of the same chord an octave lower. This we can do as follows:

APPENDIX C

```
a
c3, 1/2,,,1/32
f,,,1/32
bb,,,1/32
eb4
.
*p
*l
```

This example demonstrates once again (for good measure) that the delay must be explicitly specified if it is to be different than the duration.

12.6 Channel

The synthesizer has four audio output channels, numbered one through four. Any note may come out of one (and only one) of these outputs. (Fancy mixing is controlled by the distribution network which is not affected by *sced*.) Therefore, for this field the user specifies an integer in the range of one to four to indicate the output channel desired. If no channel has been explicitly specified during the session, channel "1" is assumed. Otherwise, the last explicitly defined channel is assumed.

12.7 Rests

With a little thought, it will be realized that rests can be specified by having notes whose entry delays are longer than their durations. Thus, each of the notes in the following three note melody are eighth notes, separated by eighth note rests:

```
c4, 1/8,,,1/4
e4,,,1/4
f#
```

Note that there is no rest after the final f#. Also, remember that the specifications of "1/4" control the time from the start of one event to the start of the next. Therefore, the duration of the rest is $1/4 - 1/8 = 1/8$, since the duration of the first note is $1/8$.

As a convenience to users, however, *sced* provides an alternative means of specifying rests when typing in note data. One simply types an "r" followed by the rest duration. Thus the next example has exactly the same effect as the previous one:

```
c4, 1/8
r1/8
e
r1/8
f#
```

Duplicate this example and print out the results. You will see that no entity is created which corresponds to a rest. Using "r" simply causes the value specified to be added to the entry delay of the preceding note. Thus, a little thought will explain why you cannot begin a score with a rest. Also, notice that there is no space between the "r" and the rest value. Finally, the duration of the rest must be specified. No implicit values are assumed. Also, the default note duration is *not* affected by the rest duration.

EXERCISE 4:

Now you know enough to let your creative urges go wild! Try and transcribe some simple melodies, using different dynamics and timbres. Try also to transcribe some homophonic (chord) progressions. How do you manage with two part counterpoint? Work on all of the above until you are fully comfortable with the "a" command. Frequently print

and play the material which you are generating to verify that you are getting what you think you are specifying.

13. SAVING SCORE FILES - The Write Command 'w'

We are now at a point where we know enough to compose material which is worth saving. This means expending some effort on administrative, rather than musical, concerns. Hopefully the musical potential of what has been learned thus far is sufficient motivation to persevere with this new material.

To save a score (the buffer contents), we must create a "file" which contains a copy of the work to be saved. (Files are the only means to save information. The moment you quit *sced*, the contents of the buffer itself are lost forever. You are, in effect, throwing your scratch pad into the garbage.) To write out the contents of the buffer into a file, we use the *write* command:

w

followed by the name under which we want the score filed. (This name being referred to henceforth as the "file name".) To save our score in a file named "junk", for example, type:

w junk

This will copy the buffer's contents into the the specified file ("junk"), thereby destroying any previous information which might have been filed under that name. Remember to leave a space between the "w" and the file name.

Sced will respond to the write command by printing the number of notes saved. If we were saving a major scale, *sced* would respond with

8

indicating that there were eight notes in the score saved.

Writing a file just makes a copy of the score -- the buffer's contents are not disturbed, so we can go on adding notes to it. This is an important point. *Sced* at all times works on a copy of a file, not the file itself. No change in the file takes place until you give a "w" command. (Writing out your score into a file from time to time as it is being created is a good idea, since if the system crashes or if you make some horrible mistake you will have a backup copy on file, even though the entire contents of the buffer may be lost.)

An important aspect of the "w" command is that it allows partial scores to be saved through the specification of scope. Thus, we could save the first three notes of a score independently in a file called "motif", as follows:

1.3w motif

In so doing, we have created a new score which has been "cloned" from the current buffer contents, but *without changing the buffer*.

EXERCISE 5:

Enter *sced* and create a simple score using:

a
[some notes]

APPENDIX C

Write it out using "w". Then leave *sced* using the "q" command. Print out your score and play it to verify that everything worked. Note, to print a score, use the command "pscore" whose usage is:

```
pscore <scorename>
```

and to play the score, use the command "play" whose usage is:

```
play <scorename>
```

Thus, with a score named "junk" you should type the commands:

```
pscore junk
```

```
play junk
```

(Remember that you can only type these commands in response to the "%" prompt by the system. For more detailed documentation on *pscore* and *play*, see the appropriate entries in the *SSSP Users Manual*.) Now try the same thing, this time experimenting with limiting the scope of the write command, thereby saving isolated parts of the buffer.

14. READING SCORES FROM A FILE'- the Edit Command 'e'

Besides typing using the "a" command, a common way to get score material into the buffer is to read it from an existing score file. This is what you do in order to edit a score which you have saved with the "w" command. The *edit* command "e" fetches the entire contents of a file, and puts it into a fresh buffer. (That is, regardless of what the buffer previously contained, after executing the "e" command, the buffer will contain the data of the score fetched, and only the data of the score fetched. The previous contents of the buffer are lost unless previously saved using "w".)

So, if we had saved our previously defined score "junk" (the 10 note version of a scale), the command:

```
e junk
```

(when called from within *sced*) would fetch the entire contents of the file into the buffer, and respond by printing:

```
10
```

which is the number of notes in the score "junk" which was read in. Remember *if anything was already in the buffer, it is deleted first!*

If we use the "e" command to read a file into the buffer, then we need not use a file name after a subsequent "w" command: *sced* remembers the last file name used in an "e" command, so if no explicit file is specified to the "w" command, then this is the file name assumed. An example of this would be as follows:

```
sced
e junk
[editing session]
w
q
```

where the "w" command saves the buffer under the file name "junk", even though it was

not explicitly specified to "w".

15. MORE ON NAMES - the File Command 'f'

We have just seen how the "e" command can cause the buffer to be implicitly associated with a particular file name for the purposes of saving the score using "w". At any time, you can find out the name of the file which *sced* is associating with the buffer. This you do using the *file* command "f". In our case, if we typed:

```
f
```

sced would reply:

```
junk
```

However, in cases -- such in our earliest examples -- where there is no name yet associated with the buffer, then typing the "f" will result in the reply

```
f?
```

which says that there is no file name known to *sced*.

In cases where no file name has yet been specified, it is important to note that the first (and only the first) time a name is given with the "w" command, that name becomes associated with the buffer.

Finally, verify that 'f' can be used to (re)set the name associated with the score being edited, when used as in the following example:

```
f lumpy
```

EXERCISE 6:

Experiment with the "e", "f", and "w" commands. Try reading, writing, and printing various files. You may get an error "?", typically because you spelled a file name wrong. Verify that

```
sced junk
```

is exactly equivalent to

```
sced
e junk
```

16. MIX, SPLICE AND THE JOIN MODE

In working through exercise four, it was probably noted that while melodic (and to a lesser degree chordal) structures were relatively easy to define, multi-voice counterpoint was not. This is due to the fact that you have to define in one stream (at the terminal) something which is essentially conceived as being in two or more streams. It would be far more "natural" from a musical standpoint to enter each voice separately, and have them "merge" or "line up" on their own. *Sced* allows us to do just that.

So far (using "a" and "i"), the notes which we have been adding have been spliced into the buffer. If, for example, we added a new note between two other notes, all notes following the first would be "pushed back" in time by an amount equivalent to the new note's entry delay. This is much like cutting a piece of magnetic recording tape and splicing in a new chunk of material. The fact that we have been splicing has always

APPENDIX C

been indicated to us by the "s" in the prompts "s*:" and "s:".

Instead of splicing, *scsd* will allow us to "mix" new material in with that previously defined. "Mix" and "splice" are the alternative *join modes*, and we can switch from one to the other by typing the command "jm". (Note that 'jm' may be typed while adding notes, as in 'a', or any time a command such as 'p' is legal.) Therefore, to add a new voice we need only go to that location in the existing material where the new voice is to start, go into input mode ("a" or "i"), switch the join mode to "mix", and begin adding notes. Let's take a simple example.: We have a C major triad ascending in quarter notes defined as follows (for purposes of clarity, prompts are shown):

```
%scsd junk
s*:a
s:c4, 1/4
s:e
s:g
s:.
s*:
```

Note that the buffer contains only the triad. We now add a second voice which consists of five notes descending the scale from G. We want the second voice to begin synchronously to the second note of the triad.: This is all done as follows:

```
s*:2a
s:jm
m:g, 1/8
m:f
m:e
m:d
m:c, 1/4
m:.
m*:
```

Note that we specified that we wanted to begin on the second note by preceeding "a" with a "2". We then switched to mix mode, as reflected by the 'm' in the subsequent prompts. Perhaps most important, we were able to define the new voice just in terms of itself. We could ignore the entry delays *etc.* of the previous voice. Nevertheless, if we look at the resulting score, we will see that *scsd* has been doing a bit of work keeping track of who gets played when. The new composite score resulting from the above example would print out as follows:

NOTE	PITCH	FREQ	DUR	OBJECT	VOL	DELAY	CHAN
1:	c4	261	1/4	default_obj	190	1/4	1
2:	e4	329	1/4	default_obj	190	0/1	1
3:	g4	391	1/8	default_obj	190	1/8	1
4:	f4	349	1/8	default_obj	190	1/8	1
5:	e4	329	1/8	default_obj	190	0/1	1
6:	g4	391	1/4	default_obj	190	1/8	1
7:	d4	293	1/8	default_obj	190	1/8	1
8:	c4	261	1/4	default_obj	190	1/4	1

The above example demonstrates one problem. While the use of mix and splice makes it easier to input voices, once this is done the new score is still hard to read. All voices are merged into one stream. While this is what finally happens at the ear, it still makes it difficult to see what is happening from the listing. Later, when we want to come in

and make changes, this increasingly becomes a bother. Be patient. We will soon see how some of the problem can be eliminated.

EXERCISE 7:

Work through the previous example to verify that it sounds as expected. Now try transcribing some examples of your own. Become as familiar as you can with mixing and splicing. How, for example, can you start the second voice half way between two notes of the previous one? Does the use of a "rest" help? Also, what happens when you switch from mix to splice "mid-stream"?

17. DELETING NOTES - the Delete Command 'd'

We have now become rather proficient in defining material. Now lets look at how to delete some of it. To do so we indicate which note, or notes, are to be deleted, and execute the *delete* command "d". Specification of scope with "d" is just the same as with "p". Thus,

d

alone causes dot to be deleted,

4d

would cause note four to be deleted, and both

*d

and

1,\$d

would cause the whole buffer to be deleted.

When "d" is used, dot is set to the note which followed the last note deleted. In the following example:

3,8d

dot would be set to what was previously the seventh note, but which is now (due to the deletion) the third note. If the last note deleted was the last note of the score, however, then dot is set to what becomes the new last note.

We still have one problem. What happens to any "hole" that appears in the music as the result of a deletion? If the mode is splice, then the deleted material is "cut out", and the material which follows shifts over to "fill in" the hole. If the mode is mix, the hole remains as a rest. All remaining notes maintain the same temporal relationships as they had before.

As a convenience, you can avoid using "jm" and set the mode in which something will be deleted by following the delete command with a "s" or a "d". This is shown in the following example:

1,3d s

Remember to leave the space between the "d" and the mode indicator. Also, the "d" and the mode indicator must appear on the same line.

APPENDIX C

Finally, to enable you to cut down on typing, there exists a compound version of "d" which prints out dot following the deletion. This is specified as "dp", and the following are a couple examples of its use:

```
3dp
```

and

```
4,$dp m
```

Note there is no space between the "d" and the "p".

EXERCISE 8:

Work through some examples where you delete notes in both mix and splice modes. After each deletion, use "p" and "l" to verify that the result was what you intended. Pay particular attention to deleting the note(s) at the end of the score, especially in mix mode. Play the score, then splice something to the end. Did you remember that you had an inaudible rest hanging on the end of the score?

18. ON-LINE HELP - the Help Command 'h'

At most any time in *scad*, if you are stuck, or want to know what your options are, you can use the *help* command, which is executed by typing

```
h
```

When *scad* is expecting commands to be input, typing "h" will give you a list of the possible commands, and a summary of their function. (This is the same summary which appears as the final section of this tutorial.) If you forget how to play a score, for example, type "h", read the list, and find out.

Often, however, you are in the middle of some special task (one which you may not use so frequently), and you forget what to do. Again, use the "h" command. It usually knows what you are trying to do, so the kind of information it gives you depends on the context in which it is called. Try typing "h" after entering *append* mode, for example. Among other things, the command will remind you of the order in which note parameters must be specified. Finally, if the *help* command doesn't help, then this tutorial or an experienced user are your only hope. Also, if you have found out what you want, but "h" is still running on, just hit RUBOUT to get back to where you were.

19. READING SCORES FROM A FILE - the Read Command 'r'

In contrast to the "e" command, sometimes we want to read a score file into the buffer without destroying anything that is already there. This would occur, for example, when we want to make a new score which is a combination of one or more previously defined scores. The way in which we do this is to use the *read* command, "r". For example, executing the command:

```
r junk
```

will enable us to read the file "junk" into the buffer without first destroying the previous contents. Notice, however, that before the file is read, the command asks for the mode of input: mix or splice. This it does by typing

```
mode:
```

You are expected to respond by typing an "s" or an "m".

So far so good, but where is the new material to be added to the buffer? In a manner consistent with other commands, this is done by indicating a single note number as scope. Thus in splice mode, typing

```
3r junk
```

will cause the notes of "junk" to be inserted between the third and fourth notes of the buffer. In the process, all notes previously following the third note in the buffer will be offset in time by an amount equivalent to the total of the entry delays in "junk". Note that the analogy of splicing in a chunk is appropriate, and that the contents of the file "junk" are unaffected by this process.

If we ran through the same example, but this time having specified mix mode, then the notes of "junk" would have been merged in with those in the buffer. The relative relationship among the notes in the buffer (or in "junk") is left intact. The two scores are aligned such that the first note from "junk" starts synchronously with the third note of the buffer.

