MUSICIAN - A MUSIC PROCESSING
AND SYNTHESIS SYSTEM

by

A. Ben and J.A. Makowsky

Technical Report #435

September 1986
MUSICIAN - A MUSIC PROCESSING AND SYNTHESIS SYSTEM

A. Ban and J.A. Makowsky
Laboratory for Computer Music Engineering
Technion- Israel Institute of Technology.
Haifa, Israel

Abstract: MUSICIAN is a system for music processing and synthesis intended to work with a digital or analog synthesizer. It is best used for experimentation with sounds and sound effects. The MUSICIAN system is a basis for building interactively high level effects from low level music and signal processing operations. In this sense it is a first step towards an object-oriented programming environment for integrated synthesizer/sequence systems.

The user of MUSICIAN can define for himself musical data types, transformations and instrument architectures which fit his aesthetic predilection in his musical work. It stresses the construction of new sounds from previously built sound structures rather than synthesizing each sound structure from scratch. It supports modularity and gives feedback in alphanumeric, graphic, or acoustic form. MUSICIAN, once miniaturized and productized, can serve as a user-friendly device for restructuring sound effects.

The present prototype of MUSICIAN is written in FORTRAN 77 and runs on a PDP11/RSX11M or VAX11/VMS environment with an attached MiniMap Array processor. The array processor is included to give the system almost real-time capabilities.

1 Current address: ELSCINT Ltd, MRI Division, 60 Medinat Hayehudim, Herzlia, Israel

2 The Laboratory of Computer Music Engineering is a joint laboratory of the faculties of Electrical Engineering and Computer Science at the Technion- Israel Institute of Technology, Haifa, Israel. It is headed by Dr. U. Shimony (EE) and Prof. J.A. Makowsky (CS). The laboratory is partially funded by grant no. 121-625/972 of the Volkswagen Foundation for the project "Notational Systems for Electronic Music", which is carried out with the collaboration of the composer Prof. J. Tal (Jerusalem).
1. Introduction.

Suppose you have built a synthesizer. It is an advanced piece of work, built with state-of-the-art technology. It has 256 channels and can synthesize 1 million samples per second. It has a complex and sophisticated communication protocol, involving matters as special as envelope generation and time scheduling. And it works. However, if you have not thought about it earlier, it will soon become apparent to you that, unless you have a software system at least as beautiful as your synthesizer to control it, you will have much trouble taking advantage of your machine’s design. Or assume you have a ready-made synthesis system, complete with its musical keyboard interface and editing functions, and you are experimenting with sounds. You may often find that you are trying to do something, which, although it seems to be within the power of the system, is nevertheless impossible to implement because of the rigidity of your user interface. Then again, you may be given the task of designing a musical system, of selecting elements and functions, and combining them in a powerful and useful way. Most often you will rely heavily on experience and published work, design a prototype, and then make major or minor revisions according to feedback. But if your system is not well-suited for doing experimentation, such a course may be difficult and messy.

The above problems may seem unrelated, but are typical of problems facing computer music systems users and designers. At bottom the questions involved are the very basic issues of flexibility, programmability, user interface, the need for R&D tools and the high effort in software development and maintenance.

These questions are not new. Neither are they especially unique to computer music. They prevail in every aspect of software design. Usually the solution to these questions depends very much on the particular application the software system was written for and general recipes like Object Oriented Programming [Lie82, RC84] or Logic Programming [BM85] do not solve these problems yet. The developers of computer music systems have been preoccupied in the past two decades by technical problems, but this activity is now reaching a stage where the more abstract and conceptual problems of software development become central and are slowing progress [Sch83, Le84, LA85].

It is with the intention of finding a way out of such problems, or rather, of avoiding having to run into them, that we have developed a new music processing and synthesis system at the Laboratory of Computer Music Engineering at the TECHNION in Haifa. We call our system MUSICIAN [Ban85].

2. Introducing MUSICIAN

As a system MUSICIAN is somewhat difficult to characterize, since it plays a double role of an independent system, and of a music-system generator. As a generator of music systems MUSICIAN offers an interactive programming language for defining functions and procedures, as well as communication protocols which enable MUSICIAN to be integrated in other systems. It is open-ended in both ends of system hierarchy, i.e. it may be used as the computational tool doing the actual synthesis, with the user-interface format prescribed from above, or alternatively, as a high-level driver for external synthesis hardware.
On the other hand, MUSICIAN serves also as an independent musical system, with its own defined user-interface and connection to an audio-DAC. In this role MUSICIAN serves as a flexible tool for development and experimentation. MUSICIAN’s alter-ego of a system generator then makes the transition from a development system to a production system easy and smooth. The spectrum of MUSICIAN users may include programmers, music theorists and musicians (the people, not the system...), which all use basically the same vocabulary in their dialogue with the system.

