The Delay Effect

Creating Digital Echo for a Guitar

Kyle Crouse
John Brown University
AR, USA

I. Introduction

The purpose of this project was expand current knowledge and practice application of embedded systems through the design of a delay processor specifically intended for use with a guitar.

Delay is an audio effect described most simply as a continuous, physically realistic echo of the input signal. It is widely used and can be seen in the effects setup of nearly any guitarist who has an effects setup. Delay has been used prominently in music, used commonly to fill out the sound of a guitar, but also on occasion to add rhythmic and percussive aspects to a guitar part. Some well-known examples of this are Where the Streets Have No Name by U2 and Welcome to the Jungle by Guns & Roses.

Considering that a delay processor is intended to echo a signal in a physically realistic way, it would be beneficial to analyze the various aspects of a physical echo. Think of yelling into a canyon. When you yell, there is a period of time you wait before you hear the echo, or put another way, there is a set period of time that you could yell for before you started to hear it echoed back. There is also a certain number of times what you yelled is echoed back, and each consecutive time it is echoed back, it is quieter. From this analysis, it can be surmised that there are at least two parameters to control the output of a delay processor. One parameter would control the period of time you could yell before it is echoed back and the other would control how much volume is taken off of an echo segment each time it is echoed back.

The delay processor designed in this project has three user-defined parameters. The first parameter, mix, can be defined as the balance of the real-time signal with the delay signal. This parameter is not analogous to an aspect of a physical echo, but is nonetheless important. Mix is expressed as a percentage. A high percentage would indicate that the real-time signal will be louder than the delay signal, whereas a low percentage would indicate that the delay signal will be louder than than the real-time signal, and a mix of 50% indicates that the real-time and delay signal are approximately equal in volume. The second parameter, delay time, can be defined as the length of time a delayed segment lasts, expressed in milliseconds. Physically, this parameter can be thought of as the period of time before a delayed segment is echoed back. The third parameter, decay, can be defined as the fraction of volume removed from a delayed segment each time it is echoed back, expressed as the inverse of the fraction plus one.

II. Methods

A. System Design

A high-level block diagram of the delay processor system can be seen in the Fig. 1 below. As shown, the guitar signal is sampled through an analog-to-digital converter (ADC), which sends the sampled signal to the processor. The processor receives as inputs, along with the sample guitar signal, the mix, delay time, and decay parameters. Using these inputs the processor adds the delay to the guitar signal, then sends the guitar signal with delay added to the digital-to-analog converter (DAC), which sends the signal to a speaker.

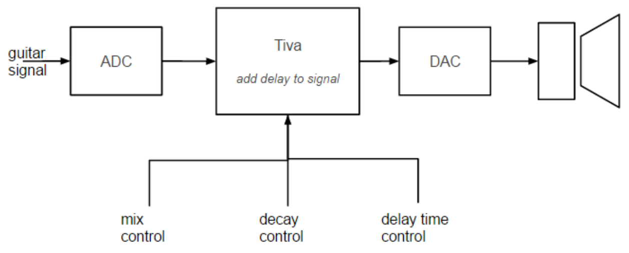

Fig. 1. High-level block diagram

A lower-level block diagram of the system can be seen in Fig. 2. As before, the guitar signal, mix control, delay time control, and decay control are received as inputs. The algorithm works on a sample-by-sample basis. A sample is taken from the guitar signal by the ADC, from the ADC, the sample is sent to the receive buffer, which sends the sample to be saved in a circular array of memory spaces. A number of samples equal to decay - 1 are then read from memory. The samples read are equally spaced in memory. The samples are each multiplied by a factor of 1/decay, to take a fraction of volume of each. The most recent sample would be multiplied by the largest factor, and the least recent sample would be multiplied by the smallest factor. After the samples are multiplied, they are added, to give the complete delay sample. This delay sample and the real-time sample are then multiplied factors defined by the mix control and then added together, to give the real-time signal plus the delay signal, that is, the total output.

