THE LAB-IN-A-BOX PROJECT: AN ARDUINO COMPATIBLE SIGNALS AND ELECTRONICS TEACHING SYSTEM

William J. Esposito[1], Fernando A. Mujica[1], Domingo G. Garcia[2], Gregory T.A. Kovacs[1]

[1]Department of Electrical Engineering
Stanford University
Stanford, CA 94305

[2]Embedded Processing System Labs
Texas Instruments Incorporated
Dallas, Texas 75243

ABSTRACT

The Stanford "Lab-In-A-Box" project comprises an open source hardware and software tool chain for teaching signal processing and analog electronics. It is intended to improve the teaching of these concepts by providing a platform that is more open and understandable and by lowering the economic barriers to students interested in the field.

To do this, the Lab-In-A-Box brings a full powered Digital Signal Processor (DSP) core to the popular Arduino microcontroller environment and marries it with a simple to use analog front end (AFE). The software platform provided with the Lab-In-A-Box includes an Arduino-like development environment that facilitates learning and quick development of signal processing applications without abstracting away the intricacies of a practical implementation. This system has been used to create several teaching examples and has been tested in courses at Stanford University.

Index Terms— Real-Time DSP, Arduino, Mixed-signal, DSP Education

1. INTRODUCTION

The Stanford "Lab-In-A-Box" project is based on a low cost (less than $100 USD) stack of open-source hardware [1] and software [2] that can act as a breadboard, power supply, analog-to-digital (ADC), digital-to-analog (DAC) signal converters, and signal processing platform. Current educational signals teaching environments tend to rely on expensive, proprietary, or cumbersome software and hardware [3]. The open nature of the Lab-In-A-Box will allow the electronics lab to be accessible outside the walls of the university. This will enable MOOC (Massively Online Open Courseware) laboratory classes in analog electronics and signal processing.

Currently, the application of signal processing concepts is often taught through programming languages like Matlab™ and Python. This approach glosses over the practical pitfalls of embedded implementations and may be less engaging than hands on experiences [4]. Other kits and systems that are designed for remote use by students tend to make compromises in order to use existing widely available low cost hardware [5].

The Lab-In-A-Box stack consists of three boards, which together comprise a lab environment, to be interfaced with any Windows, Mac, or Linux PC.

The first board is an off-the-shelf Arduino UNO, which is based on a 16 MHz ATmega328P, 8-bit microcontroller. This system was selected due to its popularity and global availability, despite its relatively modest performance compared to more modern 32-bit microcontrollers like ARM cortex based systems.

The second board in the stack is the "Analog Shield," a small daughterboard compatible with both the Arduino the DSP Shield (described below). The Analog Shield is designed to provide an improved AFE for both the Arduino and DSP Shield, and includes most of the components needed to enable the teaching of basic analog and mixed signal concepts.

The third board in the stack is the DSP Shield, a fixed point DSP that can be used either in conjunction with the Arduino and Analog Shield or as a stand-alone signal processing environment. This board is the heart of the Lab-In-A-Box, in that it provides drastically increased processing power as compared to an 8-bit microcontroller. The companion development environment called Energia [6] aims to simplify the process of developing code for DSPs.

The Lab-In-A-Box leverages the popularity and familiarity of the Arduino microcontroller board to build an expanded set of teaching tools [5]. The Arduino is a simple microcontroller board with a standardized pin header that allows for quick connection of compatible daughter boards commonly called "shields." The Lab-In-A-Box conforms to this form factor allowing the DSP and Analog Shields to be used either on their own, stacked with an Arduino, or stacked with many other microcontroller platforms that conform to the Arduino Shield footprint standard.

In this paper the hardware and software developed for the system are described. Also highlighted are some experiments created for the platform and some implementation challenges faced in the development process.

2. ANALOG SHIELD

The Analog Shield provides a support structure for the rapid construction and interface of analog circuits and signals with the Arduino, DSP Shield, and compatible microcontrollers.

