The Integration of Real-Time Synthesis into HMSL, the Hierarchical Music Specification Language.

Phil Burk (phil@mills.edu)
Center for Contemporary Music, Mills College,
5000 Macarthur Blvd., Oakland, CA, 94613 USA

Abstract
This paper describes the integration of real-time 5000 based synthesis into an existing music composition and performance language. The system currently runs on the Macintosh and the Amiga and supports osc or osc and Digidesign DSP cards or GMP 5000 kits. The 5000 software includes a library of synthesis units. An optimised oscillator is described in detail. The host resident software consists of a device independent DSP driver and object-oriented tools for connecting and controlling the synthesis units. An example composition involving a MIDI network, an interactive graphics screen and dynamic tuning is analysed.

Introduction
HMSL is a Fortran based, object oriented programming language created to help composers explore experimental techniques in music. It was designed to support arbitrary methods of synthesis and its development was started before MIDI synthesizers became available. The compositional tools that are in HMSL are based on abstract notions of composition. It supports concepts like "notes" and "pitch bend" because that is the prevailing compositional paradigm. The composer, however, is free to use any form of output available. HMSL compositions have used electric motors, graphic devices, solenoids, text files, and MIDI synthesizers, as output devices. With the availability of inexpensive digital signal processors, HMSL can now use real time software synthesis to create sounds. This obviously opens up new possibilities for composition and performance that were impractical using MIDI.

The DSP Toolkit in HMSL supports the Audio Media and Sound Accelerator cards from Digidesign, as well as the GMP 5000 kit from Digidesign. We also plan to support the soon to be released 5000 cards for the Amiga. Using this system, it is possible for the composer to design sound generating circuits that can be controlled from HMSL, along with MIDI or other output devices. The circuits can be designed at the host level using a library of existing unit generators, or can be written directly in 5000 assembly code using the Motorola assembler. We have found it possible to create several voices per card, depending on the execution time per voice.

Software Architecture
The 5000 based software consists of a library of unit generators (oscillators, filters, etc.), an active circuit control system that executes currently generating sounds, an output buffer (FIFO) feeding the Digital to Analog converter, and Post Interrupts that communicate with host software.

On the host side, there is a device dependent driver at the lowest level. Above that is code for loading object files, allocating 5000 memory, building 5000 circuits, and controlling this code from HMSL.

Units
Sound is produced by connecting various units, for example: oscillators, noise sources, filters, envelopes, etc. A unit consists of a 5000 code macro or subroutine, an associated 5000 data structure, and a host resident template that describes the 5000 data. By convention the address of this data record is passed in address register R0. The data is arranged so that it can usually be accessed by post increment addressing.

Frequently changing parameters cannot be passed in the A and/or B accumulators. For example, a simple digital oscillator has this data record:

```
OFFSET Contents
0 Current Phase, 0 - 1.0
1 Size of Wave Table
2 Address of middle of wave table
```

The frequency is passed in the A accumulator and the output sample is left in A. This is convenient when data is passed down a chain of units. Here is the 5000 assembly code for a non-interpolating oscillator:

```
Oscillator
MOVE X: (N0), X1 ; get current phase
```
ADD X1,A       ; add phase to phase increment
MOVE A1,X0    ; wrap around -1,1, without limiting
MOVE X0,X:(R0)+ ; update phase in memory
MOVE X:(R0)+,Y1 ; get size/2
MOVE X:(R0)+,A ; get address of middle of wavetable
MAC X1,X5,A   ; amp=addr = (size/2) phase + mid addr
MOVE A1,R4    ; move to address register, fraction in A0
NOP            ; allow R4 to settle, how unpleasant
MOVE Y:(R4),A ; get sample from Y memory
RTS

This oscillator can be thought of as a sawtooth generator followed by a waveshaping lookup table. The MOVE A1,X0 is used instead of A,X0 because that would cause the value in A to be limited to a value between -1.0 and 1.0. Thus our sawtooth would go up to 1.0 and stick there instead of wrapping around to a negative value. When A1 is moved there is no limiting. The NOP (no operation) is required because the address in R4 has not settled yet due to pipelining. In real time synthesis every cycle counts so this NOP is painful.