If no explicit note number is specified with "r", then dot is first set to "\$". *This is a special case, and should be remembered!*

Like the "w" and the "e" commands, "r" prints the number of notes read in after the operation is complete. This is illustrated in the following example, where we assume that the file "junk" contains 10 notes:

```
e junk
10
r junk
mode: s
10
```

The above is an example of dot being automatically set to "\$" by the "r" command. The result is that the buffer contains two copies of the notes of "junk", in the form of a repeat. (Note that after the second copy of junk is read in that the number 10 is printed, not 20. Remember: the number indicates how many notes were read in, not the buffer size.) As a result of using the "r" command, the user who wanted the repeat has been saved from re-typing the entire section. This is a very powerful aspect of *sced*. If you use the "w" operator to create files of reoccurring germinal material which makes up your score, you can then use "r" to bring that material into the "master" score. You can, therefore, compose in terms of phrases, or structures, rather than just notes.

EXERCISE 9:

Write the subject for a simple canon using *sced*. Save it using "w", and then copy it back on top of itself with staggered entries, thereby realizing the canon. Do you use mix or splice with "r" in this case? Repeat this exercise with a new canon and subject. Now, make a new score which has an A-B-A structure, where the "A" structure is canon one, and "B" canon two. Throughout, the only notes which should be typed in are those of the original subjects of the canons. Finally, change your structure to be B-A-B-A by inserting a copy of the B section in front of the previous ternary version. Try this using:

```
Or B
mode: s
```

where we assume "B" is the name of the file containing the second canon. You should be able to carry out the whole exercise without once quitting *sced*.

APPENDIX C

20. CONTROLLING TEMPO - the Metronome Command 'mm'

By now you may be wondering how to change the tempo at which a score is performed. Since we have defined all time values relative to a whole note, this is rather easy. We need only change the metronome marking given to the "I" command.

When you enter `scsd`, the metronome marking is set to quarter note = 60; that is, there are 60 quarter notes a minute. You can verify this with a watch, or by typing

```
mm
```

which will respond by typing out the current metronome marking. To change the metronome marking, you need only type "mm" followed by the new marking. Typing

```
mm 120
```

will cause the tempo to double, and

```
mm 30
```

will cause it to be half of the original 60.

The metronome marking must be specified as a positive integer, and it is important to realize that only one metronome marking can be specified for a given performance: no changes are allowed midway through.

Finally, speeding up the metronome marking in combination with the "I" command is a good way to find a particular place in a score. It is much like dragging a tape across the heads of a tape recorder in fast-forward. When you get to the place you want, just hit RUBOUT, and `scsd` will set dot to equal the last note played. You are now right around where you want to be.

21. MODIFYING ATTRIBUTES OF PREVIOUSLY SPECIFIED EVENTS

The attributes of a previously specified event, or note, can be changed using one of two approaches. First, the entire event can be re-specified using the "change", or "c", operator. Alternatively, any single attribute of an event (such as frequency, volume, or object) can be individually re-defined, using one of the "set" operators. The former case requires the entire event to be re-specified. The latter case requires only the individual parameter to be changed. Both are described below.

21.1 The Change Command 'c'

In effect, the change command is a combination of two operators encountered previously: delete ("d") and append ("a"). Used alone (without any specific scope), "c" causes the current note to be deleted, and then automatically sets the program into "input" mode (that is, the program then expects you to input event data in the same way as done in "a"). The desire for input is indicated by the program through its issuing one of the following prompts:

```
s:
```

or

```
m:
```

which indicate that the input mode is splice or mix, respectively.

The user may now input the event data to replace that of the event being changed. The replacement data may be made up of more than one event. One may continue to input

just as with the "a" or "i" operators. (Remember that at any time the join mode may be changed to mix or splice, or help may be requested, by typing "jm", or "h", respectively.)

Like virtually all the operators in *sced*, the scope of "c" can be constrained to affect a limited set of events. Thus, the line

```
3,8c
```

would cause events 3 through 8 to be deleted, and replaced by the user specified input.

21.2 The set Commands

The set commands allow specific attributes of previously defined events to be individually modified. The names and usage of the set operators are similar to commands available outside of *sced*, as part of the SSSP system. This is as a memory aid to the user. For users desiring to work in as terse a mode as possible, abbreviations for the command names are understood by the system. (The novice user should probably ignore this feature until more familiar with the system.) The operators and their abbreviations are summarized in the following table:

OPERATOR	ABBREVIATIONS
setfreq	sf, sfreq, setpitch, sp, spitch
setdur	sdur
sctobj	so, sobj, orch
setvol	sv, svol
setdel	sdel
setchan	sc, sochan
settime	st, stime

Figure 1. Summary of set operators

All of these operators are sensitive to the constraints imposed by the specification of scope. If no explicit scope is specified, the commands affect only the current event.

Beware of confusing the *set* operators in *sced* with those which are generally available outside of *sced*. Functionally, they are essentially the same; however, the operators in *sced* affect the score currently being edited, *not* a stored score. Therefore, outside of *sced*, the name of the score to be transformed must be explicitly specified. Inside of *sced* it is implicit. The difference is illustrated in the following two examples of the use of *setvol* (note that the prompt character is printed in these two examples only.)

```
%setvol minuet 190
```

and

```
*1,5setvol 190
```

The second example, the one from *sced*, emphasizes one key advantage of the internal version: the ability to impose scope on the operator.

There is one more general point to make before moving on to a detailed discussion of the individual *set* operators. This concerns the alternative available in most *set* commands to either (a) set an attribute of an event (such as its volume) to a specific, absolute value, or (b) set the attribute to a new value which has some specified relationship

APPENDIX C

relative to the previous value (such as "double the frequency"). From here on, the former will be referred to as *absolute mode*, and the latter as *relative mode*. These and other details are expanded upon in the discussion of the individual commands.

21.2.1 setfreq

This operator enables the respecification of the frequency/pitch attribute of one or more events, in either absolute or relative terms. It will permit values to be specified in terms of frequency (Hz, or cycles per second) or pitch (using ASA pitch notation, *eg.*, a4#), shifting up or down a fixed number of either cycles per second or semitones, or transposition up or down by arbitrary degrees.

In *absolute mode* the usage of the operator is:

```
[scope]setfreq <value>
```

where the scope is optional, and the value is specified in either cycles per second (Hz), or in ASA pitch notation. Hz are expressed as unsigned integers within the range of 0 to 25000. ASA pitch notation is expressed as a pitch class (*eg.*, "g" or "a") followed by an octave indicator (an integer), followed by any accidental. An example would be "c4#", which stands for middle c sharp. If the octave specification is omitted, the octave of the last event previously input is assumed. If the accidental is omitted, the event is assumed to be natural. Sharps are expressed by the number sign "#", and flats by lower case B "b". The concepts of double accidentals or key signature are unknown to the program.

Two examples of the use of absolute mode are as follows:

```
setfreq 440
```

```
setfreq a4
```

Both examples have the same effect: changing the pitch of the current event to concert A.

In *relative mode*, there are two different ways in which *setfreq* can be used. First, it can be used to *shift* the event up or down a fixed number of either cycles per second or semitones. Second, it can be used to *scale* the existing pitch by a specified factor.

For purposes of *shifting* frequency/pitch relative to the existing value, the usage of the command is expanded as follows:

```
[scope]setfreq <[sign]value[s]>
```

The sign preceding the value is either a "+" or a "-", indicating the direction of the shift is either up or down, respectively. When prefixed by a sign, the value may be an integer *only*. Suffixing the value with the optional "s" indicates that the value specified is the number of semitones that the pitch is to be shifted, or transposed, up or down. If the "s" is absent, then the value is interpreted as being the number of Hz that the frequency is to be shifted.

Shifting is illustrated in the following two examples:

```
setfreq +12
```

```
3,$setfreq -12s
```

In the first example, the frequency of the current note is shifted up 12 Hz. In the second example, the pitch of all events in the score, from the third event on (due to

the "\$" in the scope), are to be transposed down 12 semitones. In the examples, note the following points: there is no space between the sign and the value; there is no space between the value and the "s"; the value must be an integer, as it makes no sense to say something like "+c4s". Note also that shifting by semitones does not imply that either the original or the resulting pitch must correspond to one of the notes of the chromatic scale. For example, if the original note was half-way between c and c# and it was shifted up one semitone, the new result would fall between c# and d.

By *scaling* frequency attributes relative to their previous values, we mean that the previous value is multiplied or divided by some user-specified factor. In this case, the usage of the command is as follows:

```
[scope]setfreq <[op]value>
```

The optional "op" parameter which prefixes the value is either a "*" or "/" character indicating, respectively, that the frequency is to be multiplied or divided by the value. If either the "*" or "/" characters are present, then the value may be specified as either an integer (eg., "3") or as a real number (i.e. as a fixed-point fraction such as "1.5" or ".333").

Our first example of scaling:

```
3,$setfreq /2
```

has exactly the same effect as the last seen example:

```
3,$setfreq -12s
```

since dividing the frequency by 2 is the same as transposing down 12 semitones, the pitch is transposed down an octave. The same result can be achieved by yet another means, as shown in the next example:

```
3,$setfreq *.5
```

Note the use of a fractional value in this example. Note also that there must be no space between the "*" or "/" characters and the value. Finally, note that it is an error, and makes no sense, to follow the value by an "s" when it is preceded by either a "*" or a "/" character.

The usage of the *setfreq* operator is summarized in the following table:

MODE	PREFIX	VALUE	SUFFIX
absolute	none	Hz. or ASA Pitch	none
rel. shift	+ or -	Hz. or semitones	s
rel. scale	* or /	scale factor	N/A

Figure 2. Summary of setfreq usage.

21.2.2 setdur

This operator allows the duration of any one or more events to be set in either absolute or relative terms. In changing a note's duration, however, it is important to remember that neither tempo nor rhythm are really affected; rather, changing a note's duration is a question of articulation, making the note more staccato or legato. Tempo

APPENDIX C

and rhythmic characteristics are controlled using the command *setdel*, described following *setdur*. The user concerned with questions of time should also see the command *settime*. In *absolute* mode, the usage of *setdur* is as follows:

```
[scope]setdur <value>
```

where the scope is optional, and the value indicates the duration to which the event is to be set. The duration is specified just as with the "a" command. That is durations are specified in terms of whole notes: either fractions (eg., "1/2", "2/3", "3/2"), or as integers. Remember that neither the numerator or denominator of the fraction may exceed the value 255. Also, the concept of a negative duration is unknown, and therefore not permitted.

Examples of the use of absolute mode are as follows:

```
setdur 1/6
```

which would have the effect of triplet quarter notes, and

```
setdur 2/6
```

which would have the effect of triplet half-notes. Note that both examples would affect only the current note since no explicit scope was specified. Note also, that if the event affected by the second example were printed out, the duration would appear as "1/3", since the program always reduces fractions to their simplest terms. Finally, if there is confusion as to why a duration of "1/6" would result in a triplet quarter note, think for a moment and it will make sense: the fraction being expressed is a fraction of a whole note, not a fraction of a beat.

In *relative* mode, one can either add a constant offset to one or more durations, or scale them by a specified factor. To change durations by a specified offset, usage of the command is:

```
[scope]setdur <[sign]value>
```

where the scope is optional, and the value is the magnitude of the constant offset to be added to the durations. This offset is specified in exactly the same way as durations in absolute mode. The presence of the optional sign indicates that the value is, in fact, an offset. If the sign is a "+", the offset is added to the durations. If it is a "-", the value of the offset is subtracted from the durations. The following is an example of adding the duration of an eighth note to all notes of a score:

```
*sdur +1/8
```

This would result in the score being performed slightly more legato. Note that there must be no space between the sign and the value.

If the command is used to scale durations, its usage is:

```
[scope]setdur <[op]value>
```

where the scope is optional, and the optional "op" argument is either a "*" or "/" character. When the optional "op" character is present, the value is interpreted as a scaling factor. When the "*" character is present, all durations in the current scope are multiplied by the value. When the "/" character is present, they are divided by it. There must be no blank between the "op" character and the value. The value itself may be expressed as an integer, or as a real number (a fixed-point fraction). It may *not* be

expressed as a fraction.

The following example shows how all durations from dot to the end of the buffer could be halved:

```
..$setdur *.5
```

or

```
..$sdur /2
```

both of which are equivalent.

One point to note: due to the existence of both `setdur` and `setdel`, the abbreviation "sd" is ambiguous, and therefore not allowed.

21.2.3 setobj

This command permits the user to "orchestrate" one or more events with a particular timbre, or "object". Unlike most set commands, `setobj` functions in absolute mode only.

Usage of the operator is as follows:

```
[scope]setobj <name>
```

where the scope is optional, and the name identifies the object which is to be associated with that event. An example of the use of the operator is:

```
..9setobj trumpet2
```

which would set all notes from the current event to event 9 to be orchestrated with the object "trumpet2".

In orchestrating events, a few important points should be kept in mind. First, an event can be orchestrated with an object even before that object exists. The user need only specify the name (as the name used in the `setobj` command) that the object will be given when it is created. In the meantime, the system will play that note with some default timbre, such as a sine wave. At the moment that the new object is defined, however, the program will use it automatically when the event is played.

In naming objects, remember that an object name may *not* begin with a digit (0-9), the name must not have more than 13 characters in total, and that there must be no blanks in the name. As the last example demonstrated, however, the name may include digits anywhere but the first character.

21.2.4 setvol

This command allows the volume attribute of events to be set to some absolute value, or changed by some specified amount. Volume settings are specified as integers in the range of 0 to 255, where 0 is off, 255 is the maximum, and about 190 is normal.

In *absolute mode*, the usage is:

```
[scope]setvol <setting>
```

where the scope specification is optional, and the volume setting must be an *unsigned* integer (no "+" or "-" signs) within the range of 0 to 255. To illustrate, the following two examples:

```
setvol 100
```

APPENDIX C

```
.setvol 100
```

have the equivalent effect of setting the volume of the current event to 100. The next example:

```
5,9setvol 200
```

would have the effect of setting the volumes of notes 5 through 9 to 200.

Relative mode provides a means of raising or lowering the volume of notes by a specified amount. The magnitude of the change is prefaced by a '-' or '+' sign, indicating the direction of the change.