2.1 Hardware

The Analog Shield contains a fully buffered, four channel, 16-bit ADC (ADS8343) and DAC (DAC8564), each capable of

978-1-4673-9169-6/15/$31.00 ©2015 IEEE

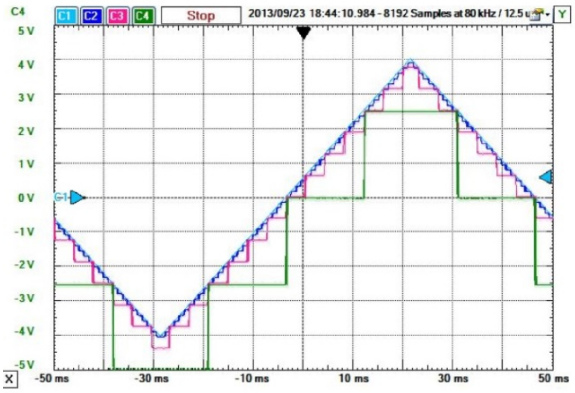

Figure 1: Quantization demonstration on the Analog Shield.

100k samples/second, time multiplexed between the four channels. It also includes a fixed bipolar ±5V power supply, and a variable ±7.5V supply. Finally, it has a prototyping area to attach a 17-row miniature breadboard. This system can be powered entirely from the DSP Shield or Arduino UNO, without the need for additional cables or external power supplies, and interfaces with the host processor via a standard serial peripheral interface (SPI).

2.2 Software

The Analog Shield has been released with a library supporting the Arduino Uno, Mega2560, DSP Shield and ChipKit Uno 32, a PIC based Arduino compatible microcontroller platform [7]. This library provides a single line functional interface with the Analog Shield, with no setup required, providing the student with the ability to read and write a single sample. This approach avoids the pitfalls of hardware setup without abstracting away the sampling process.

2.3 Analog Shield applications

Several experiments have been developed that can be taught with the Analog Shield and an Arduino. The first pair of experiments for the Analog Shield are quantization and sampling. Using the zero order sample and hold functionality of the DAC, the experiment requires students to input a signal and measure the output signal on all four DAC channels, either sequentially or simultaneously. In the quantization experiment, the software uses the libraries to sample the input signal on the analog input and digitally quantize the output to full 16-bit output, 6-bit output, 4-bit output, and 2-bit output. Figure 1 shows the various quantized outputs overlaid on each other. In the sample rate conversion experiment, the software instead only updates the output every 2nd, 4th, 8th, and 16th sample cycle, producing a simultaneous output. This result is, again, similar to the plot shown in Figure 1, and the associated documentation teaches the student about the different causes of the results.

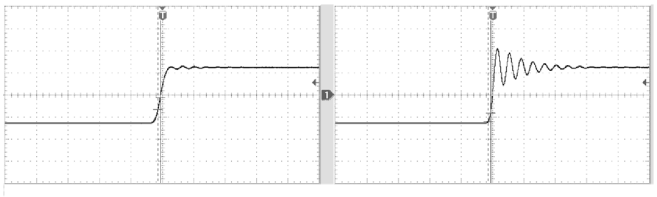

Figure 3: Comparison of square wave (left) with and (right) without software bandlimiting.

Another experiment involves the student building a simple 10 kHz bandwidth Direct Digital Synthesis (DDS) system. The system is implemented entirely in software and outputs samples to the DAC of the Analog Shield. Figure 2 shows a block diagram of the DDS software implementation [8]. The Analog Shield implementation uses a 12-bit long phase-to-amplitude lookup table, populated with 16-bit samples. The phase accumulator and tuning word are each 32-bits long.

As the limited memory of the ATmega328P chip prevents implementations of wavetable synthesis [9], the system uses a real-time phase-to-amplitude converter for waveforms other than a simple sinusoidal output. In the case of square, saw tooth, and triangle waveforms, the DDS algorithm computes the phase to magnitude conversion formulaically as each sample is sent to the DAC. For these non-sinusoidal periodic waveforms, it is beneficial to digitally bandlimit the computed phase-to-amplitude function in order to minimize edge jitter, as well as improve the response of the reconstruction filter when driven with these waveforms, as shown in Figure 3 [10]. In order to achieve this, the step function in the square wave output was replaced with a trapezoidal ramp 128 samples long, approximately 3% of the overall wave period.