MUSICIAN’s command language is highly mathematical in its form. This basic language is quite general and contains only few musical elements. This generality makes MUSICIAN also useful for many non-musical applications, especially to signal processing and mathematical-physical problems. In spite of its generality MUSICIAN deserves the title of a musical system through the ease and efficiency by which musical concepts such as FM-synthesis, non-linear distortion and other widely used music synthesis procedures can be implemented. The modularity of MUSICIAN is such that these procedures, once implemented, can be viewed as integral parts of it.

MUSICIAN is based on a number of concepts and features, some of which will be familiar from other systems, mostly non-musical. In particular we use

- APL’s concepts of an interactive language and work-spaces;
- non-procedural and functional programming style;
- aspects of concurrent programming (semaphores and data streams);
- finally, an array processor is used to speed up computations.

MUSICIAN is intended to run on small to medium computer systems. We have written its first version on a DEC PDP-11/24, with a CSPI Mini-MAP array-processor attached. Even with this rather modest architecture we are able to handle many music processing tasks in real-time. We plan to implement MUSICIAN on a micro-computer system soon.

3. The MUSICIAN System.

3.1. Interactive command processing.

MUSICIAN is an interactive system which holds a command-and-response dialogue with its user. The dialogue is held in a simple language, which, in common with many high-level languages is capable of expressing variables, constants, names, expressions etc.

---

3 MUSICIAN is currently used at ELSINT Ltd, MRI Division, for R&D and production software in as various fields as spectroscopy, magnetic field configuration, etc.
3.2 Array Expressions.

In MUSICIAN all data is numerical, and all numbers are elements in vectors. MUSICIAN uses most common operators and functions used in ordinary programming languages, redefined in some natural way as vector operations. For example:

\[ 2 + \{4,10,5\} \]

means, add 2 to each of 4,10,5 giving, in MUSICIAN notation, \( \{6,12,7\} \).

\[ \{1,2,3\} \times \{1,2,3\} \]

multiplies element by element to give \( \{1,4,9\} \).

Less trivial is the implementation of the indexing operator (notated by \[\] )

\[ A[5,4,1] \]

which gives the fifth, fourth and first elements of \( A \), in that order, or:

\[ 0:100[B] \]

which takes from the vector 0,1,...,100 the elements whose indices are given in \( B \).

3.3. Definition of Functions.

MUSICIAN comes with an extensive list of operators and primitive functions which are "built-in", but these are only given as a starting for the user's definition of his functions. Defining a function in MUSICIAN is as commonplace as adding 2 and 2. One simply takes an expression and gives it a name:

\[ \text{DEFINE STARTC C[1;10]} \]

which makes \( \text{STARTC} \) the first ten elements of \( C \). (Not necessity of \( C \)'s current value, but from whatever value may have when \( \text{STARTC} \) is invoked). The function expression is just an ordinary MUSICIAN expression (an array-expression, of course), with the added capacity of including formal parameters, denoted by \( $1, $2, $3 \) etc. For example in:

\[ \text{DEF AVERAGE SUM($)/#$} \]

($ defaults to $1, the first argument) the new \( \text{AVERAGE} \) function has one vector-argument, and its value is the sum of all elements in the arguments divided by the number of elements. \( \text{SUM} \) is a built-in function, but it just as well may have been another user-defined function, such as in

\[ \text{DEF VARIANCE AVERAGE(($-AVERAGE($))^2)} \]
which computes the statistical variance of its vector argument. This "stacking" of functions may continue without limit, in principle.

3.4 Non-procedural programming in MUSICIAN.

The combination of array-expressions with the ability to define new functions make MUSICIAN expressions very concise, elegant and powerful. It should be evident that the power of such expressions may to a large degree do away with the need for conventional, procedural programming. Some MUSICIAN operators have been included specifically with this aim in view, and can be viewed as "programming" operators, or rather as programming-replacing operators. These include the various forms of the sequence generator, the indexing operator ([]), the concatenation operator ([]), and the count-operator (#). The following example shows all of them in action in a function definition:

```
DEF ROTATE ($[#], $[1:(#·1)])
```

which rotates its vector argument circularly one element forward. In particular, loops, which are so common in procedural programming, tend to disappear when array-expressions are used, especially when the sequence generator operator is used:

```
DEF SINEWAVE SIN(0:(2·PI·$1) # $2)
```

which generates a specified number of periods of a sine wave, sampled evenly by a specified number of points.

In spite of this, we have not been tempted to depend completely on the use of non-procedural programming (as in pure Lisp), but have included such ordinary constructs as command procedures and conventional loops.

3.5 Built-In Functions

We have included in MUSICIAN about two dozen pre-defined functions, covering such diverse subjects as elementary arithmetic, trigonometry, complex numbers, random number generators as well as more specifically musical or signal-processing functions. These functions are all array functions, i.e. their arguments or result or both are vectors. Most of them are implemented in our prototype using an array processor.
3.6 Workspaces

Like in APL, we call the aggregate of all variables, functions and system settings the "current workspace" and allow it to be written or read as a unit to or from a storage device. We find the workspace concept very convenient. Indeed, a MUSICIAN user will not normally write a program in the ordinary sense, but will create a workspace containing functions, variables and procedures dealing with a specific subject.

