Random Access to the Time Domain in the AMPLE language

Chris Jordan
Hybrid Technology Ltd
Cambridge
Great Britain

ABSTRACT

AMPLE is a powerful and versatile music programming language and environment for use on microcomputers. The language 'nucleus' includes a wide range of computation and sound control functions for advanced musical applications, linked to an innovative textual music notation for more traditional musical forms. Higher-level components of the environment cater for specific user requirements such as instrument design, real-time performance control and staff notation, and low-level software interfaces provide unified access to a variety of musical input and output devices, both via MIDI and direct host-specific connections. Currently, the most popular use of AMPLE is in the Hybrid Music System for the BBC Microcomputer, including the Music 5000 Synthesiser.

AMPLE is a procedural language with a 'word' structure like that of LOGO - the user program is a hierarchy of words, each defined in terms of pre-existing user words and pre-defined system words. Words can be created and edited individually, interactively, and in a variety of forms, always including the traditional textual program form.

The user program runs in real time under the control of a time manager that merges and sorts events from all concurrent processes before playing. This allows random access to the time domain (over a range limited by system load and memory capacity), so that as well as the usual positive time intervals between events, negative time intervals can be accomodated. Thus, programs and scores are free to generate event sequences in non-time orders that better suit the algorithms of the program or score. The major reward is simpler programming of a variety of overlapping musical structures, both large and small scale, both via low-level sound control words and, in particular, AMPLE's textual music notation.

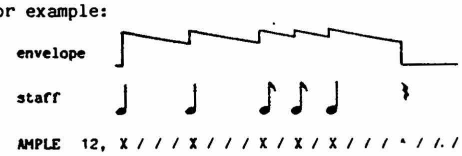
Simple applications of the random time access facility are described, along with related features of the AMPLE textual music notation.

NOTATION ELEMENTS

AMPLE music notation has words (symbols) for four basic types of musical event:

word	function
letters A-G & a-g	notes of different pitches
X	hit (pitchless note)
^	rest
/	hold (continue last event)

Other words set the parameters of events, the most important here being ',' (comma) which sets the length in timebase units. In scoring, a basic 'beat' length is set with ',', and multiples of this are achieved by extension with the / word (hold). For example:



On playing, X (hit) expands into a 'gate on' event followed by a 'duration' (time interval). ^ (rest) expands into a 'gate off' and duration, and / into just a duration. In each case, the length of the duration is that set by the previous ',', in this case 12 units.

Hence, this simple music event sequence translates into a sound event sequence of just 'gate on' and 'gate off' events with separating time intervals.

The full set of AMPLE music words is described in the Music 5000 User Guide (1986).

NEGATIVE LENGTHS

If a negative length setting is in force, the duration of each music event moves the 'time pointer' back rather than forward, so the next event will play before rather than after. Hence, the example could be re-scored backwards as follows:

```
-12,///^ ///X /X/X ///X ///X
```

The sounding result is the same, but there is a net backwards movement in time rather than a forwards one, so subsequent music events will be displaced.

BACK-HOLD

The back-hold word, \ , has the effect opposite to that of the hold word, / . It has the effect of a hold with the current length negated, so, for example, the sequence

```
/// \\\
```

has a zero net result.

One application is in the programming of the 'overhanging pick-up' musical structure - a short sequence of notes that leads-in to a section of a piece, for example a verse of a song. This can be a problem because the pick-up is temporally part of the previous section, but must be functionally attached to its own section if this is to be called-up as a procedure in different contexts.

The negative duration ability lets the pick-up be included in its rightful functional place by allowing back-spacing from the section start to the pick-up start. Using back-holds, this is quite clear, for example:

```
12, \\\ def  G/// /fed c///
    _______  ______________ _ _
     pickup  section proper
```

The pickup is quite free to overlay events in the tail end of the preceding section, generated by the same or another process.

CHORDS

Round brackets are used to denote chords, in which subsequent musical events of a group play alongside the first on successive voices. The subsequent events appear as a bracketed group after the first ('main') event, for example:

```
tune:            C     C     /     D     ^
chord sequence:  C(EG) C(EA) /(F/) D(FA) ^(^^)
```

In this simple example, the C, E and G in the first group play together as a C major chord. In the third group, one voice starts a new note while others are held, and the final chord of rests silences all voices.

For various reasons not discussed here, it is desirable to have a syntax like this in which additional notes are added after an unchanged main note. The availablity of negative lengths allows this to be implemented by simple definitions of '(' and ')', with no need for program look-ahead or retention of the last duration for possible retraction. The simplified actions of the bracket words are as follows:

word	action	comment
(	n, \	move back to main event start
	0,	make further events 'stay put'
)	m, /	restore net duration of main event
	n,	restore original ',' setting

where

n = ',' value of the main event
m = n + total duration inside brackets

Normally, m = n since the length of bracketed events is 0.