Additionally, this example serves as a teaching tool for the utility of a reconstruction filter. The reference design includes a specification for an 8th order active Butterworth filter. The student can probe both the output samples and the reconstructed sine wave after the filter, to demonstrate the critical difference between a system with a well-designed reconstruction filter and one without.

Another experiment captures an input signal at a user selectable sample rate from the ADC on the shield and displays the Fourier transform of the signal on an attached LCD. Using an Electret microphone input buffered from an amplifier and powered by the Analog Shield, the student can visualize ambient sound. Using a PC as the signal source, the student can send the shield a variable frequency sinusoidal input. When the user does not include an anti-aliasing filter, the phenomenon of spectral folding can be seen.

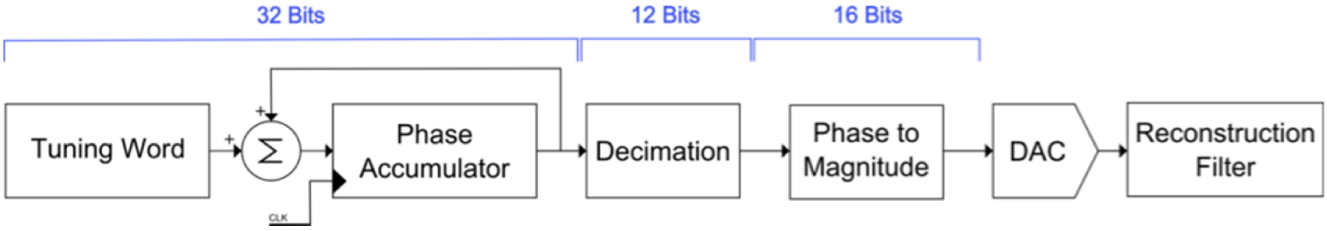

Figure 2: Block diagram of DDS teaching implementation.

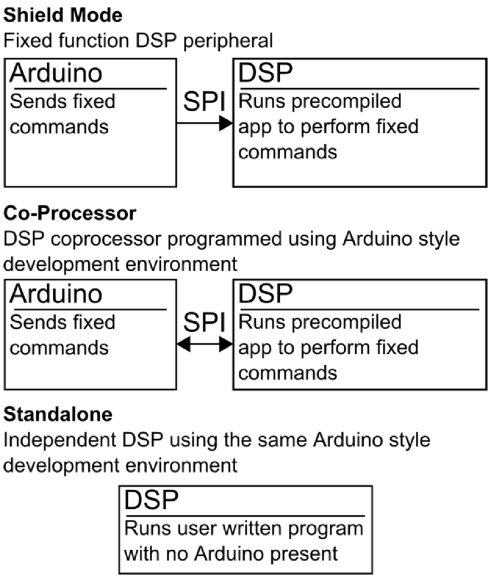

Figure 4: DSP Shield operating modes.

Building on the spectrum analyzer, the analog shield can be used as an FFT network analyzer, by constructing a simple two transistor white Gaussian noise source on the breadboard, using a transistor acting as a diode avalanche noise source [11], as well as a reconstruction filter. The student will average a number of FFTs in real time using the provided code and display them on an LCD. In this way, the student is shown the practical application of spectrally 'white' noise as an analysis tool, and gains an understanding of the basic functionality of modern network analyzer.

With only a relatively low power microcontroller, and a low cost ADC and DAC, important signals and systems concepts can be made much more concrete to a student than would be possible with simulations alone. With open-source libraries and examples, the system has zero software cost and can be extended by the community at large.