The use of relative mode is illustrated in the following example:

```
setvol +20
```

which would increase the volume of the current event by 20 units (about one dynamic marking). The next example:

```
*setvol -30
```

illustrates the use of relative mode to lower the volume of all events in a score (by virtue of the specification of "*" for the scope), by 30 units.

One final point, note that if an attempt is made to increase the volume of an event above the maximum (such as increasing an existing setting of 240 by 30), the value will be set to the maximum of 255. Similarly, values falling below 0 will be set to 0. The effect can be to lose the dynamic variation of the score, due to this "flattening-out" of the dynamics.

21.2.5 setdel

setdel permits exactly the same operations on entry delays as *setdur* permits on durations. That is, delays can be set to some absolute value, or changed to some new value relative to their current one. In relative mode, values may be changed by a constant offset, or scaled by some specified factor. Again, the usage is exactly as with *setdur*.

21.2.6 setchan

This operator allows an event to be assigned to a particular audio output channel in either absolute or relative terms.

In *absolute* mode, the usage of the operator is:

```
[scope]setchan <value>
```

where the scope is optional and the value indicates the output channel. The value must be expressed as an integer within the range of 1 to 4. An example of the command's usage is:

```
setchan 3
```

which will cause the audio output of the current event to be routed to channel three.

In *relative* mode, one can specify that the output of an event is to be shifted to some other channel, relative to the previously specified one. In this case, usage of the command is as follows:

```
[scope]setchan <[sign]value>
```

where the optional sign is either a "+" or "-" character, and when prefixed by a sign, the value is interpreted as an offset indicating the magnitude of the shift. For example, if the channel assignment of the current event was 2, then the following command:

```
setchan +2
```

would cause it to be changed to 4 (i.e. 2+2). Alternatively, the command

```
setchan -1
```

would have caused the channel to be set to 1 (2-1).

If the value resulting from combining the offset and the previous value results in a new value falling outside the range of 1 to 4, then the new value "wraps around" in order to stay in range. Thus, if the current channel was 3, then all of the following examples would have the same effect of setting it to 1:

```
setchan -2
setchan +2
setchan -6
setchan 1
```

As with commands previously seen, the absence or presence of a sign prefixing the value indicates absolute or relative respectively. Again, remember that there must not be a space between the sign and the value.

21.2.7 *settime*

Settime is a combination of *setdur* and *setdel*. It allows transformations to affect both aspects of time in the same way at the same time. The usage is exactly the same as *setdur* and *setdel*. In absolute mode, both delay and duration assume the same specified value. In relative mode, both delay and duration have the same offset added to them, or are scaled by the same amount.

EXERCISE 10:

We have just swallowed a rather large amount. Before going further, spend some time experimenting with the *set* commands. Take the B-A-B-A structure from the last exercise. Make the second instance of B much softer and more legato. Make the first version of A staccato, and double the tempo of the first. Also, transpose the second version of A up a perfect fifth. Finally, having listened to the effect of each change along the way, now "detune" the second half of the second B section by adding a constant, say 15 Hz, to all frequencies. Make up your own examples, and exercise on these commands until you are comfortable with them.

22. MORE ON SCOPE: Conditionals

Thus far, note number has been the only criterion according to which the scope of an operator has been specified. Often, however, we want to address ourselves to score data characterized by other properties. For example, we may want to play all notes below a certain pitch, or reorchestrate all notes longer than a quarter note which are currently orchestrated by some object such as "trumpet". To do so, we must have a means of unambiguously describing to sced the characteristics identifying those notes which we want affected by an operator. The semantics of doing so are described below.

The notes that we want affected by an operator can be described in terms of any attribute (or combination of attributes), including its pitch, volume, object, duration, entry

APPENDIX-G

delay, note number, or channel. In effect, we can say "all notes having a pitch of C4#", or "all notes with a loudness between 90 and 120 which are coming out of channel 3". To do so, we use an algebraic notation which is more terse and precise than English. The notation makes use of: (a) a set of "key words" representing the various note attributes; (b) parameter values; (c) various "relations"; (d) miscellaneous punctuation. The keywords are given below. The note attribute associated with each is obvious. (Note that for each keyword there are various abbreviations which are permitted.)

KEYWORD	ABBREVIATIONS
frequency	freq, f, pitch, p
duration	dur
object	obj, o
volume	vol, v
delay	del
channel	chan, c
notenumber	number, n, #

The specification of characteristics: defining the conditional scope must appear between curly brackets - "{" and "}" - immediately preceding the operator. Thus, if we accept the "=" character as being the relation "equal to", then

{freq = 100}orch trumpet

would have the effect of taking all notes having a frequency of 100 and orchestrating them with "trumpet". Other relations besides "equals" can be used in such expressions. The legal options are given below, along with their interpretation.

RELATION	MEANING
=	equal to
<	less than
>	greater than
>=	greater than or equal to
<=	less than or equal to
!=	not equal to
!	not (unary negation operator)

(In two character relations the order of the characters is unimportant. Thus "<=" is the same as "<".) The following examples demonstrate the usage of some of these operators:

{obj != fred}l	- play all notes not orchestrated with "fred"
{freq < c4}p	- print all notes having a pitch lower than c4
{dur != 1/4}d	- delete all notes which are not quarter notes
{# = 4}p	- print note 4 (which is the same as "4p")

Obviously it makes no sense to talk about objects being "greater" or "less" than each other, and

{obj < fred}d

would be an error.

Often, the notes which we want affected by some operator cannot be fully described in such simple terms. We can, therefore, make compound relations which are made up of simple relations, such as those already seen, connected by the conjunctions "and", "or", and "exclusive or". The symbols used for these conjunctions are:

CONJUNCTION	MEANING
	or
&	and
~	exclusive or (one or the other, but not both)

Thus,

```
{freq > 100 & obj = sax}setchan 3
```

would cause all notes which had both a frequency above 100 and which were orchestrated with "sax" to have their channel set to three.

In complex compound statements, it is often useful to parenthesise clauses in order to clarify the intended order of operations. This is seen in the example:

```
{{(freq < 100 & obj = sax) | (freq > 100 & obj = flute)}}l
```

which could also have been written:

```
{{{(freq<100) & (obj=sax)} | {(freq>100) & (obj=flute)}}}l
```

Both would play all "saxes" below 100 Hz., and all "flutes" above 100 Hz. This is because the scope encompasses all notes complying with either the first or second main clauses of the expression. But think about the difference of meaning due to the use of parentheses in the following two examples.

```
{freq=100 & (obj=sax | obj=flute)}d
```

```
{{(freq=100 & obj=sax) | obj=flute}d
```

Sometimes it is desirable to describe the notes to be encompassed in the current scope in terms of some parameter being equal to one in a list of specified values. For example, we could express "print all notes having a duration of a quarter, eighth, or half note" as follows:

```
{dur = 1/4, 1/8, 1/2}p
```

Each alternative in the list must be separated by a comma. Such use of lists can also be used in compound expressions. For example,

```
{{(vol < 100) & (obj = flute, trumpet, sax)}}l
```

which would play all notes having a volume below 100, and which were also orchestrated with one of the objects "flute", "trumpet", or "sax".

So far, we have been specifying the scope in terms of who is to be included. Often, however, it is easier to identify which notes are *not* to be affected by an operator. To do so we employ the unary operator "!" which stands for "not". Its use is seen in the

APPENDIX C

example:

```
{!(freq=100)}d
```

which would delete all notes whose frequency was not equal to 100. This example is rather simple, and the same result could have been obtained by:

```
{freq!=100}d
```

The real power of the negation operator comes in compound conditions such as:

```
{!(freq < 100 & obj = sax)}p
```

which would print all notes except for those who were both below 100 Hz. and orchestrated with "sax". There are other ways of specifying the same condition with our notation, but using the "not" operator is probably the most clear.

Finally, notice that the conditional method of specifying scope allows us to express exactly the same concepts as with the previously used unconditional scope. For example, the following two expressions are functionally identical:

```
3,6d
```

and

```
{# >= 3 & # <= 6}d
```

Clearly, the former is more efficient; however, the conditional form is more flexible as is seen in the following example which uses the list feature:

```
{# = 1,3,5,7,9}l
```

which plays the first five odd-numbered notes. One important point to note is that the two notations for scope can be combined. For example:

```
4,9{obj=weird}p
```

would print all notes orchestrated with the object "weird" between notes four and nine. This example illustrates the requirement that the conditional part of the scope must immediately precede the operator.

Finally, one labour-saving feature of `scsd` is that it remembers the last scope specification so that it can be reused, without being retyped. We do this by leaving the contents between the curly brackets blank. Thus, the following two commands affect the same notes:

```
{freq = c4 & dur > 1/4}orch flute  
{setvol -20
```

23. SEARCHING

From what we have already seen, we know that we can print out all notes conforming to a particular set of characteristics by using scope and the 'p' command. Often, however, we just want to find the next note which conforms. We can request this by using slashes (/) to parenthesize the characteristics, rather than curly brackets. Thus,

```
/freq = c4/
```

will cause the next note of pitch c4 to be printed. As with regular scope, the pattern between the slashes is "remembered", so

```
//
```

will find the next c4 after that, and

```
{orch flute
```

will cause all c4's to be orchestrated with "flute". (The point here is that the same "memory" is used for both scope and the search pattern: they are interchangeable.)

Finally, there is a notation for finding the last *previous* note which fits the search pattern. In this case, question marks (?) are used in place of the slashes. An example would be:

```
?dur < 1/32?
```

Experiment with forward and backward searching until you are comfortable with their use.

24. ESCAPING TO THE SHELL - the '!' Command

Often when working with *scsd*, you will want to leave the program in order to do something at the terminal, with the intention of picking-up where you left off once you are done. You may want to send mail concerning some problem, execute some other program in the SSSP system, or simply find out the time with the *date* command. In any case, *scsd* provides a means of temporary escape which permits you to do all of these things without ever leaving the program. This is done using the "!" command, which we call the *shell* operator.

At any time when *scsd* will accept a command (such as "a"), you can type the command "!" followed by the name of the program which you want to execute. Typing

```
!ls
```

for example, would cause the current contents of our directory of files to print out on the screen. Thus, if we can't remember the name of a particular score, or its spelling, here is a way to find out.

When the requested command has completed its task, control returns to *scsd*, and to indicate this fact, the program types out another "!" character. Thus, a complete listing of the transaction described above would be:

```
!ls
fred
junk
masterpiece
exercise1
trumpet
!
```

In the example, it was assumed that there were five files in the directory.

APPENDIX C

25. MOVING NOTES AROUND - the Move Command 'm'

The *move* command "m" is used for "cutting" and "pasting" the notes within a score. It lets you move a group of notes from one place in the buffer to another. If we wanted to cut the second to sixth notes of the score out of the buffer, and move them so that they will be spliced onto the end, we could do so as follows:

```
2,6w temp
2,6d s
$r temp
mode: s
```

(Do you see why?) We can do the same thing far more easily, however, as follows:

```
2,6m$
```

The usage of the command can be summarized as:

```
[scope]m[destination]
```

where the scope specifies what is to be moved, the "m" the fact that it is to be moved, and the destination says to where it is to be moved to. Scope is specified as with "p", or the other commands seen. When omitted, dot is assumed. The destination is specified as a single note number. When omitted, dot is assumed.

Examine the next example:

```
3,6m0
out mode: s
in mode: s
4
```

Notice that there is no blank between the "m" and the destination. Also, the value "0" is a legal destination, indicating that the material in question is to be moved in front of the first note of the score. The issue of mixing and splicing now arises once again. First, are the notes being moved extracted from their previous position in mix or splice mode? Second, is the material mixed or spliced into its new position? The response to "out mode" answers the first question, and the response to "in mode" answers the second. Either can be answered by "s" or "m". Finally, notice that dot is set to the new position of the last note of the group moved. (We moved four notes and placed them at the front of the score. Dot is therefore four.) This can be verified with the "p" and "l" commands.

26. COPYING SCORE MATERIAL - the Copy Command 't'

Sometimes we want to repeat a section of material at a different point in the score. We have already seen how we can do so using a combination of "w" and "r". We can, however, do so in a more direct manner using the *copy* command "t" (sorry - "c" is already used for change). Usage of the command is similar to the "m" command:

```
[scope]t[destination]
```

The scope says what is to be copied, the "t" says that it is to be copied, and the destination says where it is to be copied to. The only difference between "t" and "m" is that when we copy, the notes encompassed by the scope operator are not deleted. They are simply copied. Thus, if we want to have a copy of notes three through five mixed in to the buffer starting at note 10, then the following will do the trick:

```
3,5t10
mode: m
```

Note that the question "mode" is referring to whether the copy should be mixed or spliced into the buffer. Also, note that "0" is a legal destination with "t". Finally, dot is set to the new position of the last note copied.

EXERCISE 11:

Practice using the "t" and the "m" commands. Pay particular attention to the use of mixing and splicing. Also, see what happens when the destination falls within the range of the defined scope, as in the following example:

```
3,6t4
```

27. SCORCHESTRATION - the 'scorch' Command

This command permits the user to "orchestrate" one or more events. Unlike *setobj* (*orch*), however, *scorch* orchestrates notes with scores rather than objects. This process, which has been termed *multiplication* by Boulez, is not as strange as it might seem at first. Two examples are of use at this point.

