Moxie for the Atari ST

D. J. Collinge and D. J. Scheidt

ABSTRACT

A real-time scheduler, Moxie, has been written for the Atari ST computers. With two primitives which are complete and orthogonal to the normal facilities of a programming language it provides efficient real-time management services. The abstract nature of the primitives allows the user to write modules which specify his own compositional form and processes in structures of time and activity. It is integrated into the kernel and syntax of a Forth system. The philosophy of Moxie is briefly discussed and examples are presented to illustrate the syntax used in this implementation.

1. Introduction

For the past several years we have been using a real-time system based on Forth to write programs for our various musical projects. The combination of Forth with the Moxie concept (Collinge, 1984) has proven to be a particularly productive way to express the musical structures we have needed. In this paper we discuss the applicability of Forth to musical problems and show how we extended it to include the Moxie primitives. We briefly outline the Moxie philosophy and give examples of the syntax of our most recent implementation for the Atari ST. Lastly, we give an account of the algorithms and data structures used in this implementation which will be sufficient to guide implementations of similar systems.

2. Forth and Music Applications

Forth has been a good language for musical applications for several reasons: a) it is relatively fast; b) it has a direct interface to machine language - essential for interface to custom hardware and for hand-tuning to maximum speed; c) its interactive user interface and modular nature facilitate the rapid prototyping and modification necessary in the absence of well-defined problems.

231

ICMC Proceedings 1985
specifications, and, d) Futh can pack a great deal of complexity into a small system. Among its proponents Futh has a legendary reputation for multiplying programmer productivity and our experience has borne this out.

For these and other reasons we believe that Futh is still a viable, and perhaps the best language for musical applications.

2. Event Schedulers

Event schedulers are systems for executing some particular code at a given time. Operating systems have utilized event schedulers for many years (e.g. Unix "cron") and they were applied to music in EML (Abbot, 1981), FORMULA (Anderson and Kuivisto, 1984) and Mixie.

An important philosophical principle behind these systems is that, because time is primary in music, users will not be satisfied unless they have access to the most fundamental time management functions. The compositional methods and procedures of sophisticated users may penetrate deeply into the domain of time and they will need the most fundamental controls available.

A related principle is that time is expressed in the abstract only, in its own terms, without any traditional music or other language. Computers with their own conception of how their music is structured in time will not need our except the prefigured formulation of others.

Since time control is presented at a fundamental level and is orthogonal to other capabilities, users can add modules to express the time structure most useful to them. Others can use these modules or write their own as they see fit.

Some utility modules, such as MIDI drivers and parsers, are supplied because of their general applicability. Each user is likely to have some specific hardware configuration that calls for a custom driver module. For example, one of us makes musical systems that respond to musical input and needs a special module to handle input from the listening device. The other makes responsive installations and needs modules that control custom sensing and sound production hardware. The simplicity, power, and orthogonality of the Mixie time primitives allow us to use the same system for radically different musical purposes simply by adding the modules we need and defining our music in terms of them.

3. The Music Time Management Primitives

The Mixie primitives have been discussed in detail elsewhere (Culley, 1985) and are only sketched briefly here. The Futh system is different from that of the earlier paper so detailed but simple examples are given. More complex examples are distributed with the system.

ICMC Proceedings 1988 232
The basic unit of processing in Moxie is the event. An event is a procedure, called the action, executing at a particular time. Usually the actions have been scheduled to happen at a certain time and are executed by a process which measures and keeps track of real time. The function of this process is to organize the processor so that all the events happen at their scheduled time, regardless of how many other events or other activities happen around it.

External activities may also cause events: typing input, MIDI input, sending devices, or other external events become Moxie events when their time of occurrence is measured and they execute an action.

Any code, including actions, can ask Moxie to generate an event at a particular time by using "cause". The parameters of "cause" are the action to perform and the time at which it is to be performed. The time can be stated either relative to the time of the cause or as an absolute time.

The procedures of conventional programming languages have parameters to give them more generality. Actions may also be generalized with parameters. The parameters of an action are supplied to the "cause" when an event is generated and become part of the event. This means that the values of the parameters at the time of the cause are suspended until the event occurs, then supplied to the action. Information can thus be carried through real time from event to event independently of what occurs in the period between the events. Parameters are responsible for the power of Moxie; they make possible multiple independent instances of event structures.