3. DSP SHIELD

The DSP Shield is designed to bring an Arduino-like development paradigm to DSP. The platform supports both pre-compiled applications for the DSP Shield, controlled via a communication bus, as well as a fully functional programming environment that enables the DSP Shield to be programmed with syntax and structure familiar to students already versed in the Arduino ecosystem.

3.1 Hardware

The DSP Shield is built around a Texas Instruments TMS320C5535 DSP, a 100 MHz 16-bit fixed point DSP core, with 320kB of onboard RAM [12]. In addition to the DSP core, the board contains a stereo audio codec (AIC3204) with onboard hardware accelerated FIR and IIR filtering blocks. The board also contains a 16x96 pixel monochromatic OLED display and a pair of I^2C GPIO expanders for a 5V tolerant GPIO interface to the Arduino headers, an optional JTAG interface, a micro-SD card, three user controlled LEDs and a bank of user readable DIP switches. The DSP Shield is designed to operate in three modes, as an Arduino Shield daughterboard running a precompiled binary, as a co-processor running user developed

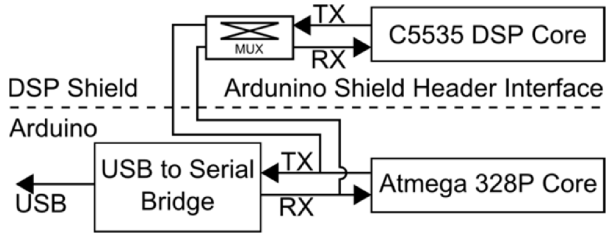

Figure 5: Serial MUX configuration.

code, or as a fully standalone DSP development system. This is illustrated in Figure 4.

3.2 DSP Shield serial communication

To communicate with a host PC and to receive new user code, the DSP Shield uses a serial (UART) communication path with a USB-to-serial bridge. In order to simplify the user interface with the DSP Shield, the system should ideally use only a single USB port on the host PC. Therefore, in order to achieve full functionality, the DSP Shield must be able to share a serial communications path with a stacked Arduino. However, when the DSP Shield is not stacked with an Arduino, it should still function as a standalone device, necessitating a secondary serial communications path. In order to achieve both goals, the DSP Shield uses several software controlled multiplexers to switch between various Serial I/O modes.

In shield or co-processor modes, the DSP Shield is physically stacked on an Arduino UNO and shares receive (RX) and transmit (TX) lines with the output of the Arduino USB-to-serial bridge. This allows messages to be passed back and forth to the host PC through the same serial terminal, although the danger of bus collisions does exist in the event of both the DSP and Arduino attempting to communicate on the bus simultaneously. The serial multiplexers can also be configured to cross-connect the RX and TX lines, allowing the DSP Shield to communicate directly with the Arduino over the serial bus, as depicted in Figure 5. When stacked, the SPI bus between the DSP Shield and Arduino is also connected, and it is typically used for inter-processor communications, keeping the serial port free for use as a debugging console or a user interface.

In standalone mode, the DSP shield's UART is configured to communicate via the shield's onboard USB-to-serial bridge and can be interfaced to a PC using the alternative USB port on the DSP Shield itself.

In the provided Energia programming environment (described below), the user simply needs to select "Standalone" or "Shield" mode from a drop-down menu and the system will build the new user program to configure the multiplexers as needed to either share the serial communications path with a stacked Arduino or to communicate using the DSP Shield's onboard USB to UART bridge.