Often, a chord will function musically as a single "gestalt". Its individual components "fuse" together into a single new timbre. In such cases it is reasonable for the composer to want to treat this group of notes as a single entity for purposes of orchestration. This can be accomplished using *scorch*. First, a new score must be created which contains just the chord (orchestrated as desired). Let us assume that the name of this chord is "fourth", and that it has three notes as follows:

NOTE	PITCH	FREQ	DUR	OBJECT	VOL	DELAY	CHAN
1:	c4	261	1/4	bssn	190	0/1	1
2:	f4	349	1/4	brass	190	0/1	1
3:	b4b	466	1/4	ww	190	1/4	1

In addition, let us assume that we have a four note melody, "root", which we want to "scorch" with the timbre defined by "fourth". For purposes of example, let "root" be defined as follows:

NOTE	PITCH	FREQ	DUR	OBJECT	VOL	DELAY	CHAN
1:	c4	261	1/4	default_obj	190	1/4	1
2:	b3	246	1/8	default_obj	100	1/8	1
3:	d4	293	1/8	default_obj	160	1/8	1
4:	c4#	277	1/2	default_obj	210	1/2	1

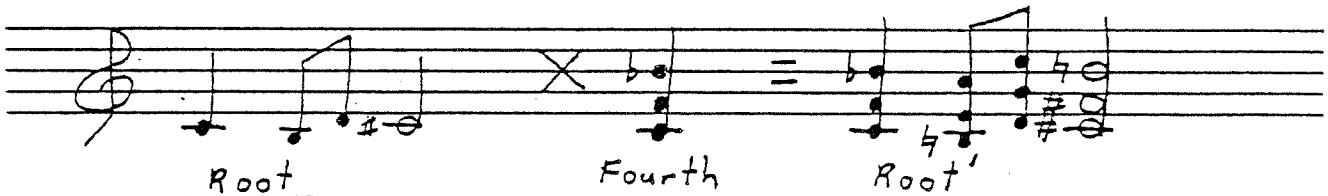
If, while editing "root", we type

```
*scorch fourth
```

we will cause each note of "root" to be orchestrated with the timbre of the fourth chord defined in the score "fourth". The result (in both CMN and sced notation) is as follows:

APPENDIX C

NOTE	PITCH	FREQ	DUR	OBJECT	VOL	DELAY	CHAN
1:	c4	261	1/4	bssn	190	0/1	1
2:	f4	349	1/4	brass	190	0/1	1
3:	b4b	466	1/4	ww	190	1/4	1
4:	b3	246	1/8	bssn	100	0/1	1
5:	e4	328	1/8	brass	100	0/1	1
6:	a4	439	1/8	ww	100	1/8	1
7:	d4	293	1/8	bssn	160	0/1	1
8:	g4	391	1/8	brass	160	0/1	1
9:	c5	523	1/8	ww	160	1/8	1
10:	c4#	277	1/2	bssn	210	0/1	1
11:	f4#	370	1/2	brass	210	0/1	1
12:	b4	494	1/2	ww	210	1/2	1



As stated, we see that "root" now consists of four fourth chords. There are several details to point out, however. First, notice that the orchestration of each chord has remained consistent with that defined in "fourth". Second, notice that only the first chord is an exact replica of "fourth", and this only by chance. The notes of each instance of the chord have maintained their relationship relative to one another. However, as a whole, each chord has been transposed in pitch, time, and loudness by an amount determined by the note being scorched. On reflection, we realize that this is consistent with the behavior of objects, whose absolute pitch, duration, and loudness are determined by the notes which they orchestrate. With *scorch*, the way this works is that the first note of each instance of the (sub) score being used for scorched is set to have the same pitch, duration and volume (not timbre) of the note being scorched. All other notes of the sub-score are then adjusted so as to maintain their relative relationship with the first. The adjusted sub-score then replaces the note being scorched.

In the previous example, notes were scorched with a score consisting of a chord. It is important to realize that the operator also works if melodic structures are used. Thus, let the following two note structure represent a score "root" which is currently in the editor's buffer:

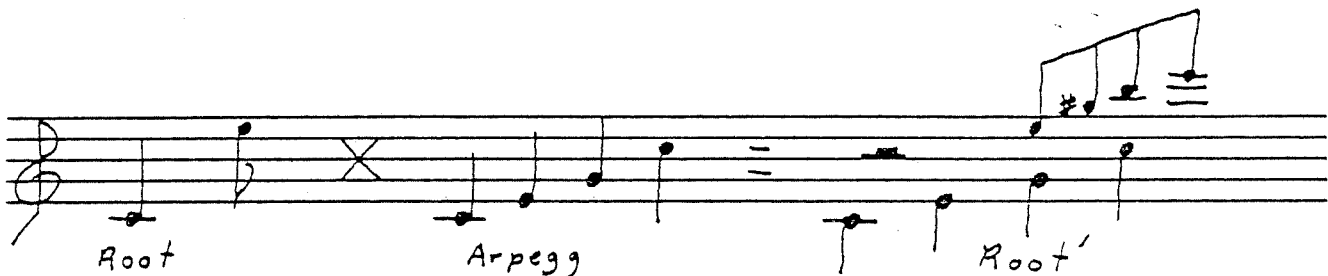
NOTE	PITCH	FREQ	DUR	OBJECT	VOL	DELAY	CHAN
1:	c4	261	1/4	brass	190	1/2	1
2:	e5	659	1/8	brass	190	1/8	1

If it is scorched with a four note arpeggio "arpegg" defined as follows:

NOTE	PITCH	FREQ	DUR	OBJECT	VOL	DELAY	CHAN
1:	c4	261	1/4	brass	190	1/4	1
2:	e4	329	1/4	brass	190	1/4	1
3:	g4	391	1/4	brass	190	1/4	1
4:	c5	523	1/4	brass	190	1/4	1

then the effect on "root" will be (notated in both CMN and sced notation):

NOTE	PITCH	FREQ	DUR	OBJECT	VOL	DELAY	CHAN
1:	c4	261	1/4	brass	190	1/4	1
2:	e4	329	1/4	brass	190	1/4	1
3:	g4	391	1/4	brass	190	0/1	1
4:	e5	659	1/8	brass	190	1/8	1
5:	g5#	830	1/8	brass	190	1/8	1
6:	c5	523	1/4	brass	190	0/1	1
7:	b5	987	1/8	brass	190	1/8	1
8:	e6	1320	1/8	brass	190	1/8	1



Notice that exactly the same process occurs, and that the instances of the sub-score are *mixed* in with the existing score. Remember also that the effect of *scorch* can be restricted to certain notes using scope, just as with any other command.

Scorchestration is a powerful and useful operator when carefully used. Among other things, it combines the effect of *mix*, *transp*, *tscale*, and *setvol* into one operator. Work with it, experimenting with simple structures so as to be able to verify that its effect corresponds with your expectations. Try using it to compose a simple canon, a descending sequence of some ornamental figure, and some "Steve Reich" type pattern piece.

28. REHEARSAL MARKINGS

It is possible to "mark" specific notes with labels which can function like rehearsal markings. The command used is 'k'. It is immediately followed by *one* lower-case alphabetic character name (such as 'a' or 'g'). The following example will label 'dot' as 'g'

kg

while the next example labels note 7 as 'x'

7kx

Note the following points. The label is associated with the specific note, not the note number. If the note is deleted, so is the label. If the number of the note is altered, the label remains with the note, regardless. Finally, there is only one note allowed per label, hence only one note number can be given as scope to 'k'.

APPENDIX C

A note label which has been thus defined, can then be used in any context where a note number would be legal. The only constraint is that the label *must be prefaced by a single quote character (')*. (How else could *scsd* distinguish between the command 'a' and a label of the same name?) An example of the use of this feature would be as follows:

```
'a,'bp
```

which could be interpreted as, "print all the notes between rehearsal markings A and B." One point to note, these markings are not saved from session to session.

29. MACROS

It often happens that the same thing is typed over and over during a work session. When this repeated text is long, this becomes tiresome. There is a feature which allows any string of typed characters to be saved, associated with a name, and every time that name appears, it is replaced by the characters that it represents. Such a feature is called a "macro".

In *scsd*, a macro is defined by specifying the macro name; followed by an equals sign (=), followed by the text to be saved. Two examples would be:

```
%fred = {obj = sax}
```

and

```
%frank = d
```

The second is silly, since the name is longer than the text it represents. Note one rule that is illustrated, however. All macro names *must* begin with the percent character (%).

Now, any place that "{obj = sax}" would be legal, we can simply type "%fred". Finally, more than one macro may appear in a single line, as seen in the following:

```
%fred %frank
```

which is the same as

```
{obj = sax}d
```

Two additional commands are provided to help with book-keeping. The first is "show", which will print all macro names and their associated text. The second is "eval" which, followed by a macro name, will print the associated text.

30. SUMMARY OF COMMANDS

GENERAL

a	append (add) new notes.	m	move notes.
c	change note.	mmn	redefine metronome.
d	delete.	p	print.
dp	delete and print.	q	quit: end edit session.
e	edit new score.	r	read score file.
f	redefine file name.	scorch	scorchstrate scores.
h	help	t	copy notes.
i	insert new notes.	vmm	verbose prompt switch.
jm	join mode Mix/Splice.	w	write (save) score.
l	listen: play score.	.=	print note number.
lm	switch listen mode.		

SPECIAL

*	all notes.	[CR]	advance dot and print.
<int>	set dot.	!!line	Shell escape with line.
&	dot through dot + 20.		

APPENDING

order:	freq, dur, obj, vol, del, chan
r<dur>	for rests
h	for help

MODIFY COMMANDS (with abbreviations)

setfreq	sf, sfreq, setpitch, spitch, sp
setdur	sdur
setobj	so, sobj, orch
setvol	sv, svol
setdel	sdel
setchan	sc, schan
settime	st, stime

REHEARSAL MARKINGS

ka	'k' is command to set marking, followed by single letter to identify the name of the marking.
'a	a defined name preceded by a single quote functions in place of note number (in scope for example).

MACROS (OR ABBREVIATIONS)

%name = <...>	defines "%name" as abbreviation for: <...>
%name	evaluate <..> defined as above.
eval <..>	extrapolate all macros in <..>
show	list contents of all macros

APPENDIX C

SCOPE (by example)

Key words: freq, dur, obj, vol, del, chan, #

Relations:

<i>RELATION</i>	<i>MEANING</i>
=	equal to
<	less than
>	greater than
>=	greater than or equal to
<=	less than or equal to
!=	not equal to
!	not (unary negation operator)

Conjunctions:

<i>CONJUNCTION</i>	<i>MEANING</i>
	or
&	and
^	exclusive or (one or the other, but not both)

h gram will cause examples to print on screen.

{freq < c4}p

{{(freq < c4) | (freq > c6 & vol = 190)}p

SEARCH

{.....} print all notes satisfying condition
defined in {.....}

/...../ print next note satisfying given condition

?.....? print previous note satisfying given condition

APPENDIX D - A Tutorial Introduction to PROD

by

Mark Green, Steve Hull and William Buxton

1. INTRODUCTION

Prod is a program for composing music scores. With the program, the user can exercise complete control over the music; however, the program also allows for various compositional decisions being made by the system using "controlled" random processes. Scores produced using *prod* are compatible with all other scores and programs in the SSSP system. That is, scores generated with *prod* may be edited with *sced*, for example, or combined and transformed like any other score.

One attraction of using *prod* is that it permits scores to be specified by their underlying structure, rather than note-by-note. This is because the input to the program is a *grammar* defining the score's structure. The output from *prod* is a score whose structure conforms to that grammar. Simply stated, a grammar is a set of rules (or *productions*) which specify the recipe to be followed in putting a score together.

The grammars used in *prod* may be *deterministic*, where the score is completely and explicitly defined by the rules of a particular grammar. Grammars may also be *non-deterministic*, where the score is not completely specified by the grammar. In this case, *prod* functions as a *composing* program. With non-deterministic grammars, *prod* makes compositional decisions according to random (or *stochastic*) processes. *prod* provides a means for the user to exercise some control over these random processes by allowing the composer to specify the alternatives available, and for each alternative, the relative probability of their being selected. Furthermore, *prod* enables the user to reproduce any score generated by random processes.

2. THE USE OF PROD IN COMPOSITION

prod is just one of the many tools available to the composer using the SSSP music system. The composer should use *prod* only for those parts of the composition for which it is best suited. *Prod* concentrates on two particular composing tasks.

The first of these tasks is the generation of parts or sections of scores in a hierarchical fashion. These sections may be used with other composing tools or combined with other parts of the score which were produced by other composing techniques. The score sections produced by *prod* may vary from one or two notes to major components of the piece.

The second composing task involves piecing together the sections of a composition to produce the final score. Some musicians develop their compositions as a number of separate sections. This could be the result of using different composing tools or techniques on each section or writing the sections at different times. Through the use of the "score" feature (see Section 9) *prod* can be used to combine these different sections into a final score. The grammar for putting together the sections can be stored on a file and called up whenever a copy of the final score is required.

3. PRODUCTIONS

The rules of a grammar are called *productions*. A production in a *prod* grammar is made up of two parts: a left side and a right side. The right side is the rule for producing a new structure, or element. The left side is the name of the new element (*non-terminal*) which results from applying this rule. The two sides of the production are separated by an "=", and the end of the production is signalled by a ";". As an example, the following rule states that the non-terminal "ex1" is composed of a single note.

APPENDIX D

```
ex1 = note ;
```

This example is complete in itself, and will generate a one note score.

4. SCORE GENERATION AND PROD USAGE

In actual practice, *prod* can be used in two ways. In the first case, usage is:

```
prod <scorename>
```

where "scorename" is the name of the score to be generated by the user-specified grammar. After invoking the command (including typing RETURN), the user types in the productions of the grammar. Once all of the productions have been specified, the user types (on a new line) CONTROL D by holding down the terminal key labelled "CTRL" and typing the letter "d". The new score will be generated and the program will be exited. Generating and playing the score defined in the first example would therefore be as follows (the per cent signs are generated by the computer):

```
%prod demo
ex1 = note;
CTRL D
%play demo
%
```

When the specified grammar has few productions, the above technique is satisfactory. However, there is no method to correct or modify productions typed on previous lines. Also, the grammar is not saved once the score is generated. To get around these problems, the best approach is to prepare your grammar as a text file *before prod* is called. This can be done using the text editor *ed* (Kernighan, 1975b). The grammar can then be fed into *prod* from this text file, rather than by direct typing. This is seen in the following example:

```
%ed gram    - start editing the file "gram"
a           - enter append mode
ex1 = note; - enter text defining grammar
.           - leave append mode
w           - save the file by writing to disk
12         - 12 characters written (computer-generated)
q           - leave the editor
%prod demo < gram
%play demo
```

Following this usage, notice how the name of the file containing the grammar appears as an argument to *prod*. Most importantly, the name of this file *must* be preceded by a left angle bracket "<". This indicates that "gram" is being fed into *prod*.