Optimization using Parallel Moves

In order to allow more oscillators to be active simultaneously, I have written a unit that does two oscillator calculations in parallel. This takes advantage of the parallel move instructions of the 56000. This is done by putting one unit data structure in X memory and the other in Y memory. One frequency is passed in A and the other in B. The results are returned in A and B. Now instead of getting 1 oscillator in 10 instructions, we can get 2 oscillators in 12 instructions.

In the interpolating version of this oscillator, we can calculate 1 oscillator in 14 instructions or 2 in 20.

Host Interface

The interface between the Host software and the 56000 software is through the 56000 Host Interface Port. This appears as a memory mapped peripheral to both computers. The lowest level of this code is device dependant. For the Macintosh, there is code that opens the Digidesign device driver and allocates one or more cards for use by JinML. It also gets the address of the Host Interface Port on the 56000 chips. The code that interfaces to the OMP Kit is dependant on the technique used to access the hardware. On the Amiga, I have developed an interface that uses the printer port as a multiplexed 8 bit address and data bus. At the top level of the driver are words for reading and writing the HI registers of the 56000. Above that level, the code is device independent.

The primary interface words allow the host computer to read or write individual X,Y or P memory locations in the 56000. Here, for example, is the word to store into X memory:

DSP.X! (24-bit=VALUE 56000-X+ADDRESS -- )
(In Forth, words are documented using stack diagrams. Input parameters are to the left of the `::` and output to the right.) These words are implemented using Host Command Interrupts. One Interrupt is used to pass an address in the TX register. The other interrupts use this address to fetch or store memory. The address is autoincremented after each use to allow easy block moves.

Using these memory access words, the host can download software to the 56000, then control the synthesis by changing parameters in data records. The allocation of data records in 56000 memory is managed by the host. This is to avoid placing a burden on the 56000 and because the host must know where everything is anyway in order to control synthesis.

**Object Oriented Host Control**

HMSL is based on an object oriented dialect of Forth called ODE. The host based HMSL code keeps track of 56000 based circuits by defining classes of objects corresponding to various types of circuits. A circuit is a collection of units patched together in software. For efficiency circuits are divided into audio rate circuits (eg. filters, oscillators) and control rate circuits (eg. envelopes, LFOs). Control circuits are executed once for every 16 executions of the audio rate circuits. Circuit classes keeps track of how much X and Y memory is needed for data and can allocate that memory when a circuit is downloaded. Here is a description of some of the methods supported by circuit classes:

**COMPILE:** generate and download 56000 code described using a simple macro language. The language supports unit generator calls, moving data between memory and registers, and scaling and mixing of parameters.

**MAKE:** allocate space in X and Y memory.

**SETUP:** set parameters in 56000 memory to reasonable defaults.

Circuits also have generic methods for specifying frequency or other parameters which are found in most circuits, eg. PUT_FREQ or PUT_LEVEL. This allows us to write pieces that send generic messages to any of a variety of different circuit types.

**Example: a simple modulated oscillator circuit**

As an example, let's examine some of the code necessary to construct a circuit with one audio rate oscillator whose frequency is modulated by a single control rate LFO. Here is the host data structure that describes what units and plugs are needed by the circuit.

```
UNIT U_SIMPLE_VOICE
  plug usv_lfo_rate
  unit u_oscilLor usv_lfo
  plug usv_lfo_depth
  plug usv_osc_freq
  plug usv_new_freq \ modulated by LFO
  unit u_oscilLor usv_osc
  plug usv_mix

UNIT
```