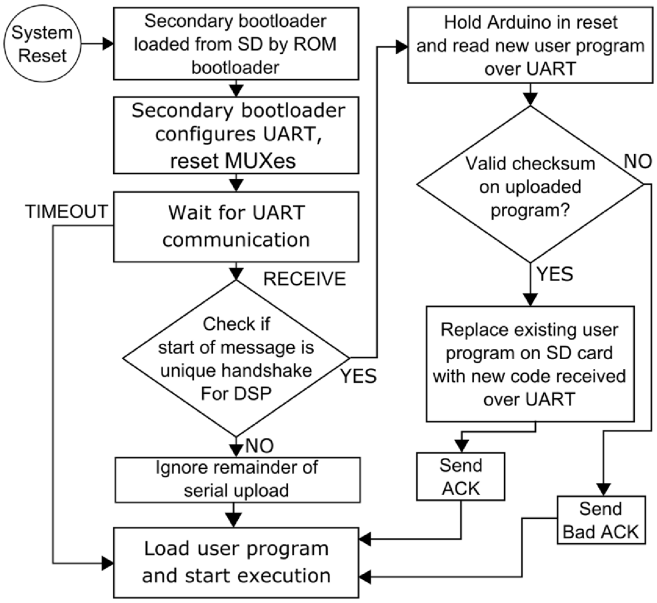

Figure 6: Bootloader flow diagram.

3.3 DSP Shield bootloader

The DSP Shield is designed to make the user uploading experience as straightforward as possible. In lieu of the conventional JTAG based upload to DSP RAM, the system accepts a serial upload of code from the user and stores that user program on an SD card inserted into the DSP Shield. This allows for user program persistence from boot to boot, and use of the device as a standalone embedded system. This design decision guarantees availability of a nonvolatile mass storage device, which enables the design of interesting applications using stored datasets.

In order to achieve this type of transparent upload, the DSP Shield bootloader follows the flow sequence depicted in Figure 6. It first monitors the shared serial bus for a handshake message after restarting. If the first bytes received are a unique handshake for the DSP, it will hold the Arduino in reset and accept a new user upload via the Serial port. Once the upload is complete it validates the code with a checksum and then stores the new program as a binary on the SD card. If the serial upload is not intended for the DSP Shield, or no upload is detected, the system loads the pre-existing binary from the SD card.

3.4 DSP Shield programming environment

A key hurdle in the transition between teaching signal processing theory using a simulated environment like Matlab™ and DSP hardware is usability of the programming environment itself. This problem has been approached in the past, with high-level "block diagram" type solutions like Simulink. These tools automatically generate C code for a DSP and uploads it to a development board [13]. This has the advantage of removing frustrations revolving around implementation details, but simultaneously abstracts away many important real world design challenges like block processing and frame edges that are critical challenges in developing real signal processing applications.

The DSP Shield was designed with a different approach. The software implements the paradigm used in the Arduino programming environment. The Arduino Integrated Development Environment (IDE) is itself based on work done at the MIT Media Lab for the Processing programming environment, which is built as an extension of the Java programming language [14].

The Arduino IDE includes an abstracted C++ Hardware Abstraction Layer (HAL) Application Programming Interface (API), which gives development of applications for the Arduino a similar syntax to desktop applications written in the Processing language. The DSP Shield IDE is based on Energia, a forked version of the Arduino IDE that supports the MSP430 microcontroller and other Texas Instruments processors [6]. The HAL API extends the Energia IDE to provide the DSP Shield with microcontroller like programming interfaces and single click compile-and-upload capability using the serial bootloader system described above.

In addition to Arduino style C++ object oriented interfaces for traditional microcontroller subsystems like GPIO and timers, the DSP Shield HAL API includes a number of interfaces designed to make the teaching of signal processing and development of applications easier without obscuring the underlying structure of the system. Specifically, it includes an interface to the onboard audio codec that provides single-line setup and an interrupt service routine with predefined input and output vector buffers for the user to manipulate.

The API also supports several core signal processing algorithms, designed to integrate easily with the audio codec API. Many of these functions are abstractions or wrappers built on the pre-existing Texas Instruments "DSPLIB" functions which provide core signal processing functionality like trigonometric functions or Fast Fourier Transform (FFT). These include a direct-form FIR filter, which was abstracted for ease of use with the audio codec API and tested with filters of arbitrary length ranging from 3 to 511 coefficients. This structure lends itself perfectly to the implementation of audio effects filters due to their well-behaved overflow characteristics and linear phase. The FIR implementation also lends itself well to the implementation and teaching of adaptive filtering algorithms, like LMS, for teaching applications like noise cancellation.