There is one other feature of the text editor which facilitates the use of prepared grammars. One need not exit the editor to call *prod*. The temporary escape feature can be used by preceding the call to *prod* with an exclamation mark "!". Thus in the previous example, *prod* could have been called and "demo" played as soon as the file "gram" was written to disk. The commands would have been as follows:

```

w
12
!prod demo < gram
!
!play demo
!
```

This example demonstrates how intermediate results can be heard using the *play* command. Before progressing any further, try each of these procedures using the simple grammar defined by "ex1". Furthermore, type in and play the grammars defined in all future examples.

5. NOTES

In the context of *prod*, a note is the lowest level element in a score structure. Notes, therefore, are referred to structurally as *terminals* (notes are not the only terminals: see Section 9). A production may have more than one note (terminal) on its right-hand side, thereby resulting in a multi-note score. This we see in the following example:

```
ex2 = note note note ;
```

The results can be heard; however they are not especially interesting. Each note has the same pitch, volume, duration, and timbre. This is because no specifications concerning these properties have been given in any of the examples seen thus far. Therefore, they were derived automatically from "default" values, which are assumed for parameters left unspecified by the user. The composer can, however, exercise control over such parameters.

In the SSSP system, there are six aspects of a note which can be specified (Buxton, 1978). They are:

- *Frequency/Pitch*: specified in either cycles-per-second or Acoustical Society of America pitch notation (e.g. c4, g3#);
- *Duration*: specified, as in *scsd*, as any value from 1/255 to 255 whole notes (e.g. 1/4 for a quarter note);
- *Timbre*: specified by the name of the *object* (instrument) which plays the note;
- *Volume*: specified as a number between 0 and 255, with 190 being about *mf*;
- *Entry Delay*: specified in same way as duration; represents the time delay before the start of the note which follows, thereby controlling rhythm;
- *Channel*: one of the four audio output channels.

These six parameters can be specified for each note by appending a list of their values, enclosed in parentheses, to the "note" symbol. Thus, we can repeat our first example, but specifying all aspects of the note this time. Note that the attributes of the note must be specified in the order given above.¹

```
ex3 = note(c2#, 1/8, chime, 190, 1/8, 0);
```

It is not necessary for the user to explicitly specify all attributes of a note. For example a quarter note played by a trumpet at a frequency of 440 hertz could be specified

1. Users familiar with *scsd* will notice that *prod* uses the same order of parameter specification.

APPENDIX D

as

```
note(440, 1/4, trumpet)
```

For parameters left unspecified, the corresponding attributes of the preceding note are carried over. Where an attribute has not previously been specified (e.g. the first note of a piece) the corresponding default value is assumed from the following list of default values: a4, 1/4, sine, 190, 1/4, 1. The following production will generate a three note sequence, with each note having the same duration (1/4), volume (150), and orchestration (trumpet).

```
ex4 = note(c4, 1/4, trumpet, 150) note(d4) note(f4);
```

Alternatively, we can give a similar example in which the timbres vary note-to-note and all else remains constant.

```
ex5 = note(c4, 1/4, trumpet, 150) note(.,flute) note(.,sax);
```

In the example, note that where parameters are left unspecified, the leading commas must be included to enable *prod* to know for what parameter a particular value is intended.

The entry delay parameter is used for differentiating between chords and melodies. This parameter specifies the time delay between the start of the current note and the start of the note immediately following. If the entry delay is zero ('0', '0/1', etc.), a chord is formed (*i.e.* there is no delay between the start of the current and following notes). If the entry delay is the same as the duration of the note, then a melody is formed. Entry delays greater than 0 but less than the note duration result in overlapping sequences, whereas entry delays longer than the duration result in rests between notes. A two note chord is produced by the following production

```
chord = note(c4, 1/6, trumpet, 100, 0/1)
       note(c4, ..., 1/6);
```

Pitches can be specified in relative, as well as absolute, terms. This is shown in the next example, which is functionally identical to the previous one.

```
chord = note(c4, 1/6, trumpet, 100, 0/1)
       note(third(c4), ..., 1/6);
```

The pitch specification of the second note is given as an interval name followed by the reference pitch (that is, the second note will have a pitch a major third above c4). The reference pitch must be enclosed in parentheses. The following table lists the intervals understood by *prod*:

NAME	INTERVAL
third	major third
fifth	perfect fifth
seventh	major seventh
octave	octave

6. A NOTE ON TYPING

At this point we can mention the following points regarding the format in which productions may be typed in to *prod*:

- no spaces are needed preceding or following the equal sign (=) which separates the left and right sides of a production.
- no spaces are needed preceding the semicolon (;) at the end of a production.
- the elements on the right side of a production must be separated by at least one space.
- productions need not be typed on a single line: wherever a "blank" may occur, several "blanks", "tabs", and/or a "return" character (ie. a carriage return) may occur.
- in any grammar, a particular name may occur once and only once on the *left* side of a production.
- anywhere a number is called for, an arithmetic expression may be used; it can include any combination of +, -, *, / and parentheses.

7. NON-TERMINALS

The type of element allowed to appear in the list on the right-hand side of a production is not restricted to note terminals. The names of non-terminals, that is, names appearing on the left-hand side of other productions, may be used as well, either alone or mixed with other non-terminals or terminals. Each time the name of a non-terminal is used in the right-hand side of a production, it serves as an abbreviation for the structure defined by the production in which it appears as the left-hand side. For example, we can use one production to specify that a composition is in A-B-A form:

composition = A B A ;

and then define two additional productions which specify the details of the "A" and the "B" structures. For purposes of simplicity, we present a trivial example of an A-B-A structure: a simple Plagal Cadence:

A = note(c3,,,0/1)
 note(e3,,,0/1)
 note(g3);

B = note(c3,,,0/1)
 note(f3,,,0/1)
 note(a3);

In the example, note that the default value assumed by the entry delay parameter is determined in a different manner than the default for other parameters. Namely, when left unspecified, the delay assumes a value equal to the duration of the current note, rather than the delay of the previous one.

The above three productions form a grammar which can be used as input to *prod*. The resulting I-IV-I cadence is shown in Figure 1.

The underlying (or "deep") structure of this particular composition is shown in Figure 2. Diagrams like this can be constructed from the productions in any grammar. This type of diagram shows how the whole composition is divided into parts, and the parts into sub-parts. Each non-terminal in a grammar is described in terms of the terminals

APPENDIX D

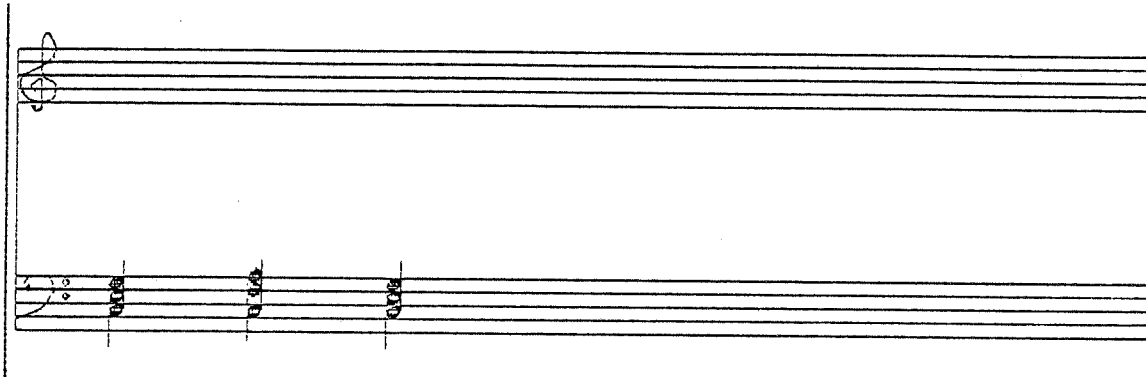


Figure 1

and non-terminals from which it is constructed. In the above grammar the non-terminal "composition" is described in terms of the non-terminals "A" and "B". Each of these lower level non-terminals is in turn described in terms of its constituent notes. Descriptions of this type which are based on different levels of detail are called *hierarchical*.

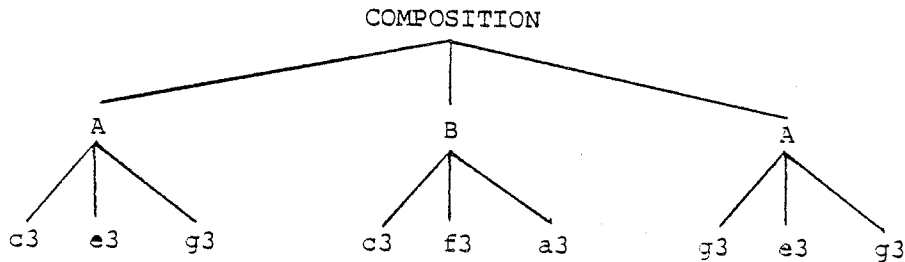


Figure 2

8. PARAMETER PASSING

In previous examples we used productions to generate chords. In these productions the bass note (root) of the chord was fixed. That is, if we wanted one major chord with d4 and another with e4 as the root, we would have to write a separate production for each. To circumvent this problem, there is a way to define productions such that particular values are not defined until the production is used. For example, we can define a production which specifies everything about a major chord *except* its root. If the left side of such a production were called "major", then the actual root of that major chord would not be defined until "major" was used in the right side of some other production. Furthermore, a different root may be specified each time "major" appears.

In such productions, the values which may change each time the production is used are called *parameters*. When the production is defined, those fields (such as pitch or object) which are to be *parametricized* in this way have their values marked by special symbols. A production for the non-terminal "major" described above would be:

```
major =      note($1, 1/4, trumpet, 150, 0)
            note(third($1),,,,0)
            note(fifth($1),,,,0)
            note(octave($1),,,,1/4);
```

Note that each place where the pitch should appear, the special symbol "\$1" is used. This notation is used to tell *prod* that the specification of the actual pitch is to be deferred until "major" is used. (Note also how the interval functions are used in order to build up the chord.)

Each time the non-terminal "major" is used in a production the pitch of the bass note must be specified. Such a parameter is specified by appending its value, enclosed within parenthesis, to the non-terminal symbol when it is used on the right side of a production. Note the similarity to specifying the parameters of notes. An example of the use of the non-terminal "major" is:

```
series = major(c4) major(d4) major(e4) ;
```

Each of the pitches used with "major" in the above example will replace the symbol "\$1" when it comes to generating a chord. This will give three different chords all produced by the same production. The result is seen in Figure 3.

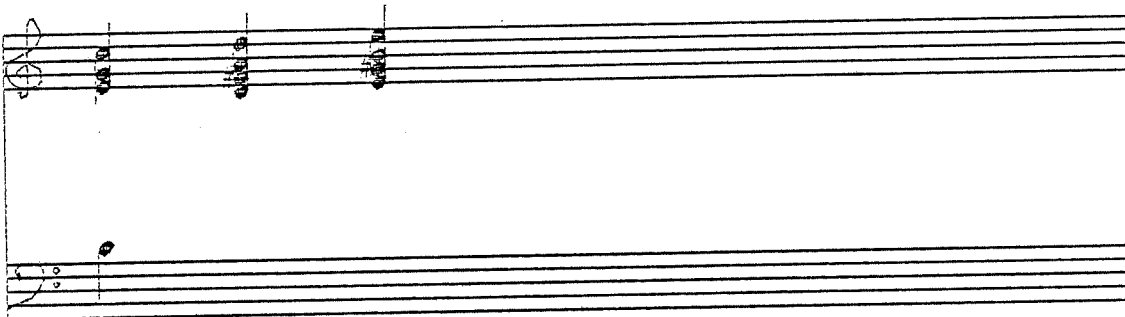


Figure 3

A more general chord production would also include parameters for object, duration, and volume of the chord. The following is such a production

APPENDIX D

```
major = note($1, $2, $3, $4, 0)
        note(third($1),,,,0)
        note(fifth($1),,,,0)
        note(octave($1));
```

The parameters to this production are (in the order in which they must be specified): pitch (\$1), duration (\$2), object (\$3), and volume (\$4). Since there is more than one parameter, there must be some means of distinguishing among them. That is the purpose of the number which follows the "\$" sign: The "\$" indicates that the value is parametrized, and the number indicates which one. Notice that the parameters which remain unchanged in all notes of the chord are explicitly specified for the root only. It is possible to leave any number of parameters unspecified. In such cases the corresponding values of the preceding note are used in their places. If one does not wish to change any parameters, the non-terminal is simply used without parenthesis.

"Major" could be used in the following way:

```
series = major(c4, 1/4, trumpet, 190)
        major(f4, 1/8, sax, 200)
        major(g4, 1/2, piano, 150);
```

Note that the parameter order specified by the user in the production *must* be adhered to when that non-terminal is used elsewhere, as it is the only way *prod* has of distinguishing the parameters.

9. THE SCORE TERMINAL

Besides "note", another terminal which can be used in *prod* grammars is called "score". This terminal is used to include pre-composed scores in the composition described by the grammar. This terminal has one parameter, the name of the score to be included. If we had two scores, "parta" and "partb", and we wanted to use them to produce a new score with the structure A-B-A the following grammar could be used:

```
composition = A B A.;
A = score(parta);
B = score(partb);
```

The score referenced by a "score" terminal can be any score in the SSSP system. It need not be one that was previously generated by *prod*. Finally, both terminal types may be used in the same production (or mixed in any combination with non-terminals), as seen in the following example:

```
examp = note(c4) score(minuet) note(f4);
```

10. NON-DETERMINISTIC GRAMMARS

Each of the grammars that we have seen describes only one score. That is, every time one of these grammars is used as input to *prod* the same score is produced as output. We call this type of generation process deterministic. One can also construct grammars which describe more than one possible score. When one of these grammars is used as input to *prod* one of the scores it describes will be generated. A different score may or may not be produced each time the grammar is used. This is referred to

as a random or non-deterministic generation process.

In order to construct a grammar which describes more than one score, we use productions with right sides consisting of several alternatives. Each of these alternatives is made up of terminals and non-terminals and is similar to the right sides of productions we have seen so far. When *prod* encounters a production which has several alternatives, one of them is chosen at random. The symbol "|" ("or" bar) is used to separate the alternatives which make up the right side. Thus, a production with two choices would be written as

$$A_OR_B = A | B ;$$

This production states that when the non-terminal "A_OR_B" is found in a production it can be replaced by either A or B.