This describes the amount of data memory needed by each circuit and the offsets of individual parameters within each circuit. Using this description we can now write a COMPILE method which will use 56000 code macros to feed the parameters to the unit oscillators and to perform simple scaling operations.

```
\m COMPILE: ( -- , compile into 56000 )
\ Control Rate code
56k
  usv_lfo_freq x+ 56k:x0 \ move lfo freq to A register
  usv_lfo.x+ 56k:oscillator1 \ calculate lfo
  \ new_freq = F0+depth + osc_freq
  usv_lfo_depth x+ usv_osc_freq x+ 56k.**
  usv_new_freq x+ 56k.x7 \ save for audio rate circuit
  56k:rts
\ Audio Rate code
56k
  usv_new_freq x+ 56k:x0 \ modulated osc freq to A reg
  usv_osc x+ 56k:oscillator1 \ calc oscillator
  usv_mix x+ 56k:scale \ scale output
```

ICMC 17
Consider the control rate code. The 56K.xr@ macro compiles a MOVE from the particular plug specified to the A register where it can be fed into the LFO as a frequency parameter. The 56K.xr@ compile compilation of several instructions including a MACRO to achieve the necessary scaling and offset. Once the COMPILER and other methods have been defined we can create as many of these circuits as memory allows. The only real limitation is that eventually the 56000 will run out of time if we play too many at once. Here is some code that instantiates two of those circuit objects and then loads and tweaks one of them.

```lisp
; \ mix with other voices
; 56x.rts
; 56x.audio

ob.simple.voice USV1 ( create two circuits )
\n;\ simple.voice USV2
; \( * \) change sound of usv1
load; usv1 ( calls make: compile: and setup: )
on: usv1 ( starts execution of circuit
\begin ( play notes in just intoned scale )
\many: tuning choose ( select pitch index
\) calculate.frequency rut.free: usv1
\200 choose $ 200 * put.rxt: usv1 ( LFO )
\2 choose 1+ 12 * \#delay ( for 12 or 24 ticks )
\until ( until a key is hit on the keyboard )
\off: usv1
\n\)
\This was a simple example of using Fort+ to directly control the circuit. In a larger composition, a hierarchy could be created that contained information to be sent to the circuits. A case of object called an INSTRUMENT would translate the control information (pitch indices, etc.) to circuit parameters, perform voice allocation, and send the information to the 56000.

Example Composition using Relative Just Intonation

The author has written a performance piece called RELNET using this system. It was first performed March 16th, 1991 at Mills College by the author and Steve Curtis. RELNET uses the concept of relative versus absolute just intonation. In absolute tuning, pitches are whole number ratios of a fundamental pitch. In a relative tuning, a new pitch is a whole number ratio of the previous pitch. This piece is performed using two or more host computers linked together in a MIDI ring network. Each composer has its own DSP system. Note event tokens are passed around the network. Each token consists of a system exclusive message containing the frequency of the previous note, and its duration. When a host computer receives a token it calculates a new pitch, plays it, then passes a token to its neighbor. The output token is delayed by the note duration. The performer can specify a series of whole number ratios using a mouse driven graphical interface. Ratios are selected in sequence from those ratios marked as "on". The performer can also specify ratios for note duration that are relative to the previous duration or based on an absolute duration. Carrier modulator ratios for FM synthesis can also be specified to control timbre. The 56000 circuits for this piece consist of a simple FM pair with an amplitude envelope. The performer can also control the creation of new tokens, e.g. incoming tokens, and switch between relative or absolute pitch and duration.

56000 based synthesis was found to work well for this piece because of the very accurate control over pitch that is possible. The frequency of the output of the oscillator is given by this equation:

\[ \text{frequencies} = \text{SampleRate} \times \text{FrogParam} / 2.0 \]

FrogParam is a 24 bit fixed point number between 0.0 and 1.0. It turns out that for pitches around 440 hertz, we have a pitch resolution of around 1/1000 of a cent. The few MIDI synthesizers which support alternative tunings typically have a resolution of only about one cent. Although this high resolution is well beyond a human's ability to resolve pitch, it can be useful particularly when generating accurate beat frequencies.

References

ICMC 18