Another signal processing algorithm implemented for the DSP Shield API was an Infinite Impulse Response (IIR) block-filtering algorithm. The IIR algorithm implements a cascade of direct form I (DF-I) biquad blocks, developed from a previous C6x algorithm, modified to implement "fraction saving" [15]. The transfer function of each biquad (sans fraction saving) is given in equation 1.

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{2^{a_0} + a_1 z^{-1} + a_2 z^{-2}} \quad (1)$$

Even with relatively moderate cutoffs, the effect of quantization is noticeable in this implementation. To improve quality, the algorithm was modified to include fraction saving. Fraction saving leverages the 32-bit summing node in the DF-I implementation. By saving the remaining fractional bits before dividing to quantize the output and then adding the saved fraction back as an input to the next sample, the algorithm approximates a 32-bit signal chain without requiring 32-bit inputs and outputs. The resulting algorithm with fraction saving results in difference equation 2.

```
#include <SPI.h>
#include <mailbox.h>
#include <DSPShield.h>

void setup() {
  //initialize
  DSPShield.init();

  //start shield audio loopback
  DSPShield.startLoopback();

  //load a 201 coeff 3kHz LPF
  DSPShield.setFIRFilter(CHAN_BOTH, LOW_PASS, 201, 3000);
}

void loop() {}
```

Figure 7: Shield App library call.

$$
\begin{aligned}
y_0 &= x_0 b_0 + x_1 b_1 + x_2 b_2 - y_1 a_1 - y_2 a_2 \\
y_0 &= y_0 + frac \\
frac &= y_0 \& ((1 \ll a_0) - 1) \\
y_0 &= y_0 \gg a_0
\end{aligned}
\tag{2}
$$

This IIR implementation proved satisfactory for implementation of a wide range of filter orders and cutoffs, as seen in the algorithm's use in section 3.5. Also included in the IDE is support for PCM WAV playback and recording, using the Audio and SD card components of the HAL API.

3.5 DSP Shield applications

Many users of the Lab-In-A-Box stack are expected to need access DSP functionality without directly developing DSP applications or code. For these users, precompiled applications can be uploaded to the DSP Shield and interfaced to the Arduino (or other compatible microcontroller platforms). To address the needs of most of these users, a high-level general purpose signal processing application was developed. This application is the heart of the "Shield" mode described above, running as a layer on top of "co-processor" mode to provide pre-developed signal processing functionality. This application exposes a number of major signal processing algorithms on the DSP as single-line API calls to a library running on a stacked Arduino, an example of which is seen in Figure 7. Figure 8 shows how this library passes instructions as messages to the DSP Shield via the SPI (Serial Peripheral Interface) connection. The messages are formatted to instruct the precompiled "Shield App" application running on the DSP to configure its operation accordingly.

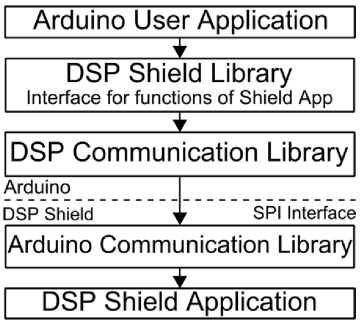

Figure 8: DSP Shield mode application stack.