To show how this type of production is used, consider the following example. We want to compose a 12-note melody which consists of randomly ordered pitches from a whole-tone scale. The melody can be defined by the following production:

```
mel = rand rand rand rand
      rand rand rand rand
      rand rand rand rand;
```

Each instance of the non-terminal "rand" implies a random pitch from the whole-tone scale. We now define the production "rand" such that this will actually take place:

```
rand = note(c4) | note(d4) | note(e4) | note(f4#)
       | note(g4#) | note(a4#) | note(c5);
```

In English, we can read the "|" bar as "or". Thus, each occurrence of "rand" in "mel" will give us c4 or d4 or e4, etc. One of the possible scores generated by this example is shown in Figure 4. NB: Repeating the above example with the twelve tones of an octave instead of the whole-tone scale will *not* necessarily produce a tone row, as any pitch may occur more than once.

11. WEIGHTED PROBABILITIES

In the previous example, any pitch in the scale is as likely to result as any other when "rand" is used in a production. In this case we say that each alternative has an *equal probability* of occurring. This may not be what we want. Let us use just the first three pitches of the scale in an example. Suppose we wanted the probability of the c4 occurring to be the same as the *combined* probability of d4 and e4. We can rewrite "rand" to accomplish this:

```
rand = note(c4) | note(c4) |
       note(d4) | note(e4);
```

By including c4 in the production twice (that is, two of the four alternatives are c4), we accomplish our objective.

From the example we see that one way to increase the probability of some alternative in a non-deterministic production is for that alternative to appear more than once. There is a shorter way of accomplishing this. Instead of repeating the alternative, we



Figure 4

follow it with a number which indicates that it has the same *weight* or *probability* that it would have if it were repeated that many times. The number which specifies this weight must be separated from the alternative by a tilde, ie. a "~". Thus, we can rewrite the last example as follows:

```
rand = note(c4) ~ 2 | note(d4) | note(e4);
```

12. RANGES.

We have seen how certain aspects of a score may be defined non-deterministically. *prod* can randomly combine notes, scores and combinations of notes and scores with little effort on the part of the user.

However, there are some things can only be made random in a very cumbersome manner. For example, if we wish to define a production which produces a note randomly pitched between c4 and e4, we must express it as:

```
c4_to_e4 = note(c4) | note(c4#) | note(d4)
          | note(d4#) | note(e4);
```

With a little imagination, the reader should be able to comprehend the horror facing a composer who wishes to randomly produce a single note in the range c3 to c6, having a volume from 32 to 220, one of four possible channels and a duration somewhere between that of a breve and a hemidemisemiquaver...

Fortunately for the poor composer, there is a much less painful solution. *prod* has the facility to randomly generate any numerical value in a specified range. This means that anywhere in the score that a number occurs one may, by replacing it with a construct we call a *range*, cause a random value to be used.

The syntax for a range is quite simple:

[m , n]

where "m" and "n" may be any number, ASA specification or arithmetic expression valid for the place where the range is being used.

Thus, we could solve the aforementioned composer's nightmare with the following production:

```
the_horror = note( [c3,c6],[2,1/64],[32,220],[1,4] );
```

Several things should be noted about ranges. First, any number of blanks, tabs and carriage returns may be placed in and around the range without affecting it. Second, it does not matter which of the two values specified is the larger. Finally, it is important to realize that the value is selected randomly from all possible numbers in the range. As a result, our definition of the above note using ranges differs from the "long-hand" version in that we are not limited to diatonic notes, conventional durations or even integer volume values. For example, the range [c2,c4] in no way implies that the resultant pitch will be one representable by ASA notation; in fact it is more likely not to be.²

One last point about ranges: as stated before, they can be used anywhere a value is normally used. This includes weightings, as in:

```
a = b ~ [1,5]
| c c d d c c ~ [3,2];
```

However, it also may include other ranges. Thus it is perfectly valid to use a range to set one of the bounds on another range. This perverse construction could be used for a weighting effect; for example, one could allow pitches from c2 to c4, but make the lower pitches more probable by the following construct:

```
low_pass = note( [c2, [c2,c4] ] );
```

Here the lower bound on the note's pitch is c2, but the upper bound can be anywhere from c2 to c4, thus increasing the likelihood of a lower pitch. This weighting could be increased by adding further self embedding. Since it doesn't matter which bound of the range is larger, we can bias the range towards any specific value in a similar fashion. For example, to choose a number between one and ten but preferably one close to five, we could use:

```
[ 5, [1,10] ]
or
[ [1,10], 5 ]
```

13. RECREATING RANDOM SCORES:

The heading of this section appears to be a contradiction in terms. If the score is random, how can one be sure of recreating it, and if it can be consistently recreated, why is it called random?

² However, the SSSP command *mode* can be used to cause all frequencies to be adjusted to the nearest semitone on the chromatic scale.

APPENDIX D

The solution lies in the method by which computers generate random numbers. A random number generator takes a number called a *seed*, and proceeds to produce a complex series of numbers using various mathematical tricks. This series of integers doesn't favour any numbers over any other, but it is *not* random in the sense of being unpredictable. For any given seed, the same series of numbers will always be produced.

The random number generator used in *prod* normally gets its seed from the time and date when it is called, so it is unlikely that you will ever get the same seed twice. However, it is possible for the user to specify the seed that *prod* uses, and thus cause it to pseudo-randomly generate the same score again and again. To do this, simply call the system with the usage:

```
%prod <scorename> [seed]
```

where the seed is a positive or negative integer. If a grammar file is used, it may be specified in the same manner as before.

It should be remembered that seeds are used only with *non-deterministic* scores; if a score is deterministic, random numbers are never required and the seed does not affect the score.

When a non-deterministic score is used, *prod* will finish up by typing the line

```
Seed used to generate this score: <number>
```

Thus, if your non-deterministic grammar file produces a score which you particularly like, you may reproduce it at any time by specifying this number as the seed.

Sometimes, if *prod* is called with a very large number, the seed returned at the end will not be the same number. This is because the number originally specified was cut down to a more manageable size before being used as a seed, and the number returned will give exactly the same results as the original large number.

14. RECURSIVE PRODUCTIONS

A non-terminal can be used in the right side of the production which defines it. This is known as a *recursive* production. However, if such a production has only one alternative on the right side problems will arise. Consider the following example.

```
A = A B;
```

This production says that A is to be replaced by itself followed by B. Since there are no other alternatives which can be used to replace A, this replacement process will try to continue forever. This will result in an error, and *prod* will not generate a score. On the other hand, if a production has at least one alternative which does not contain the non-terminal being defined, then an infinite sequence of replacements will not be produced. The sequence will stop whenever one of the alternatives that does not contain the non-terminal being defined is used in a replacement.

This type of production can be used for producing sequences of notes or chords which are of an arbitrary length. We can illustrate this, based on the earlier example of generating a 12 note melody made up of random pitches. We will use random pitches as before, and also let the melody length be random, but within some bounds. This we do as follows:

```
mel = rand mel ~ 11 | rand;
```

If this grammar were used, we would expect "mel" to be about 12 notes long. This is because the first alternative has probability 11/12 and each time it is selected the production will be repeated. "mel" could, however, be as short as one note long. The probability of this is only 1/12; which means it would happen on average once every twelve times the grammar is used. It could also generate a very long melody, but this would happen very rarely.

APPENDIX E

APPENDIX E - An Introduction to CONDUCT

1. THE NATURE OF A CONDUCTABLE SCORE

The main motivation for developing the system was to provide the musician with a tool which would enable pre-composed scores to be *conducted* in performance. A score is named group of notes which has been previously composed using a composing tool such as *scriva* or *sed*. The score may consist of a *single* note, or a more complex structure made up of up to a maximum of about 800 notes.

Scores can be compared with *sequences* as used in conventional analogue systems. There are two important distinctions, however: First, each note of a score may be *orchestrated* with a different timbre. Second, the structure need not be a monolinear string. That is, notes may overlap, and the number of simultaneous voices may vary between zero (tacet), and the maximum supported by the synthesizer (16). Finally, it is important to consider the notion of a *composition* as being made up of a number of *parts* (for which the division of much vocal music into "soprano," "alto," "tenor," and "bass" serves as an example). For our purposes, we consider each of these parts as a separate *score*. Thus, in order to conduct the entire composition, we must be able to conduct more than one score at a time. This we can do, with the obvious benefit being that we can now express "conductor-like" gestures such as: "a little more from the brass, and more staccato in the violins." That is, by providing a facility to independently conduct several scores simultaneously, we are provided a much-needed "handle" on the *scope* of conducting commands.

2. CONDUCTABLE PARAMETERS

For the time-being, let us consider the simpler task of conducting a single score. There are 7 parameters of the score which we can affect. Figure 1 shows these parameters in the manner in which they are labelled on the system's CRT.

SVE TEMPO ARTIC AMP RICH CYCLE ON/OFF

Figure 1. Conductable Parameters

We can now describe each of these parameters in detail.

2.1 Octave:

In composing a score, each note is notated at a specific pitch. By varying this parameter from its default value (0), one can cause the score to be performed n octaves higher or lower than originally notated.

2.2 Tempo

This parameter allows the speed of performance of a score to be altered. What is actually being scaled is the time interval separating the start of one note and the start of the next. As with conventional music, the tempo is specified as a metronome marking, indicating the number of "beats per minute."

2.3 Articulation

The previous example demonstrated how the timing *between* note attacks could be scaled. This parameter allows the user to scale the *durations* of those notes. Scaling all the durations by .5, for example, results in a staccato-like effect, while extending the durations beyond how they were notated, causes a legato-like effect. Notice the potential here for compensating for room acoustics (which may be very resonant, or

dry, for example). Notice also that tempo is unaffected by this change. Timing *between* event attacks is orthogonal to the timing of event *durations*.

2.4 Amplitude

The next parameter to be described is rather straightforward. It enables the performer to scale the dynamics, or loudness, of a score from how it was originally notated.

2.5 Richness

This parameter enables us to transform the *timbres* of the notes from how they were originally defined. The effect is similar to that of having an adjustable filter affecting the signal generated by a score. In the case of the *conduct* system, the effect of adjusting the parameter is intimately linked with the technique of sound synthesis employed. For current purposes, the synthesis technique used is *frequency modulation* (FM; Chowning, 1973). The effect of the richness parameter, therefore, is to scale the specified "index of modulation" affecting the timbre of individual notes.

2.6 Cycle

The function of this parameter is to enable the performer to specify what occurs when a playing score comes to its end. There are two options available: one, the score will stop; two, the score will repeat. This latter case we call *cycle mode*. Thus, this parameter is a binary switch specifying whether the score is in cycle mode or not.

2.7 On/Off

Finally, the seventh parameter is another binary switch which is used to control whether the score is on or off. When the value is set to "1" the score begins (is *triggered*); when it is changed to "0" the score stops, and resets.

3. TECHNIQUES OF CONTROL

3.1 General

At all times, the status of each *active* score³ is displayed on the CRT. A simplified version of the format in which these data are displayed is seen in Figure 2.

SCORE	8VE	TEMPO	ARTIC	AMP	RICH	CYCLE	ON/OFF
demo	0	60	60	0	0	1	0

Figure 2. Simplified Display of an Active Score's Attributes.

As can be seen in the figure, there is a field for the score's name, as well as one for each of the seven conductable parameters. The fields are labelled, and the current value of a particular parameter of a score is shown in the appropriate field opposite the score's name. In this case, for example, both the tempo and articulation parameters of the score "demo" are set to the default value 60.

For the purpose of control we can consider the conductable parameters as falling into two categories: *switches* and *variables*. Like a light switch, switches can be either on or off. The two switchable parameters are *cycle* and *on/off*. The others, *octave*, *tempo*, *articulation*, *amplitude* and *richness*, are all continuously variable. They are scaling factors which allow parameters to be transformed from their notated values, during

3. An active score is a score which is currently conductable. While several parts, or scores, may be conducted throughout the course of a performance, only eight scores may be conducted, or active, at any one time.

APPENDIX E

performance.

3.2 Direct Control

3.2.1 Switches

To change the state of a switch, the tracker is positioned above the switch and the cursor Z-button depressed. The switch immediately changes state, and the screen is updated ("1" and "0" representing "on" and "off", respectively). When finished playing, a score in cycle mode will repeat; otherwise, it will stop playing and the display will be automatically updated. A score may be stopped at any point during performance, at which time it will reset to its beginning and await to be restarted. (A flag to enable a score to "pick up" from where it was interrupted also exists, but has not been made available to the user in the current implementation due to problems of screen density. Using an alphanumeric terminal, we can only display 24 x 80 characters.)

3.2.2 Continuously Variable Parameters

3.2.2.1 Typing One technique for changing the value of a variable during performance is to position the tracker over the variable, and type the new value. If the performer wishes to transpose a score up an octave from where it was originally notated, he need only point at the octave field and type a "1". Alternatively, typing "-1" will lower the pitch by an octave. In either case, the change takes place immediately. The screen is updated, and if the score is playing, the result heard.

The typing interaction requires two hands: one for pointing and one for typing. To facilitate this one-handed typing, certain system-specific conventions have been adopted. First, to avoid the awkwardness of depressing the "return" key after typing a value, any numeric value can alternatively be terminated by depressing any non-numeric key. Second, in order to increase the speed of typing negative numbers, the minus sign can alternatively be indicated by depressing the "space-bar", which is equally accessible from any point on the keyboard, and whose physical appearance resembles a minus sign. These redefinitions of the keyboard have the dual advantage that they are easy to remember, and they significantly improve the bandwidth and reliability which can be achieved through one-handed typing.

3.2.2.2 The 'Last-Typed' Technique While we have attempted to make typing as efficient as possible, in many cases it is not the most appropriate means of communication. Often during performance there is simply no time to type. One alternative exploits the observation that we often assign the same value to more than one field. The system takes advantage of this redundancy by designating cursor button-3 as the "last-typed" button. Placing the tracker over a variable and depressing button-3 causes the last value typed to be assigned to that variable. Again, the display is updated and the effect may be heard immediately.

