OOPS: An Audio Synthesis Library in C for Embedded (and Other) Applications

Michael Mulshine
Princeton University
310 Woolworth Center
Princeton, NJ 08544
mulshine@princeton.edu

Jeff Snyder
Princeton University
310 Woolworth Center
Princeton, NJ 08544
josnyder@princeton.edu

ABSTRACT

This paper introduces an audio synthesis library written in C with "object oriented" programming principles in mind. We call it OOPS: Object-Oriented Programming for Sound, or, "Oops, it's not quite Object-Oriented Programming in C." The library consists of several UGens (audio components) and a framework to manage these components. The design emphases of the library are efficiency and organizational simplicity, with particular attention to the needs of embedded systems audio development.

Author Keywords

Library, Audio, Synthesis, C, Object-Oriented, Embedded, DSP

ACM Classification

Applied computing Sound and music computing, Computer systems organization Firmware, Software and its engineering Software libraries and repositories

1. INTRODUCTION

In our lab, we've moved toward creating instruments that revolve around embedded audio synthesis. As opposed to developing general purpose controllers for sound synthesis software on multimedia computers, embedded audio synthesis allows us to create instruments that have distinctive and long-lasting identities. We avoid maintenance issues associated with connection to personal computers, such as updates to the operating system and changes in USB data transfer protocols.

One way to achieve embedded audio synthesis is to use embedded Linux computers, such as the Raspberry Pi [13] or the BeagleBoneBlack [1]. While these embedded computers allow designers to develop using C/C++, as well as higher-level music-specific languages like Supercollider [9] and PureData [12], these platforms come with longer startup times, less direct access to low-level peripherals like SPI and I2C, and the weight of an operating system. One example of an ingenious system to address many of these concerns is the Bela[10], which takes advantage of the extensive features of the BeagleBoneBlack, but uses Xenomai RT Linux[1] to make audio and analog/digital input processing the highest priority (even above the operating system). For many researchers, installation artists, and instrument builders, this solution meets their needs.

If the developer's goal is to affordably build attractive sound installations and standalone electronic musical instruments, the aforementioned embedded computers aren't always suitable. They are expensive and, depending on the physical design specifications of an application, somewhat bulky. Embedded Linux computers have comparatively high current demands, and have little control over power consumption. Furthermore, they generally rely on 3rd-party hardware, which may cause issues if the project has commercialization goals.

We opt instead to use 32-bit microcontrollers, such as the STM32F7 [14]. These microcontrollers are small, inexpensive, and low-power. They run "bare-metal," independent of an operating system, using only interrupts to control program flow. 32-bit ARM microcontrollers have begun to approach the processing power that was previously only available through the use of dedicated DSP ICs. They enable very low latency and deterministic run-time control while providing enough overhead for useful audio calculation. While we use a custom-designed board in our lab (a minimal design with nothing but a microcontroller, an audio codec, and a voltage regulator), there are affordable development kits for these microcontrollers that include audio codecs for input and output[2]. An example of a use case where this solution is the best option is a sound art project we are developing that involves hiding tiny audio circuit boards in birdhouses that need to run indefinitely on solar power, while occasionally doing complex audio tasks. Using 32-bit microcontrollers has allowed us to control the current consumption and still have the necessary processing power available when needed.

While developing various embedded audio projects using ARM platforms like the STM32F7, we found ourselves frequently rewriting basic audio synthesis code. We determined that a single repository of portable and efficient code, low-level and general enough to meet the needs of all of our projects, would ease development. We began work on our audio synthesis library, OOPS, in the fall of 2016.

`https://github.com/mulshine/OOPS`

1.1 Why another audio library, and why in C?

C and assembly are the primary languages for embedded development. Most existing libraries, frameworks, and drivers

[1]Xenomai's home page, https://xenomai.org/

[2]ST Microelectronic's STM32f7 Discovery provides I/O audio codecs as well as interfacing compatibility with the Arduino: `http://www.st.com/en/evaluation-tools/32f746gdiscovery.html`.

NIME'17, May 15-19, 2017, Aalborg University Copenhagen, Denmark.

for microcontrollers are written in C. Despite a few strong arguments encouraging the transition to C++ in the embedded realm[3], we set out to write our pseudo-object oriented library in C. On one hand, this was because we already had a sizable C codebase. On the other hand, our projects do not require the higher-level class structures and subclassing capabilities provided by C++, which we were concerned could slow performance. In a recent performance evaluation, we verified our concerns, at least for our specific compiler and embedded environment.See "Case Studies and Performance Evaluation" section for more information.