4. The Background MUSICIAN

4.1. Working in the Background

The Background MUSICIAN is a key feature of the MUSICIAN system, which enables MUSICIAN to be integrated in a "production" system. This feature enables a complete musical system to take advantage of all of MUSICIAN's capabilities, without putting any serious limitation on the design of synthesizer architecture or user interface.

In the Background MUSICIAN mode, MUSICIAN works under the supervision of another program running concurrent. In this mode MUSICIAN will usually disappear into the background and let the concurrent process do the talking with the operator in whatever means it finds suitable. At the same time MUSICIAN continues to function in the context of its current workspace, but this activity is transparent to the operator. The two processes are linked together in several ways:

- MUSICIAN continues to process commands in its own usual syntax, but these now originate from its parent process on an internal software queue.
- MUSICIAN shares a common block of memory with its parent, thus enabling the two processes to share data. Specific MUSICIAN commands enable to map MUSICIAN variables on this common block, thus allowing the co-processes to define a common data-structure.
- A collection of semaphores (or event flags) are used to synchronize the processes and to inform one of the other process' current status.
- MUSICIAN's output stream is made available to its parent. (MUSICIAN's output stream is a specific data area which is filled by some MUSICIAN commands, and is usually used to connect the system to an audio-DAC).
- The parent task may switch temporarily to conversational mode, thus giving the operator direct access to MUSICIAN. MUSICIAN reverts to Background mode when the operator is done.
The Background MUSICIAN is essentially a programmer's option, since a front-program has to be coded. We cite a few possible applications to show its power:

- The front process may act as a controller of a musical keyboard and an array of editing function keys, such as in many commercial synthesizers. The front process' task is to translate the effect of these function keys into MUSICIAN commands and functions. These commands will be executed in the context of a work space written to implement the synthesizer's functions. Notice that the actual effect of the user interface may be modified at any time with no programming involved, by entering conversational mode and making alterations in the work space.

- The Background MUSICIAN may be used to connect MUSICIAN to a synthesizer which implements synthesis functions in hardware. This synthesizer may use a private communication protocol requiring data structures such as envelope or waveform tables, time scheduling parameters and so on. The front task is needed here to translate MUSICIAN's output stream into a format which will conform with the communication protocol. The front task's activities in this example consist of activating its synthesizer and MUSICIAN, then switching to MUSICIAN conversational mode. Throughout the rest of the session the synthesizer will be manipulated by the operator using MUSICIAN commands, while the front-task, now itself in the background, is continuously converting MUSICIAN's output stream to synthesizer control commands.

- The above two examples may of course be combined, to enable MUSICIAN to work both with a prescribed user interface and prescribed audio equipment.

- MUSICIAN's output stream is meant to be connected to an audio-DAC or some other audio equipment. For cases where real-time synthesis is not possible or if the audio equipment cannot be connected, a simple foreground task can be written to continuously record the output stream on a storage medium for offline playing.

4.2 Supporting features.

MUSICIAN contains several supporting features which make it a practical independent system:

- A graphic display of vectors, operating on a normal alphanumeric terminal. This representation mode is indispensable when working with large vectors.

- A facility is included for defining and executing command procedures in the MUSICIAN language.

- Special commands and functions enable the system to read data from any data set file, with only elementary restrictions on format.

- A sophisticated keyboard routine enable to edit commands as well as to recall and modify previously entered commands.
5. Conclusions and Further Research.

MUSICIAN is intended to run on small to medium computer systems. We have written its first version on a DEC PDP-11/24, with a CSPI Mini-MAP array processor attached. Even with this rather modest architecture we are able to handle many music processing tasks in real-time. We plan to implement MUSICIAN on a micro-computer system. Currently, MUSICIAN-II is being written in the language C. It is being developed on a VAX 780, but intended to run on an INTEL 310 with an AP-4 array processor. In this configuration we plan to use it as a host for the AMOS synthesizer, designed by Y. Jehuda and implemented in our laboratory. AMOS was presented at the International Computer Music Conference, Paris 1984 [Jeh85].

MUSICIAN-II was designed in summer 1986 [Nah86] and will be equipped with a composer-friendly interface which allows for modular definitions of sound characteristics in the style of programming in the large and which has an innovative-graphic presentation of sounds.

MUSICIAN-II is also intended to serve as a basis for an Instrument Specification System (ISS) which will allow to specify, define and modify musical instruments in a modular way and to use them in real-time after compilation. The latter is being developed in our laboratory as a Ph.D project by S. Markel [Mar86] and several student projects under the guidance of Dr. U. Shimony and the second author.

To conclude the paper we quote from [LA85, p.260]: "It seems that there are two streams of music language development: those that primarily address the complexity of music representation and are secondarily concerned with efficiency, such as Pla and FORMES, and those that emphasize efficiency in order to address real-time signal processing control, and are less concerned with high level representation, such as Moxie and 4CED. Ultimately this gap will have to be closed.

We think that MUSICIAN and its subsequent developments will do exactly this.

6. References