3.2.2.3 Default Set Another often typed value was observed to be the "default", or "normal" value for each variable field ("0" for all parameters except *tempo* and *articulation*, which have a default of 60). These are the values that cause the score to be performed "as notated". To facilitate the frequent desire to restore a parameter to its default, cursor button-2, has been designated the "default" button. Using the technique seen in "last-typed" mode, any variable can be reset to its default by placing the tracker over that parameter, and depressing button-2.

3.2.2.4 Dragging Perhaps the most effective technique for directly modifying the value of a variable is the technique that we call "dragging". This is a direct approach analogous to reaching out and turning a knob on a console. With dragging, the tracker is placed over the variable to be updated. By moving the cursor in the vertical (y) domain, *while holding down the Z-button*, the value is, in effect, "dragged" up or down.⁴ During this process, the screen is continually updated with the current value,

and the results can be heard simultaneously. There is then, an immediately accessible "virtual" potentiometer available for each continuously variable parameter *without any special purpose hardware*. Pots can be added, moved, or scaled using this technique without any physical change to the system. The technique is direct, fluent, intuitive, inexpensive and only utilizes one hand. Finally, it is clearly adaptable to many other control applications, not the least of which is digital sound recording.

3.3 Indirect Control

3.3.1 Triggers

3.3.1.1 Manual Triggers One shortcoming of the control techniques described above is that they only allow one parameter to be changed at a time. The deficiency of this can be seen in contexts such as unison starts: starting more than one score with a single gesture. In the case of the *on/off* parameter, the way around this problem is to allow several scores to be started by firing a single "trigger". The use of such a trigger can be considered in two phases. The first is the "set-up" phase: grouping together the scores to be fired by a particular trigger. The second is the actual trigger firing.

There are many ways that scores may be triggered. Two of them, triggers nine and ten (T9 and T10) can be fired manually with the cursor Z-button. Opposite the *on/off* parameter for each score is a control field to which a trigger number can be assigned. This field is initially set to "-", indicating the default "no trigger assigned" condition. This can be seen in Figure 3.a. A score can be linked to a particular trigger by pointing at the control field and typing the trigger number. Therefore, as is illustrated in Figure 3.b, score "test1" can be linked with T9 by pointing at its control field and typing "9". The second score "test2" can then be grouped to the same trigger simply by pointing at its control field and pressing cursor button-3 (using the "last-typed" technique described above).

Specifying the trigger number constitutes the set-up phase. In order for the scores to be started, the trigger must be fired. In the case of triggers nine and ten, this is performed by placing the tracker over the appropriate light-button (T9 or T10 shown in Figure 3.c), and depressing the cursor Z-button. All *on/off* switches controlled by that trigger will then change state. As is indicated in Figure 3.d, this means that if one score is on and the other off, and both are controlled by the same trigger, firing that trigger will cause the one to switch off and the other to switch on. Anywhere from zero to eight scores can be controlled by any trigger, but only one trigger at a time can control a particular score. Trigger assignment may be changed at any time during performance, and the trigger control of a particular score can be cleared by pointing at the control field and depressing the cursor Z-button (or button-2, the "default" button).

There is another manual trigger, T11, that does not appear on the screen. The reason is that it is fired by hitting the touch tablet rather than with the tablet cursor. Thus, by assigning the number '11' in the trigger field of one or more scores enables them to be triggered as from a drum.

The clavier can also be used to trigger scores. Each pitch class (white notes only) is capable of generating a different trigger. Thus, a different score can be connected to each key and triggered on that key's depression. Chords and motives of scores can thereby be performed. There are 7 distinct clavier triggers. They are numbered -1 to -7 (always preceded by '-'), which are associated with the pitch classes c through b (any octave).

4. In order to prevent values at the top of the screen from being discriminated against (in terms of "dragging-room"), the mapping of the tablet co-ordinates to screen co-ordinates leaves a "margin" area at the top of the tablet co-ordinate space.

APPENDIX E

SCORE	8VE	TEMPO	ARTIC	AMP	RICH	CYCLE	ON/OFF
test1	0	60	60	0	0	1	0 -
test2	0	60	60	0	0	1	0 -

a) Parameters including the control fields ("-") for remote triggering.

SCORE	8VE	TEMPO	ARTIC	AMP	RICH	CYCLE	ON/OFF
test1	0	60	60	0	0	1	0 9
test2	0	60	60	0	0	1	0 9

b) The same two scores with trigger 9 linked to each.

SCORE	8VE	TEMPO	ARTIC	AMP	RICH	CYCLE	ON/OFF
test1	0	60	60	0	0	1	1 9
test2	0	60	60	0	0	1	1 9

T9
T10

c) The same two scores playing after a unison start triggered by "firing" T9, shown for the first time. If the value 10 was in the control field, rather than 9, the scores would be fired by firing T10.

SCORE	8VE	TEMPO	ARTIC	AMP	RICH	CYCLE	ON/OFF
test1	0	60	60	0	0	1	1 9
test2	0	60	60	0	0	1	0 9

T9
T10

d) The "flip-flop" nature of triggers. Firing T9 will cause "test1" to stop, and "test2" to start.

Figure 3. Examples of Trigger Usage

3.3.1.2 End of Score Triggers An important concept which we wanted to incorporate into the system was to allow trigger events to be generated by events in the music itself. While this feature has not been implemented in a general way, one type of event in the data can generate a trigger. Any time a playing score comes to its end (regardless of whether it is in cycle mode or not) it generates a signal which can be used as a trigger. These trigger events are numbered "1" to "8", corresponding to the eight scores which may be active at one time. Score "a" can trigger score "b", and *vice versa*. Complex combinations of score material can thereby be built up, either in sequence, or in parallel. The only constraint is that a trigger is only generated at the "normal" end of a score, not when the score is turned off mid-way. (Note that a score can trigger itself, although that would be redundant, given the cycle switch -- which is implemented using the trigger mechanism.)

3.3.2 Groupings of Continuously Variable-Parameters

3.3.2.1 Groups Like the *on/off* switches, continuously variable parameters can be grouped together and indirectly controlled as a single unit. The approach taken is conceptually similar to the use of a "sub-master" control in a conventional audio mixer. As with *on/off* switches, associated with each variable is a control-field which is initialized to "-", or "null".

SCORE	8VE	TEMPO	ARTIC	AMP	RICH	CYCLE	ON/OFF
test1	0 -	60 -	60 -	0 -	0 -	1	0 -
test2	0 -	60 -	60 -	0 -	0 -	1	0 -

GROUPS	RAMPS	TRIGGERS
G1 -	a 0 0	T9
G2 -	b 0 0	T10
G3 -	c 0 0	
G4 -	d 0 0	
G5 -		
G6 -		
G7 -		
G8 -		

a) A simplified view of the screen layout showing the control fields (marked by the '-' character) for both score parameters and groups.

SCORE	8VE	TEMPO	ARTIC	AMP	RICH	CYCLE	ON/OFF
test1	0 1	60 2	60 2	0 3	0 -	1	0 -
test2	0 -	60 2	60 2	0 3	0 -	1	0 -

GROUPS	RAMPS	TRIGGERS
G1 sldr1	a 5 1	T9
G2 sldr2	b 5 -2	T10
G3 x	c 0 0	
G4 -	d 0 0	
G5 -		
G6 -		
G7 -		
G8 -		

b) The use of groups is illustrated. Group 2 is controlled by slider two. The articulation and tempo of both scores are members of this group. The octave of "test1" is the only member of group one which is controlled by slider one. The amplitudes of the two scores form group three, which is controlled by the x-mouse.

Figure 4. The Use of Groups

This is diagramed in Figure 4.a. Any variable can be controlled by any one of eight

APPENDIX E

group controls, numbered "1" to "8". Group set-up takes the form of pointing at the control field and indicating the group number (either by typing, or by using the "last-typed" technique). This is illustrated in Figure 4.b where the *tempo* and *articulation* of both scores has been assigned to group two, the *octave* of "test1" to group one, and the *amplitude* of both scores to group three.

One task remains in order to complete the set-up phase: a transducer must be assigned to control each group to be used. This additional level of indirection is important in that it allows any group to be controlled by any single transducer, and a single transducer to control more than one group.

3.3.2.2 Group Control Transducers There are currently ten transducers available as group controllers. They fall into five generic types: sliders (2), the cursor as "mouse" (x and y), the touch tablet, the clavier, and software ramps (4). All transducers have one important quality in common: they are all *motion*, rather than *position*, sensitive devices. That is, they increment or "delta modulate" the parameters which they control. The instantaneous value (or type) of the individual parameters being controlled is irrelevant to the transducer's function. Values are simply incremented or decremented when the transducer moves.⁵ Thus, any transducer can control many parameters, all having different instantaneous values, without any concern for context. The "nulling problem" which plagues most automated mix-down systems is thereby avoided. The resulting ability to rapidly switch the context of a transducer results in a maximum of control bandwidth from a minimum number of physical transducers.

TRANSDUCER	TYPED VALUE
slider 1	1
slider 2	2
x-mouse	x
y-mouse	y
x-touch	h
y-touch	v
clavier	k
ramp a	a
ramp b	b
ramp c	c
ramp d	d

Figure 5. Summary of Group Control Codes

A transducer can be assigned to a group by pointing at the group control field and specifying the transducer identifier. Typing '1' while pointing at the control field of group one (the '-' opposite the label G1 in Figure 4.a), assigns slider one to control that group (as seen in Figure 4.b). Moving the slider upwards will increment all members of the group (the *octave* variable of "test1" in the figure), while moving it downwards will decrement all values. Similarly, we can use the tablet cursor as a group controller. In this case, relative motion in the horizontal (x) and vertical (y) domain can each be used to control a group. This is illustrated in Figure 4.b where we have specified that motion in the horizontal domain should control group three (by typing "x" in the control field opposite G3).⁶ As a result, all horizontal cursor motion which takes place

5. A special note should be made about the clavier in this regard. It is used in an unusual way. The centre f# has no effect when the clavier is used to control a group. Hitting any other key will cause a change in the parameter values controlled by the group. The magnitude of the change will be proportional to the distance from the central f# of the depressed key. The direction of the change depends on if the key depressed is to the right (decrease) or left (increase) of the f#.

while cursor button-1 is depressed will affect the amplitude of both scores. Alternatively, we could have typed an "a" opposite G3, thereby specifying that group three is to be controlled by ramp: a. A ramp is a software transducer which provides the benefits of an automatic fader, whose direction and speed can be easily controlled. Each of the four ramps, as illustrated in Figure 4.b, is associated with two parameters. The first indicates how often (in 50ths/sec.) the controlled group's members are to be incremented. The second field indicates the size of that increment. In the example given, ramp "a" will provide an increment of one every five time units, whereas ramp "b" would provide a decrement of two. Thus, a simple mechanism is provided which enables the parameter to be dynamically varied in a controlled manner, while leaving the hands free for other purposes.⁷

3.3.2.3 Negative Groups The final new concept to be introduced concerning groups is the notion of a "negative" group. When specifying that a variable, such as articulation, is to be controlled by a particular group, one has the option of prefixing the group number with a minus sign. The effect of this is that when the members of group "n" are incremented, members of group "-n" are decremented by the same value. The control structure thereby provides a built-in facility which allows cross-fades to be controlled by a single transducer. Duration can vary inversely with tempo, richness with amplitude, and the whole process is independent of which transducer is controlling the group.

3.4 Additional Performance Variables

3.4.1 Score Selection

We have already pointed out that the performer may conduct up to eight scores at a time. These are what we have called the eight "active" scores. In the course of a composition, a performer may wish to utilize more than eight score files. A mechanism has been provided, therefore, whereby active scores can be replaced by those from a "reserve" list.

The reserve list is made up of the set of all scores specified by the performer during the set-up phase of the *conduct* program. They are added to the list as their names are typed, and they are read into primary memory. Once initialization is completed, the first eight of these scores will automatically appear on the display as active scores. In addition, in the bottom right-hand corner of the display, there appears a list containing three names. This is illustrated in Figure 6 (which is the first complete facimile of the display shown thus far). This list is a "window" showing the names of the first three scores on the reserve list. Using two special keys on the keyboard ("↑" and "↓"), we can cause the names in the list to (circularly) scroll up and down, thereby enabling us to display the name of any score on the reserve list.

To have a new score appear in the upper half of the screen where it can be conducted, one points to the name of some score which is already there, but which can be replaced. If the old score is not playing, depressing the cursor Z-button will cause the score whose name appears at the top of the reserve list window, to replace it. Using Figure 6 as an example, pointing at the name "jig" and depressing the Z-button will cause it to be replaced by "bass". At the same time, all variables associated with that score are set to their default values. Therefore, to access any score on the reserve list, one need only scroll through the list until that score's name appears at the top of the window.

The active list may be thus updated without disturbing any other scores which may be playing. An important point is that there may be more than instance of a particular

6. Note that in typing alphabetic data, any non-alphabetic character function as an alternative to the "return" character.

7. A summary of the special characters used to specify each transducer for the purpose of group control is shown in Figure 5.

APPENDIX E

SCORE	EVE	TEMPO	ARTIC	AMP	RICH	CYCLE	ON/OFF
test1	0 1	60 2	60 2	0 3	0 -	1	0 -
test2	0 -	60 2	60 2	0 3	0 -	1	0 -
jig	0 -	60 2	60 2	0 3	0 -	1	0 -
mel	0 -	60 2	60 2	0 3	0 -	1	0 -
treb	1 -	60 -	30 -	0 4	0 -	1	1 -
treb	0 -	60 -	60 -	0 4	0 -	1	1 -
treb	-1 -	60 -	120 -	0 4	0 -	1	1 -
rotten	0 -	60 -	60 -	0 -	0 -	1	0 -

GROUPS	TRIGGERS	RAMPS	RATE
G1	sldr1	T9	a 5 1
G2	sldr2	T10	b 0 0
G3	x		c 0 0
G4	a		d 0 0
G5	-		
G6	-		bass
G7	-		joe
G8	-		mel

Figure 6. The Complete Screen as Seen by User

score on the active list at any given time. Each instance of the score may have a completely different set of transformations affecting it, and all may be playing simultaneously. This is illustrated in Figure 6 by the three instances of the score "treb". Significantly, regardless of the number of instances of a particular score, there is only one copy of that score in primary memory!³ This is an important feature given the system constraint that all score material must be in primary memory before the start of a performance, and that there is only about 16K words of data memory once the program is loaded.