Moreover, no C/C++ audio synthesis library exists that is both well-suited for microcontroller-specific needs and general enough to be used on many microcontroller platforms. Some of the most popular C++ audio synthesis frameworks, like STK by Perry Cook and Gary Scavone [5] and CSL (CREATE Signal Library) by the researchers at UCSB [11], are tailored towards use on personal computers. They provide features such as debug logging and rely heavily on stdlib and dynamic memory allocation, which are not well-suited for or supported by most embedded platforms. TeensyAudio [15] by Paul Stoffregen is a complete audio library for a microcontroller platform; however, it is designed and optimized for a specific board, the Teensy [16]. The existing C options did not meet our needs for similar reasons (e.g. dependence on memory allocation). Axoloti [18] provides a C repository of code for embedded audio synthesis, but it is designed specifically for STM32f4. We aimed to produce an easily reconfigurable library that would suit the needs of more than one platform.

1.2 What it is

OOPS is a collection of pseudo-classes and functions written in C. It includes audio components that function similarly to traditional UGens, as introduced by the MUSIC-N languages of Max Mathews [8]. OOPS provides various oscillators, filters, envelopes, waveshapers, reverbs, delays, and other basic utilities (midi-to-frequency conversion, clipping, etc.). The DSP implementations are derived from a variety of sources. Many of these algorithms are collected from the Music-DSP mailing list archives and discussions or ported from STK, while some are original. Lookup table optimizations are preferred over computationally intensive routines. However, a near-future aim for OOPS is to provide both accurate (less efficient) and efficient (less accurate) versions of most components in the library. The OOPS audio library is structured for ease and flexibility of use, with the needs of embedded developers in mind.

Included in the OOPS repository are the OOPS library itself, a Python script for the quick creation of lookup tables (wtgenerator.py), and the source for an audio plugin developed using the JUCE framework [17]. The plugin features an easily reconfigurable UI of sliders, buttons, and menus, and provides the basic structural skeleton for writing audio synthesis code using OOPS.

1.3 What it is not

OOPS is not a music-specific programming language, like Supercollider [9] or ChucK [19]. Unlike some of the closest relatives of OOPS (JSyn [3], Common Lisp Music [7], CSound [2], RTCMix [6]), OOPS does not provide any high-level functionality for managing an audio stream or connecting UGens together in a DSP chain. OOPS does not include a scheduler. It is assumed that the user has a system in place with an audio callback that happens per frame and another "process audio" function that runs the per-sample processing. This is reasonable to assume in the embedded environment, and frameworks like JUCE provide for this functionality on more complex computers running standard operating systems. The standard embedded audio application relies on a powerful MCU with one or more fast clocks. A clock is divided down to audio sample rate to generate callbacks for the developer to perform signal processing, sometimes involving audio input from an ADC, and fill an output buffer. The output buffer's data is periodically funneled to a DAC. Every architecture accomplishes this basic configuration differently. It would be very difficult to create a C audio library for embedded development that could manage this particular kind of configuration for every embedded project, and OOPS does not attempt to provide this functionality.

[3]See Saks and Associates' founder Dan Saks' keynote presentation, C++ for embedded C Programmers: `http://www.dansaks.com/talks/ESC-205.pdf`.

2. THE LIBRARY

In the following section, we provide a description of the features and structure of the library.

2.1 Library Structure

The library consists of a set of audio components. A component is a self-named type-defined structure and a set of functions that initialize and act on that structure. The structure definitions live in OOPSCore.h, while the APIs provided for each component live in appropriately named header and source file pairs. For example, the API and implementation for the tCycle (sine oscillator) component lives alongside the tSawtooth, tNoise, and various other oscillator components in OOPSOscillator.h/c.

Every component has an initialization and "tick" function. The initialization function returns an instance of the component. The tick function performs the actual sample-by-sample signal processing or generation. The other functions provide read and/or write access to parameters of the audio component (frequency, resonance, filter coefficients, gain, and more). These functions, along with the tick function, require as their first argument a pointer to an instance of the appropriate component structure.

