A Framework for Developing Signal Processing and Synthesis Algorithms for the Motorola 56001
Kurt J. Hebel
Symbolic Sound Corporation, P. O. Box 2530, Champaign, IL 61825-2530
Tel: (217) 328-6645  Electronic Mail: symbolic.snd@applelink.apple.com

The developer's version of the Kyma System provides an object-oriented framework for interactively developing and testing digital signal processing and synthesis algorithms written in the assembly language of the Motorola 56001 digital signal processor. There are several advantages to developing code within this framework. The framework handles memory allocation, input and output functions, and task scheduling; the programmer develops short code segments accomplishing a single function and plugs them into the framework to test them. The multiprocessor hardware (the Capybara) provides the computational power to develop and test these segments interactively. The large set of code segments already contained within the framework allows the programmer to quickly test a new algorithm on a variety of input signals and in a variety of contexts.

Introduction
The Kyma System is a highly flexible, open-ended environment for sound computation. Among its strengths are its direct manipulation user interface, and real-time software (direct) synthesis capabilities [1, 2, 3, 5].

The Kyma System is composed of software — the Kyma language — and hardware — the Capybara. The Kyma language combines software synthesis, digital recordings, real-time processing of A/D inputs, MIDI, and algorithmic composition in one environment. The Capybara is a high performance parallel processor containing from two to nine Motorola 56001 digital signal processors.

The Kyma language is based on objects called Sounds that represent streams of digital audio samples. Sounds are analogous to functions — all Sounds are either 0-ary functions (and are therefore called atomic Sounds) or are functions of one or more other Sounds (therefore called composite Sounds). Because of their functional nature, Sounds can be combined or shared with other Sounds to construct complex networks that can describe any level of detail from signal processing to compositional processes [6]. This functional representation also makes it possible to partition the sample stream computations for execution on the multiple processors of the Capybara.

The generality of Sounds allows them to represent any stream of digital audio samples including entire compositions [4].

Every Sound object is an instance of a specific class. The class defines the structure and behavior shared by all of its instances; the structure contains the parameters of the Sound, and the behavior describes, among other things, how a Sound of that class produces its stream of samples.

One can create new classes of Sounds from a combination of other Sounds. In this manner, signal processing and generation algorithms can be defined in terms of pre-existing Sounds [7].

There are times, however, when one would like to develop algorithms directly in the DSP assembly language: Combinations of small general purpose Sounds are not as efficient as a highly specialized monolithic Sound, and it may not always be possible to construct an arbitrary algorithm out of the built-in Sounds.

This paper describes extensions to Kyma that provide an object-oriented framework for interactively developing and testing digital signal processing and synthesis algorithms written in the assembly language of the Motorola 56001.

The Kyma Virtual Machine
In the Kyma language, the signal processor is treated as a virtual machine; that is, a computer whose "machine language" consists of digital signal processing and synthesis algorithms (e.g. Sin, Product, Oscillator, SineOrderFilter, etc.). When a Sound is played on the Capybara, it is compiled into a program consisting of sequences of these machine language instructions.
All signal processing and generation algorithms must be made up of some combination of those instructions. Each instruction corresponds to a Sound class in Kyma; the instruction, known as a MicroSound class definition, implements an algorithm that computes the next sample of the Sound's output stream.

By using the virtual machine model of the hardware, the Kyma language remains independent of the actual signal processing hardware. Since the virtual machine instructions correspond to Sound classes, compilation is a translation of one network of Sounds into another network containing only Sounds that have corresponding MicroSounds.

The Virtual Machine Implementation

One of the main benefits of using a virtual machine is the insulation of large portions of the software from the specific details of the signal processing hardware. Since Kyma builds programs in the virtual machine's instruction set, adding new signal processing and generation algorithms is the same as extending the instruction set of the virtual machine.

The virtual machine interpreter can be divided into two parts: the Executive and the virtual machine instruction set (see Figure 1). The Executive is responsible for handling communications among the various DSPs and input/output devices (such as MIDI and the I/0 converters). It performs task scheduling and memory allocation as well as fetching and decoding the virtual machine instructions and their operands. The virtual machine instruction set is a collection of 5601 subroutines; each subroutine corresponds to a virtual machine instruction.

Adding New MicroSound Classes

There are several decisions to be made when designing new signal processing or synthesis algorithms.