3.4.2 The Rate Control

The *RATE* parameter seen in the bottom right-hand side of Figure 6 is a frequency control for the master clock of the system. Lowering its value (to a minimum of "0"), by typing or dragging, speeds everything up. Conversely, raising its value slows things down. It is a rather coarse control which determines the rate at which the synthesizer is updated, with the minimum value resulting in a rate of 50 Hz. The main benefit of this control is to overcome the limitations of the computational bandwidth of the processor. It enables the user to set the update rate such that the system is able to finish computing the current update data before an interrupt comes requesting that for the next set. It can, of course, also be used to effect global accelerandos and retards; however, these are better realized through the use of groups.

3. The use of instances is further explained in Buxton, Reeves, Baecker, and Mezzi (1979).

3.5 Concluding Comments on the Control Structure

The point to emphasize in considering the control structure is that it supports parallel control functions.

BUTTON	VARIABLES	CONTEXT	
		SWITCHES	CONTROLS
Z	drag	change state	clear
1	<---- x/y	"mouse" mode	---->
2	last-typed	N/A	last-typed
3	set default	N/A	clear

Figure 7. Summary of Cursor Button Functions

For example, the members of a group can be incremented by moving slider one, while another value is being dragged up using the cursor.⁹ Given the serial nature of most digital computers, and given most current programming languages, this parallel control is one of the most difficult constructs to deal with in an elegant manner. This is one area of research to which we are currently devoting much of our attention. In the meantime, we find it rather ironic that those of us who jumped on the all-digital bandwagon are now spending so much of our energy trying to emulate the parallelism inherent in the analogue systems which we were so quick to abandon.

9. A summary of the special functions associated with the cursor buttons is shown in Figure 7.

APPENDIX F

APPENDIX F - User Defined Shell Commands

A combination of alphanumeric command primitives and the editor (*ed*) allow compositions to be defined in terms of their underlying structure, or "recipe", rather than by note-by-note specification. The benefit of this approach arises in cases where the composer conceives of a composition in terms of a set of basic "germinal" ideas, or scores, which then undergo a set of transformations, the net result of which is the final composition. In this case, the germinal score material is composed using the normal tools available with the SSSP system. The composition is then specified in terms of these germinal scores, and the transformations possible through the command repertoire of the music system. We now work through a set of examples which illustrate this approach. To use the facility described, however, it is essential that the composer have an understanding of the text editor "ed". Familiarity with the score editor "sed" will make this an easy task.

Rather than typing commands and having them executed directly, execution can be deferred by typing exactly the same thing as text into a file, using "ed". If there existed a file called "fred" which contained the following two lines:

```
retro minuet : temp
transp temp a4
```

then the two lines of the file "fred" could be caused to be executed simply by typing the following:

```
sh fred
```

The result would be exactly the same as typing the two lines which "fred" contains: that is, a score called "temp" would be created which would be a retrograde version of minuet, transposed to start on a4.

With the preceding example, we have not gained very much. It would have been just as fast to type the two lines. The real benefit occurs in cases where "fred" would contain a longer sequence of often repeated commands. Let us pursue this for a moment. In the tape studio, to hear your ideas, a long period of assemblage must take place. If one wants to change one of the constituents in this assembly, one which has several instances, a great deal of work must be faced, not the least of which is remembering exactly how the piece was put together. (Not at all an uncommon problem.) However, using the editor as outlined above, we have an explicit record, or "recipe", of the piece's structure. At any time, one of the constituent "germinal" structures in the work can be modified, and the new version of the score assembled, following the stored recipe. The composer need only type one command: sh. Furthermore, at any time, the composer can update, correct, or extend his recipe.

Up until now, we have only dealt with cases where we wanted an exact repetition of the instructions in the recipe file (which is properly called a *shell file*, (from whence the name of the "sh" command derives). In many cases, however, a composer will want to carry out the same set of transformations on different source material. That is, the deep (or syntactic) structure may be the same, but the semantics different. By introducing a strongly coupled pair of concepts, that of the *argument* and of the *variable*, we see that the use of shell files can be addressed to this point, as well. This is accomplished by enabling the user to use shell files to create their own operators, operators which are made up of sequences of existing primitives, or commands, but operators which will work on any valid file. The following example demonstrates how this is accomplished. Consider that the file "fred" now contains the following:

```
retro $1 : temp
transp temp a4
```

If one now typed:

```
sh fred minuet
```

The result will be exactly the same as in the first example above. The reason is that, on execution, the "\$1" inside the file "fred" is replaced by the the first (and only) argument (minuet) following "fred" when the sh command was invoked. The argument \$1, by convention, is a variable representing the first argument following the shell file name when the shell file is executed.

Let us expand on the previous example with another. Let us say that "fred" now contains:

```
retro $1 : $2
transp $2 a4
```

Then, executing the shell file as follows:

```
sh fred minuet temp:
```

will have exactly the same effect as all the other examples. This is by virtue of the fact that on execution each instance of "\$2" in "fred" is replaced by the second (corresponding to the 2) argument following the shell file name when the shell file is executed: that is, \$2 is replaced by temp.

We now can see a pattern emerging, where each instance of variable "\$n" in the shell file is replaced on execution by the nth argument following the shell file name in the command line which invokes sh. Think, then, how we could use a third variable to allow this "fred" transformation to be generalized to constitute retrograde followed by transposition to some arbitrarily specified pitch. Your solution should be as follows:

```
retro $1 : $2
transp $2 $3
```

which is properly invoked by:

```
sh fred minuet temp: c4
```

One point to note, unlike the basic music commands, shell commands of your own manufacture obviously require that the arguments appear in a specific order. It is up to you to remember them.

Don't forget that you can go and change the recipe at any time, simply by using ed. Also, your commands can be made to look even more like the regular music commands. By using the command "mx" as follows:

```
mx fred
```

"fred" (or any other shell file appearing as an argument) is made executable (mx = Make Executable) on its own, without having to use the "sh" command. Thus, after using mx, the following line will invoke our recipe fred:

```
fred minuet temp g5#
```

Finally, once a shell file command has been defined using the techniques described above, the resulting command can then be used as an operator within yet another shell file, and that within another ... This can be seen if we consider a file "joe" as

APPENDIX F

containing:

```
fred minuet temp g5#  
fred trio junk c4  
splice temp junk : new  
rm temp junk
```

Executing the command:

```
sh joe
```

will result in a score called "new" being created, which has two main sections: the score "minuet" transformed by "fred", with the "fred" transformed version of the score "trio" spliced on to it. In the example, note that the "temp" and "junk" are only intermediate versions of the score, which need not be saved. They have, therefore, been removed once "new" has been created, using the command *rm*.

The facility offered by the use of shell files is far more extensive than described above. Features such as loops and conditional execution are also available. They are, however, beyond the scope of this document. Users wishing to obtain more details should consult the UNIX Programmers Manual.

Synthesizer Specifications

APPENDIX G - Synthesizer Specifications

The synthesizer used by the SSSP system was developed at the University of Toronto. A detailed description of the device can be found in Buxton, Fogels, Fedorkow, Sasaki, and Smith (1978). A summary of its specifications are as follows:

- number of oscillators: 16
- number of simultaneous voices: 1 - 16
 - depends on object type
 - FM and Waveshaping: 2 osc/voice
 - VOSIM and Fixed Waveform: 1 osc/voice
 - Additive synth.: 1 osc/partial
- number of output channels: 4
- sampling rate (per oscillator): 50 kHz
- frequency resolution: 1 Hz. (integer)
- frequency range: -25 kHz - +25 kHz
- dynamic range: over 100 dB
- signal to noise ratio: better than 68 dB

APPENDIX H

APPENDIX H - SSSP PUBLICATIONS . MONOGRAPHS

- Baecker, R. (1979). Towards an Effective Characterization of Graphical Interaction. Presented at IFIP W.G. 5.2 Workshop on Methodology of Interaction, Selliac, France.
- Baecker, R., Buxton, W. . Reeves, W. (1979). Towards Facilitating Graphical Interaction: Some Examples from Computer-Aided Musical Composition. *Proceedings of the 6th. Man-Communications Conference*, Ottawa, May 1979: 197-207.
- Buxton, W. (1977). A Composer's Introduction to Computer Music. *Interface G*: 57-72. French translation: Les ordinateurs et le compositeur: une introduction generale. *Revue FAIRE* 4/5: 15-20. Italian translation in: Polo, N. (Ed.) *Musica e Elaboratore*, Venice, Laboratorio permanente per l'Informatica Musicale della Biennale di Venezia.
- ed. (1977). *Computer Music 1976/77: a directory to current work*. Ottawa: The Canadian Commission for Unesco.
- (1977). Towards a Computer Based System for Music Composition and Performance. Invited paper presented at the ACM/SIGLASH meeting, New York University, October 1977.
- (1978). Design Issues in the Foundation of a Computer-Based Tool for Music Composition. *Technical Report CSRG-97*. Toronto: University of Toronto.
- (1981). *Music Software User's Manual*. (Second Edition) Toronto: SSSP/CSRG, University of Toronto.
- Buxton, W. . Fedorkow, G. (1978). The Structured Sound Synthesis Project (SSSP): an Introduction. *Technical Report CSRG-92*, Toronto: University of Toronto.
- Buxton, W., Fedorkow, G., Baecker, R., Reeves, W., Smith, K.C., Ciamaga, G., . Mezei, L. (1978). An Overview of the Structured Sound Synthesis Project. *Proceedings of the 1978 International Computer Music Conference*, Dept. of Music, Northwestern University, Evanston Illinois. Vol. 2: 471-485.
- Buxton, W., Fogels, A., Fedorkow, G., Sasaki, L., . Smith, K. C. (1978). An Introduction to the SSSP Digital Synthesizer. *Computer Music Journal* 2.4: 28-38.
- Buxton, W., Patel, S., Reeves, W., . Baecker, R. (1980). On the Specification of Scope in Interactive Score Editors. Presented at the Fourth International Conference on Computer Music, Queen's College New York, November 1980.
- (1980). "Objects" and the Design of Timbral Resources. Presented at the Fourth International Conference on Computer Music, Queen's College New York, November 1980.
- Buxton, W., Reeves, W., Baecker, R., . Mezei, L. (1978). The Use of Hierarchy and Instance in a Data Structure for Computer Music. *Computer Music*

Journal 2.4: 10-20.

- Buxton, W., Reeves, W., Fedorkow, G., Smith, K. C., . Baecker, R. (1979). A Computer-Based System for the Performance of Electroacoustic Music. *AES Preprint 1529 (J-1)*.
- (1980). A Microcomputer-Based Conducting System. *Computer Music Journal 4.1: 8-21*.
- Buxton, W., Reeves, W., Patel, S., . O'Dell, T. (1979). *SSSP Programmer's Manual*. Toronto: Unpublished manuscript, SSSP/CSRG, University of Toronto.
- Buxton, W. . Smith, K. C. (1979). Brief on SSSP Hardware: Current and Proposed. Toronto: unpublished manuscript, C.S.R.G., University of Toronto.
- Buxton, W. . Sniderman, R. (1980). Iteration in the Design of the Human-Computer Interface. *Proceedings of the 13th Annual Meeting, Human Factors Association of Canada*.
- Buxton, W., Sniderman, R., Reeves, W., Patel, S. . Baecker, R. (1979). The Evolution of the SSSP Score Editing Tools. *Computer Music Journal 3.4: 14-25*. 3.4 (December 1979).
- Fedorkow, G. (1978). *Audio Network Control*. Toronto: M. Sc. Thesis, Dept. of Electrical Engineering, University of Toronto.
- Fedorkow, G., Buxton, W., Patel, S., . Smith, K. C. (1980). An Inexpensive Clavier. Presented at the Fourth International Conference on Computer Music, Queen's College New York, November 1980.
- Fedorkow, G., Buxton, W. . Smith, K. C. (1978). A Computer Controlled Sound Distribution System for the Performance of Electroacoustic Music. *Computer Music Journal 2.3: 33-42*.
- Green, M. (1980). PROD: A Grammar Based Computer Composition Program Presented at the Fourth International Conference on Computer Music, Queen's College New York, November 1980.
- Hogg, J. . Sniderman, R. (1979). *Score Input Tools Project Report*. Toronto: Unpublished manuscript, SSSP/CSRG, University of Toronto.
- Laske, O. E. (1978). Considering Human Memory in Designing User Interfaces for Computer Music. *Computer Music Journal 2.4: 39-45*.
- Patel, S. (1979). *Score Editor Design*. Toronto: unpublished manuscript, SSSP/CSRG, University of Toronto.
- Reeves, W., Buxton, W., Pike, R., . Baecker, R. (1978). Ludwig: an Example of Interactive Computer Graphics in a Score Editor. *Proceedings of the 1978 International Computer Music Conference*, Dept. of Music, Northwestern University, Evanston Illinois. Vol. 2: 392-409.
- Roads, C. (1979). Advanced Directions for Computer Music. Toronto: unpublished manuscript, Computer Systems Research Group, University of Toronto.
- Sasaki, L. (1977). *Macro Music I User's Guide and Language Reference Manual*.

APPENDIX H

Toronto: unpublished manuscript, Dept. of Electrical Engineering, University of Toronto.

----- (1978). A Description Language Approach to Compositional System Design. Toronto: unpublished manuscript, Dept. of Electrical Engineering, University of Toronto.

Sasaki, L. H. . Smith, K. C. (1979). *Digital to Analogue Converter Systems for Audio*. Toronto: unpublished manuscript, Dept. of Electrical Engineering, University of Toronto.

----- (1978). *Music Synthesis*. Toronto: unpublished manuscript, Dept. of Electrical Engineering, University of Toronto.

----- (1980). A Simple Data Reduction Scheme for Additive Synthesis. *Computer Music Journal* 4.1: 22-24.