Oscillators	Filters	Utilities	Other
tPhasor	tOnePole	tRamp	tPluck
tCycle	tTwoPole	tEnvelope	tStifKarp
tSawtooth	tOneZero	tEnvelopeFollower	t808Snare
tTriangle	tTwoZero	tCompressor	t808BDTom
tSquare	tPoleZero	tDelay	t808Cowbell
tNeuron	tBiQuad	tDelayL	t808Hihat
	tSVF	tDelayA	
	tSVFE	tPRCRev	
	tHighpass	tNRev	

2.2 The OOPS Core

OOPS.h is the only file the user should need to directly include in their project to begin using the full feature set of the OOPS library. In it lives a brief API for initializaion of OOPS and preprocessor includes for compilation of the needed OOPS components. At the core of OOPS is a type-defined struct (called OOPS), which indexes memory for the user-provided sample rate, inverse sample rate, and random number generating function pointer. It is required that the user provides their own random number generating function [0.0f, 1.0f) because the rand/randf function in C's standard math library is not supported by every embedded architecture. The OOPS core also contains registries (arrays) for instances of each type of OOPS component. Components refer to the OOPS core for calculations based on the sample rate and random numbers for noise generation. An important feature of the OOPS core registry is its ability to call

component-specific functions on all instances of each component during run-time. The OOPS library takes advantage of this ability to allow run-time changes to the global sample rate.

2.3 Static Allocation and Memory

One challenge embedded audio developers face relates to memory/storage and memory allocation. Most embedded architectures come with hardware-defined memory limitations that are puny compared to the average personal computer. Moreover, many architectures discourage or don't fully support the use of the C standard library memory allocation functions malloc(), calloc(), and free(), so dynamic memory allocation is difficult or impossible. With the OOPS library, we wanted to address these concerns by making static allocation of any number of high level audio components simple and easily reconfigurable. To meet the needs of our own embedded applications, we eschew dynamic allocation of components altogether, opting for compile-time static/automatic allocation. This should be advantageous for most embedded developers.

Any number of user-defined OOPS components are statically allocated as arrays of components in the OOPS core structure (see OOPSCore.h). The authors provide a memory configuration file (OOPSMemConfig.h), which contains a set of preprocessor macros. Each macro is related to a specific component and defines the number of instances of the component to be statically allocated before run-time. If the developer is facing memory restrictions due to hardware, or would like to use more of a particular component in his/her application, s/he may conveniently redefine these macros. If for whatever reason the allocation limit is reached during run-time, new instances of the overloaded component will not be initialized and thus will behave in undefined ways or not work at all.

2.4 Usage

To begin using OOPS, the developer should add the library source to their project and include OOPS.h in a convenient header file. The OOPS core needs to be initialized with a call to OOPSInit(), which takes as arguments the desired system sample rate and a pointer to a random number generating function. Developers using OOPS should create pointers to audio components and assign to them the return value of their associated initialization functions. At that point, they may begin setting parameters of their components and ticking them to process and refill the audio buffer in the main audio callback. Developers should remember to define the appropriate number of instances of each component in OOPSMemConfig.h if they are facing memory restrictions or need more of a certain component. See the OOPS repository README.md for more details on usage.

3. CASE STUDIES AND PERFORMANCE EVALUATION

We have used OOPS in a variety of embedded projects designed by the second author, all with great success: the Drumbox, an electronic drum circuit, which uses an STM32f4 microcontroler; the Genera, a generic testing/experimentation board for audio synthesis or processing, which uses an STM32f7 microcontroller; and the MantaMate, a digital controller to CV Eurorack module, which uses an AVR32UC3a microcontroller. Both the Drumbox and Genera feature audio signal pathways built only with OOPS components. The MantaMate does not perform any audio synthesis but rather relies on handy OOPS utilities, notably several instances of tRamp, a basic ramping tool. In all of these projects, each using different microcontrollers and distinct development environments (Keil uVision 5 or Atmel Studio 7.0), OOPS was easy to integrate and work with. We found ourselves spending much more time making musical decisions about our instruments' sound, usability, and appearance and much less time writing and re-writing basic audio code from scratch.

We have also used OOPS in the software domain to develop unique audio plug-ins using the JUCE framework and XCode. In particular, we have developed a neuron model algorithm synthesis plugin and an 808 percussion plugin, both of which have been or will be used in performance by the Princeton Laptop Orchestra.