First, one must decide on how to partition the problem. The built-in Sounds can often be combined to yield a solution. It may be that only one small portion of the algorithm needs to be coded in assembly language. The object-oriented nature of Sounds and MicroSounds encourages greater reusability of small, general-purpose MicroSounds, while larger, more specialized MicroSounds offer more efficiency. Additionally, the computation of an algorithm made up of many small MicroSounds can occur on different processors in parallel, whereas the computation of a monolithic algorithm must occur on a single processor. All of these considerations must go into the decision of how to factor the algorithm.

Second, the assembly language portion of the algorithm must be coded. This involves deciding on the number, kind and order of parameters in the MicroSound. The user MicroSound browser (see Figure 7)...

![Figure 7. The user MicroSound browser](image)

The top portion of the browser lists all of the known MicroSound definitions; the bottom portion is a text editor on the assembly language of the selected MicroSound class definition. Shown here is a definition for a simple exponential signal generator.
This algorithm is shown in the MicroSound subroutine in Figure 2. We have decided that the two parameters, the current amplitude value and the amplitude decay factor, occupy consecutive X memory locations in the MicroSound. The Executive calls the subroutine with pointer registers set to point to the parameter values (called inputPtr), any input sample streams, (called inputPtr2) and the address to write the output sample (called outputPtr).

Lines 1 and 9 of the assembly language are required to name and delineate the algorithm definition. Lines 2 and 3 fetch the current amplitude and the amplitude decay factor values from the parameters and line 4 computes the next amplitude value. Line 5 saves the amplitude value back into the MicroSound’s parameters, and lines 6 and 7 output the left and right channel values.

Since this MicroSound is atomic, i.e., it does not operate on an input sample stream, we will use the DSPProgram (rather than DSP-ProgramWithInputs) Sound. The assembly language expects the amplitude value in the X half of the first parameter word, and the multiplication factor for the exponential decay in the X half of the following word. To generate a decaying exponential that starts at 1 and decays at the rate of 50% per sample we could use the following Smalltalk-80 code in the DSPProgram Sound to specify the initial parameter values:

\[
\text{maxAmp \cdot factor = maxAmp} \cdot \text{SignalProcessor maximumAmplitude.}
\]

\[
\text{factor := 0.5 \times maxAmp rounded.}
\]

\[
\text{self initialValueAt: 0 put: maxAmp \cdot factor \cdot outputPtr: 0.}
\]

\[
\text{self initialValueAt: 1 put: factor \cdot outputPtr: 0.}
\]

An intuitive way to set the parameters of the exponential would be to specify that the amplitude should decay from the value 1 to some minimum amplitude over some duration. The decay factor \( f \) can be found from:

\[
f = m^{\frac{1}{t}}
\]

where \( m \) is the desired final amplitude of the exponential, and \( t \) is the desired duration in samples of the exponential.

The following Smalltalk-80 code, which uses variables for the duration and the minimum amplitude

```smalltalk
ICMC 511
```
value could be used to specify the initial parameter values (Figure 3):

```java
maxAmpl factor =

#define_gapped
(7200.0/Samples - 1) reciprocal

self maxAmplValue = 0 xPut: maxAmpl yPut: 0.
self maxAmplValue = 1 xPut: factor yPut: 0.
```

This exponential could be combined with a Product Sound to form an exponential envelope, or used in conjunction with any other Sound in Kyma. A DSPProgram Sound with this code could also be used as the basis for a lightweight exponential signal generator class; instances of the lightweight class would have a graphical editor with fields for the nave, duration, and minimum amplitude [7].

Summary

There are three tools for adding assembly language algorithms to Kyma. The user MicroSound browser is used to enter and debug the assembly language algorithm. The DSPProgram and DSPProgramWithInputs Sounds are used to map user-specified parameters into the parameters needed by the assembly language subroutine. Finally, the lightweight class editor is used to create a new Sound class from the DSPProgram Sound; the new class then has its own icon and visual editor, making it indistinguishable from the built-in Sounds.

Conclusion

There are several advantages to developing code within the framework just described. The framework handles memory allocation, input and output functions, and risk scheduling; the programmer develops code segments and plugs them into the framework to test them. The multi processor hardware provides the computational power to develop and test these segments interactively.

The large set of built-in Sounds within the Kyma language allows the programmer to quickly test a new algorithm on a variety of input signals and in a variety of contexts.

References