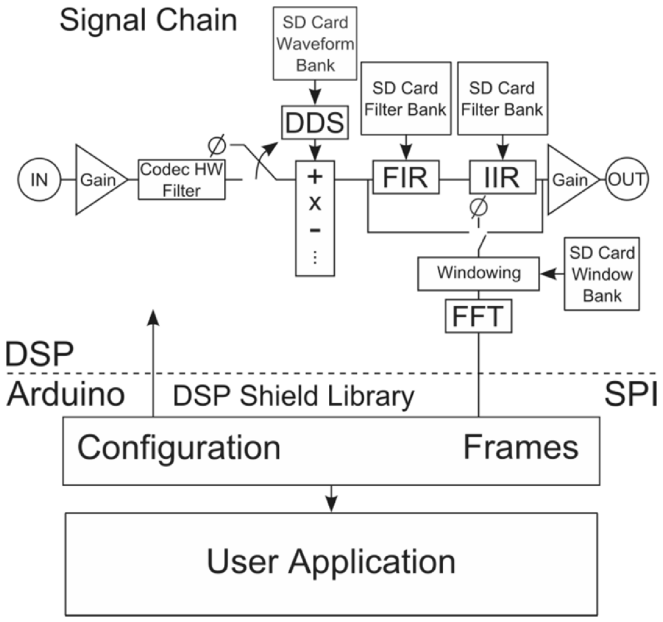

Figure 9: Shield App block diagram.

The shield app supports the following DSP operations: DDS, Chirp, FIR filtering, IIR filtering, Spectrum, and codec control functionality. These are implemented as blocks in a predefined signal chain, seen in Figure 9. Each block is switchable and configurable via messages received over the SPI bus. The Shield App itself is designed to take advantage of the large storage space provided by the presence of an SD card to allow for the storage of tens of thousands of pre-computed digital filters. The DSP filtering application includes a bank of these pre-computed coefficients for high- and low-pass FIR filters in 1 Hz steps from 20 Hz to 20 kHz with order ranging from 4 to 511 coefficients. Filters are loaded on demand from the user application on the Arduino by the Shield App, simply by specifying the cutoff and order desired. Additionally, it accepts manually uploaded filter coefficients from the Arduino, allowing custom filters designed by a student to be used as needed.

The same functionality has been implemented for IIR filtering, using the IIR algorithm described above. Included on the SD card with the Shield App are Butterworth, Bessel, Chebyshev, and Elliptical high- and low-pass filters of 2nd to 14th order in 1 Hz increments from 20 Hz to 20 kHz, although the minimum and maximum cutoffs are dependent on the order and type of filter.

The Shield App can also be configured to periodically return the spectrum of a signal passed through the DSP Shield via SPI. This spectrum can be windowed on the DSP Shield using an arbitrary window vector or one of several predefined vectors stored with the shield app on the SD card.

Finally, the Shield App has a DDS subsystem, which can be used to generate an arbitrary waveform from a single period sample. A selection of such samples are stored on the SD card, allowing for a large number of waveforms to be reproduced, and allowing new waveforms to be added to the system by the user. This is in contrast to the previously described DDS using the Analog Shield alone. Additionally, since the code runs asynchronously, it can be used in conjunction with full functionality filtering and even as an input to

one or both channels of the predefined signal chain. Included in this functionality is a sine sweep register allowing for linear sweep chirps of arbitrary rates. This enables swept sine style network analysis applications to be developed.

Continuing development of the Shield App will bring more functionality to shield mode, and the DSP Shield library is designed to be correspondingly extensible. The source code for the Shield App has been released publicly and was developed in Energia, in order to encourage contribution to the project by the community.

3.6 DSP Shield teaching examples

The DSP Shield development environment ships with examples of all the libraries described above. These examples include basic microcontroller functionality as well as signal processing functionality, including IIR, FIR, and PCM WAV audio recording and playback.

In Stanford's EE122A analog electronics course, a laboratory exercise was developed to compare filtering signals across several domains. Students are provided with pre-built passive, active op-amp, and switched capacitor boards, as well as a DSP Shield. In the course of a few hours, they are asked to configure and test filters across all of these domains. Students then compare and contrast the performance, energy consumption, cost, and flexibility of implementing their signal conditioning in these various filtering domains. Through this hands-on comparison, students come to understand the filter domain tradeoffs made in design scenarios.

In EE264 at Stanford, the DSP shield was the core of a new lab portion of the course which required that students perform exercises on the DSP Shield including flow control, fixed point arithmetic, sample rate conversion, IIR filtering using hardware accelerators, FFT spectral analysis and convolution. These were part of a more theoretical graduate course that previously had not been able to provide hands-on DSP experience to the students [16].