Additional actions (not shown) control voice selection.

BROKEN CHORDS

The bipolar duration implementation of chords allows more advanced chord-like groups to be scored very simply.

The programmer may explicitly use ',' inside the brackets to set a non-zero length so that there is a time interval between bracketed events. This gives broken (arpeggiated) chords:

```
          normal chords       broken chords

          48, C(EGB) F(ACE)   48, C(8,EGB) F(8,ACE)

Voice 4   B-----E-----            B-----E-----
Voice 3   G-----C-----          G-----C-----
Voice 2   E-----A-----        E-----A-----
Voice 1   C-----F-----      C-----F-----
```

The brackets ensure that the net duration is always that of the main note. If the total length inside the brackets exceeds that of the main event, then a net negative duration is needed - the 'strum' overlaps the next main event:

```
          48, C(24,EGB) F(24,ACE)

Voice 4            B-----E-----
Voice 3         G-----C------
Voice 2      E-----A-----
Voice 1   C-----F-----
```

Alternatively, the arpeggiating ',' setting can itself be negative, so that the strum anticipates the main event:

```
          48, C(-8,EGB) F(-8,ACE)

Voice 4   B-----E-----
Voice 3    G-----C-----
Voice 2     E-----A-----
Voice 1      C-----F-----
```

ECHO

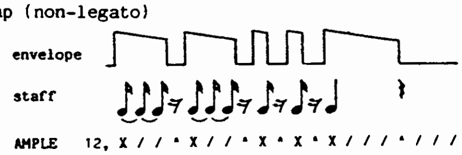
Echo simulated on successive voices is very similar to broken chords from the point of view of time domain access. The difference is that the events that are directed to successive voices with successive time displacement are copies of the main event, rather than additional scored events. The word 'Echo' (provided in a special effects extension) employs AMPLE's music event vector to generate copies of each event and displace them in time using / and Ω. The interval between echoes (positive or negative) and the number of echoes can be specified.

GATE ARTICULATION

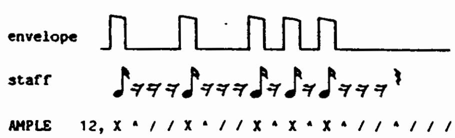
A particularly illustrative application of random access is simple control over articulation via the gate signal. Note-separating gaps of fixed length (non-legato or 'portando') are achieved by each note or hit reaching back to prematurely terminate its predecessor. Staccato style is achieved by the note or hit reaching forward to terminate itself a fixed time in the future.

Both these reaching actions have a zero net change on the time position, so can simply be added to the standard interpretation of music events using the event vector. The articulation actions can themselves be expressed in terms of simple music events - rest, hold and back-hold. This is more obvious when the articulations are thought of as rests added between notes:

gap (non-legato)



staccato



An interesting observation is that the articulation action is a simple form of 'macro' action; one that expands each music event into a sequence of music events with derived parameters, normally used for more advanced compositional processes.

The simplified definition of the 'gap' action is as follows:

```
ACT(        % start vector sequence
4, \^       % add leading rest
ACT         % perform default note action
)ACT        % end vector sequence
```

Notice that the second line is identical in essence to the over-hanging pickup score - each separating gap can be considered a rest that over-hangs from the following note.

The staccato action is:

```
ACT(        % start vector sequence
-4, \^      % add trailing rest
ACT         % perform default note action
)ACT        % end vector sequence
```

An important point to note here is that it is the sounding portion of the note that adopts the fixed length of the rest, and the silent 'rest' portion is the variable-length remainder - the interval between the rest's 'gate off' event and the next note's 'gate on'. The duration of the rest is negative, serving to return the time position to the start of the note.

The music event sequence on the second line of the definition is not only the most compact representation (no back-rest is available) of the function, but also illustrates the commonality in structure of the two actions. In practice, both these actions are performed by a single word 'Len' (supplied in a special effects extension), which takes the length setting as a simple, postive or negative numeric argument.

Since durations can pass over existing events, there is nothing to prevent a gap or staccato interval being longer than the notes themselves. Under these conditions, each note reaches beyond its immediate predecessor and successor to modify events further removed in the time stream. Complicated interactions of note length and sequence arise, giving interesting articulation and phrasing effects. There is clearly much scope for further experimentation along these lines, for example, the use of multiple actions controlling a range of parameters as the basis for musical sequence generation and transformation in advanced compositional processes. This is well within the capabilities of current implementations of the AMPLE language.

REFERENCES

Music 5000 User Guide (1986). Hybrid Technology Ltd, Cambridge, Great Britain