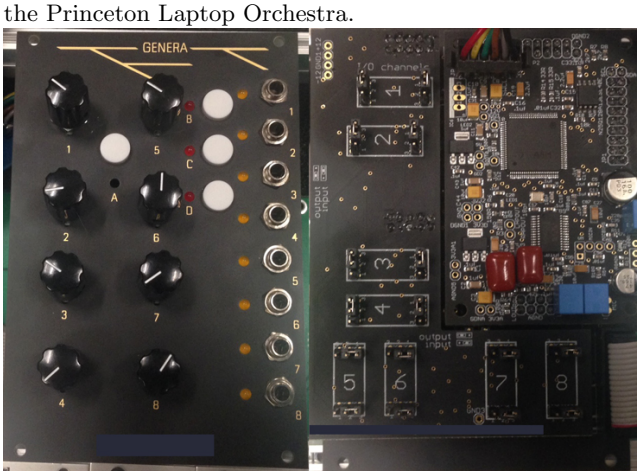

Figure 1: The Genera embedded synthesis engine in Eurorack synthesizer format, front and back view

We have created some test patches to evaluate the efficiency of the library. With a buffer size of 512 at 48K sample rate, we can produce up to 34 OOPS sine waves or 16 bandlimited sawtooth waves at once, each with frequency controlled in real-time via analog input. We have been able to run some interesting waveguide physical models on the hardware, although given the limited CPU speed (216MHz for the STM32f7), we run into performance limitations with more complex models. There is room for improvement, particularly if we add support for the CMSIS DSP library of accelerated commands (which would take advantage of SIMD capabilities, but would be contrary to our platform-independence goal), or if we create fixed-point 16-bit versions of our functions. These are both on our list of planned future work.

Many of the design decisions in the library development were influenced by how the Keil compiler for ARM handled the various possibilities for the code structure. For example, we decided to favor fewer function calls over greater encapsulation after noticing that the compiler tended to produce slower code when more function calls were used. In our most recent tests, we evaluated the performance of basic OOPS components against functionally identical C++, inline-C, and OOPS-alt[4] versions of the components on the Genera board described above. We performed the same tests with two Keil compiler settings: one with -C, -C99 and –CPP flags, and the other with only -C and -C99. The table below presents the performance of OOPS, OOPS-alternative, C++ and inline-C implementations of the same audio task. In each case, the task is to compute and fill a buffer with

[4]An OOPS component with pointers to the functions in its API embedded as members of the component struct. This is closer to object-oriented and C++ programming, but adds a function call.

512 samples of a 220Hz band-limited sawtooth oscillator. The tests were performed by setting a pin high at the start of every buffer frame, setting it low once all 512 samples were computed, and measuring how long that pin stayed high during each frame with a Salaea Logic 8 analyzer. The values presented below are averaged over approximately ten of these measurements.

	CPP (ms)	Not CPP (ms)
OOPS	0.7702	0.7610
OOPS-alt	0.7703	0.7617
Inline-C	0.8744	0.8746
C++	0.8107	n/a

4. CONCLUSIONS AND FUTURE WORK

There are two main areas for improvement that we intend to explore in the future. First is the need for more audio processing functions. There are several common areas of synthesis, such as granular techniques and FFT effects, that are not yet represented in the library. We would also like to include more unusual and experimental sound methods, such as Xenakis's stochastic methods [20] or those described in Nick Collins's "Errant Sound Synthesis" papers [4]. The second area for improvement is in the efficiency of the functions, especially for embedded platforms. Since our primary use cases are ARM Cortex M-series MCUs such as the STM32 series, we intend to build in functionality that takes advantage of the MAC and SIMD capabilities of those processors when they are targeted. Vectorization of data, rather than use of a sample-by-sample tick function, would increase efficiency and take advantage of the available ARM optimizations. We are currently exploring how to incorporate these changes while maintaining the simplicity of the library's API. Our current plan is to create additional functions that operate on arrays of data rather than single-sample inputs. This might necessitate the creation of a UGen connection scheme to organize which operations may be done in parallel, but that would add significant complexity to the library.