5. Moxie and Forth on the Atari ST

An event scheduler can be added to almost any programming language but Forth is particularly suitable because of its extensibility, modularity, and accessibility. We have implemented a Moxie system for the Atari ST and integrated it into the Forthbased Forth system by Bradley Forthwaite. This is the same system used to implement FORMULA, another music system for the Atari ST.

The following discussion of the Moxie concept refers to the Atari ST version but, except for the syntax, also applies to several other Moxie systems. The MIDI interface used in the examples is one of several such useful modules supplied with Moxie for the ST but which are not discussed because they are peripheral to the Moxie concept.

5.1. Actions

The actions are essentially Forth word definitions but are defined slightly differently because they have named parameters, which color definitions do not normally have. Here is an action which sends a MIDI message to turn on key number 49:

233 ICMC Proceedings 1988
action: ckeyoff
40 3 midinotoff
; action

Actions also may take named parameters. Normally, Forth
words are executed sequentially and take their parameters on the
data stack. Actions, however, may be executed at any time and
their parameters must be kept safe in memory called a parameter
block. An action with parameters which turns off a given key on a
given MIDI channel:

action: keyoff at fkey; fchan
fkey & fchan & { get the parameters... }
midinotoff { ... and send message. }
; action

5.2. Cause

The scheduler is invoked with the cause request, which gen-
erates an event which will occur at some time in the future. The
time can be specified in two ways, relative to the current
time, or relative to the time of initialization of Moxie, which we
call absolute. We have chosen the millisecond as a sufficiently
small unit of time. A sample cause request in which the time is
relative:

keyoff event
43 fkey !
12 fchan !
1000 relative
; cause

The word "event" establishes a context in which the tie named param-
eters are accessible as if they were ordinary Forth variables.

Actions can cause other events. An action which will produce
notes with a given duration:

action: note at nkey, nnot
nkey & | midinotoff { always channel 1 }
keyoff event
nkey & { Get note's key... }
key ! { ... and give it to keyoff. }
nnot 0 relative
; cause
; action

Now, if we schedule "note" with the code:

note event
40 nkey ! 1000 nnot ;
100 relative

ICMC Proceedings 1988
in 100 milliseconds "note" will turn on key number 40 and 1000 milliseconds later "keyoff" will turn it off. The same action can be used to generate many overlapping notes since each "cause" generates its own state to represent the event. The multiple instances of the parameter "dur", for example, are independent and do not interfere with each other.

5.2. Processes

A process is represented as a sequence of events formed when an event repeatedly schedules a similar event. The actions frequently reschedule themselves using "cycle", which causes the same event to be rescheduled after a given delay. This differs from "cause" in that cause generates a new event which disappears along with its parameters after it occurs; with "cycle" the event recurs with the values of the parameters as they were left at termination of the action. This effect could also be produced by using "cause" to schedule a new event and copying all the parameters to the new event but "cycle" is more convenient and efficient.

A very simple process which plays ascending chromatic scales starting and ending on given notes, each note with a given duration:

```
action: scale p( current last dur )
  note event                           [ New note ASAP. ]
  current @ nikey !
  dur $ ndur !
  0 relative
  cause
  1 current +1                          [ Increment note. ]
  current @ end & @                    [ If more notes to play... ]
  if dur $ & cycle                     [ ... reschedule this action. ]
    . then                               [ Else, don't; i.e. stop. ]
  else                                 [ End of "pit". ]
  jack
```

The state of the process is maintained in the parameters so multiple instances of a process may co-exist without contention among the variables. Therefore, as many scales as desired can be started up and will run and terminate independently. As in the "note" and "scale" examples above this fact can be used to write general-purpose modules for generating and controlling structures in time. Because the events are independent of each other these modules are free of the re-entrancy problems that plague other approaches to time management.

6. Internals

The simple and elegant interface of "action" and "cause" conceals a kernel of considerable sophistication. A description of the
seen machinery follows.