4. CONCLUSION

The ultimate goal for the Lab-In-A-Box stack is to improve the teaching of signal processing and analog electronics both inside and outside the laboratory classroom. To achieve this goal, two boards were developed, along with and accompanying software stack. The first of these boards, the Analog Shield, provides an ADC and DAC. It has been used in several examples, which teach signals concepts in a hands-on setting while remaining portable. The second board—the DSP Shield links a full-featured digital signal processor to the popular Arduino development paradigm and enables a wide range of complexity levels in teaching signals applications, from low level algorithm implementation to highly abstracted single command calls from a host microcontroller.

By providing a low-cost, robust and open toolset that connects the systems that students are using today with high fidelity hardware and software interfaces, the Lab-In-A-Box makes the transition from hobbyist to student easier. It is designed to improve the state of teaching signal-processing and analog electronics lab experiences far from the physical laboratory. As software development continues, it will help enable the kind of massively online open courseware that is reshaping the face of courses worldwide to be leveraged for the laboratory experience.

6. ACKNOWLEDGEMENTS

Development of the Lab-In-A-Box began when, having successfully used the Arduino platform as a teaching tool, Prof. Kovacs met with Martin Izzard, then at Texas Instruments. With his support, Prof. Kovacs began the process of bringing existing embedded analog and digital signal processing tools together with the open source architecture of the Arduino ecosystem. Additional thanks for support for the Lab-In-A-Box, go to Cathy Wicks, Robert Wessels, Laurent Giovangrandi, and Corey McCall.

7. REFERENCES

[1] Open Source Hardware Association, "Definition," [Online]. Available: http://www.oshwa.org/definition/.

[2] Open Source Initiative, "The Open Source Definition," [Online]. Available: http://opensource.org/osd.

[3] R. Krneta, " Blended Learning of DSP Trough the Integration of On-Site and Remote Experiments," *TEM Journal,* vol. 1, no. 3, p. 10, 2012.

[4] "Hands-on, simulated, and remote laboratories: A comparative literature review," *ACM Computing Surveys (CSUR),* vol. 38, no. 3, 2006.

[5] J. Sarik and I. Kymissis, "Lab kits using the Arduino prototyping platform," *Frontiers in Education Conference (FIE),* pp. T3C-1 - T3C-5 , 2010.

[6] R. Wessels, "Energia," [Online]. Available: http://energia.nu/. [Accessed 15 4 2015].

[7] Digilent Inc., "About Us," [Online]. Available: http://chipkit.net/about-us/. [Accessed 26 3 2015].

[8] Analog Devices, "MT-085 Fundamentals of Direct Digital Synthesis (DDS)," [Online]. Available: http://www.analog.com/media/en/training-seminars/tutorials/MT-085.pdf. [Accessed 26 3 2015].

[9] T. &. S. J. Stilson, "Alias-free digital synthesis of classic analog waveforms," *Proc. International Computer Music Conference,* 1996.

[10] P. Symons, in *Digital Waveform Generation*, 2014, p. 216.

[11] R. M. Marston, in *Diode, Transistor & Fet Circuits Manual: Newnes Circuits Manual Series*, Jordan Hill, Oxford , Linacre House, 1991, p. 107.

[12] Texas Instruments, "SPRS737C - TMS320C5535 Datasheet," Dallas, 2011.

[13] Y.-K. C. W. G. a. W.-T. T. W.-S. Gan, "Rapid prototyping system for teaching real-time digital signal processing," *Education, IEEE Transactions,* vol. 43, no. 1, pp. 19-24, 2000.

[14] B. F. Casey Reas, Processing: A Programming Handbook for Visual Designers and Artists, MIT Press, 2014.

[15] R. Y. a. R. Lyons, "DC Blocker Algorithms," *IEEE SIGNAL PROCESSING MAGAZINE ,* vol. March, 2008.

[16] F. A. Mujica, "Teaching Digital Signal Processing With Stanford's Lab In A Box," *submitted to 2015 DSP/SPE Workshop,* 2015.