Another area of future development will be the creation of lower and higher resolution versions of the objects. In order to take advantage of the single-precision floating-point unit (FPU) in the STM32f MCUs, we chose to use single precision floats throughout (defined as float literals, e.g. 20.0f) and pass data between functions as floats. There are situations in which using doubles would be necessary for acceptable precision, but the STK-style solution of a MY_FLOAT define would create additional casting that could slow the execution time. In the other direction, many of the hardware-specific optimization possibilities of STM32f MCUs can only take advantage of 16-bit fixed point values (such as computing two 16-bit values within a single 32-bit register), so there are efficiency gains to having 16-bit fixed point versions of functions available. When using 16-bit fixed-point computations, sending data between functions as floats becomes a performance bottleneck, since each component must handle the conversion before and after performing its operation. The design goal of a simple, clean library is somewhat at odds with the desire for flexible control of the performance/accuracy continuum. We intend to eventually include multiple resolution versions of each component, but we still need to determine how to handle input-output standards in that case.

There are several projects in our lab that use the OOPS audio synthesis library, and we have so far found it very useful. We believe that this library also could be useful to researchers and designers beyond our own lab, so we hope others will take advantage of this work. We especially believe it will be of interest to designers of custom instruments and installations who, like us, are interested in achieving more complex audio synthesis tasks on embedded processors without the use of an OS. We would be excited to see others contribute to the library and expand the range of UGens it contains.

5. REFERENCES

[1] BeagleBoneBlack. `http://beagleboard.org/black`. Accessed 2017-04-17.

[2] R. Boulanger. *The Csound Book*. M.I.T. Press, Cambridge, Massachusetts, 2000.

[3] P. Burk. Jsyn - a real-time synthesis api for java. In *Proceedings of the International Computer Music Conference (ICMC 2003)*. University of Michigan - Ann Arbor, Michigan, 1998.

[4] N. Collins. Errant sound synthesis. In *Proceedings of the International Computer Music Conference (ICMC 2008)*. Sonic Arts Research Center, Queens University - Belfast, Belfast Ireland, 2008.

[5] P. R. Cook and G. P. Scavone. The synthesis toolkit (stk). In *Proceedings of the International Computer Music Conference (ICMC 1999)*. Beijing, China, 1999.

[6] B. Garton. Rtcmix website. `http://rtcmix.org/`. Accessed 2017-01-24.

[7] F. Lopez-Lezcano and J. Pampin. Common lisp music update report. *International Computer Music Association*, 1999:399–402, 1999.

[8] M. Mathews. *The Technology of Computer Music*. M.I.T. Press, Cambridge, Massachusetts, 1969.

[9] J. McCartney. Supercollider: a new real time synthesis language. In *Proceedings of the International Computer Music Conference (ICMC 1996)*. Hong Kong University of Science and Technology, China, 1996.

[10] A. McPherson. `http://bela.io/`. Accessed 2017-04-17.

[11] S. T. Pope and C. Ramakrishnan. The create signal library (sizzle): Design, issues, and applications. In *Proceedings of the International Computer Music Conference (ICMC 2003)*. Gothenberg, Sweden, 2003.

[12] M. Puckette. Pure data: Another integrated computer music environment. In *Second Intercollege Computer Music Concerts Proceedings*, pages 37–41. Tachikawa, Japan, May 1997.

[13] RaspberryPi. `https://www.raspberrypi.org/`. Accessed 2017-01-24.

[14] STMicroelectronics. Stm32f7 series. `http://www.st.com/en/microcontrollers/stm32f7-series.html?querycriteria=productId=SS1858`. Accessed 2017-01-24.

[15] P. Stoffegren. Teensy audio. `https://www.pjrc.com/teensy/td_libs_Audio.html`. Accessed 2017-01-24.

[16] P. Stoffegren. Teensy board. `https://www.pjrc.com/teensy/`. Accessed 2017-01-24.

[17] J. Storer. Juce. `https://www.juce.com/`. Accessed 2017-01-24.

[18] J. Taelman. Axoloti website. `http://www.axoloti.com/`. Accessed 2017-01-24.

[19] G. Wang. *The ChucK Audio Programming Language: A Strongly-timed and On-the-fly Environ/mentality*. Princeton University, Princeton, New Jersey, 2008.

[20] I. Xenakis. *Formalized Music: Thought and Mathematics in Composition*. Pendragon Press, Hillsdale, New York, 2001.