6.1. Event Descriptors

The central data structure hidden by Moxie is the Event Descriptor. An Event Descriptor contains all the information necessary to specify an event: the time of the event in milliseconds from initialization, the action to invoke, the values of the parameters to be passed to the action, and some control bits and values whose function is explained below. Event descriptors are generated by "cause" and represent knowledge of real time.

6.2. The Event Queue

Events which have been caused but have not yet occurred are called pending events. Pending events are organized into a queue of linked Event Descriptors sorted in order of their time of occurrence. When a new event is generated by "cause" it is inserted into the Event queue so as to maintain the sort.

6.2. The Event Dispatcher

Accounting of real time is done by the Event Dispatcher. It has access to some kind of interval timer for timing the intervals between events. In this implementation the interval timer simulated with a periodic interrupt but in other implementations a hardware interval timer was used. The Event dispatcher simply sets the timer to interrupt when the next event is due. When it interrupts the timer is set for the time of the next event.

6.4. The Foreground

Every Event Descriptor which has been processed by the Event Dispatcher is no longer pending; it is due; its action should be executed as soon as possible. Usually there is only one due event - the one just processed. Each time the Event Dispatcher processes an event it calls the Foreground which executes the actions of the due events. When there is only one due event the Foreground simply executes the action and returns to the Event Dispatcher.

6.5. Parameters

Parameters are kept in fixed-size blocks called parameter blocks. The blocks are of fixed size to simplify the task of memory management. The size is chosen to accommodate most needs; if more parameters are needed then more than one block is allocated to a single set of parameters. Since all allocation operations use the parameter blocks allocation is done from a free list and is very fast.

ICMC Proceedings 1981 236
6.6. Processes

Processes, such as the scale example, are represented as sequences of events. The conventional representation is that used in operating systems: each process is a separate thread of execution with its own state kept in a program counter, variables, and stacks. In this model the process is synchronized with real time using the "wait" primitive. When a new thread is generated (e.g. Unix fork()) new copies of the stack must be allocated to keep the state of the new process separate from other instances. Allocating and freeing variable amounts of memory is an expensive operation so generating a new process is either heavy or is done statically.

When processes are represented as Moxie events all the state is kept in the parameters. All actions run to completion so the stack does not carry any information from event to event. This allows Moxie processes to be dynamic: the programmer need not worry in advance how many scales, for instance, the program might generate.

6.7. Background

When there is nothing left for Foreground to do it returns to whatever was interrupted. This activity is called the background and is used for continuous low-priority processes that use surplus cycles.

6.8. Out of Real Time

Moxie can operate perfectly only if the processor is infinitely fast. Since we never seem to be able to afford processors with quite that performance the scheduler must compromise when processor resources are scarce.

When many events are scheduled in a short period of time the Event Dispatcher may produce more due events than Foreground can diagnose of in the processor cycles available. When this happens Foreground chooses among several due events on the basis of importance and latency. Importance is represented in the Event Descriptor as a priority which can be assigned when the event is caused. Foreground chooses the latest event among those with the highest priority. If priorities are not given all user-generated events are given the same priority and latency is Foreground's only criterion.

Since the Event Dispatcher has very high interrupt priority real time is maintained even when the processor is very busy. Normally Foreground slips out of real time only for a short time, too short to be perceptually significant. In some situations Foreground can monopolize the processor leaving no cycles for the background. This is undesirable since the background normally runs operator interface functions, such as shell, with which an operator could correct the conditions leading to the overload. Therefore, Foreground measures the processor time it uses and makes sure that the background gets no less than a user-specified fraction of the
2. Conclusion

The productive working environment, modularity, and extensibility of FORTH make it a good language for music programming. Extending the FORTH toolbox with the simple Moxie concept adds powerful time structures to its repertoire cleanly and without extraneous conceptual formulations. We have used the combination of FORTH with Moxie for a number of projects and find it powerful and adaptable to a wide variety of approaches to the problems of using a computer to make music.

REFERENCES


AVAILABILITY

This system is available for the cost of the media from the authors. Send a 50 mm disk in the format of your choice to:

D. J. Collinge,
School of Music,
University of Victoria,
P.O. Box 1700,
Victoria, B.C.
CANADA

V9W 2T2

Mark the format on the disk label in case the format is disturbed accidentally in transit.