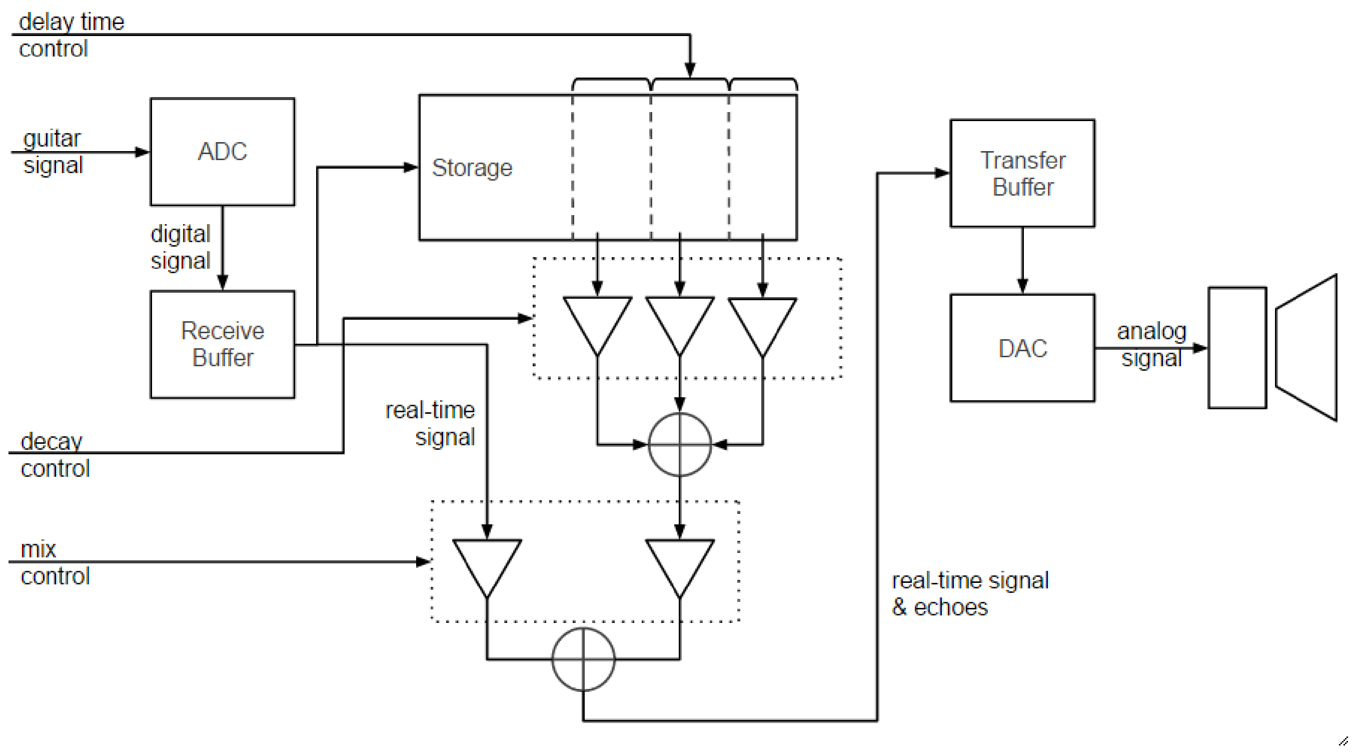

Fig. 2. Low-level block diagram

This result is then sent to the transfer buffer, which sends the sample to the DAC, where the sample is sent to a speaker.

The full circuit schematic can be seen in Fig. 3. The op-amp circuit on the left side acts as a biasing circuit for the guitar signal. The guitar signal was found to have a voltage output range of approximately -100 mV to 1.5 V. Using the information found in Ch. 4 of Op Amps for Everyone by Ron Mancini, a single-supply op-amp circuit with the transfer function $V_{out} = 1.875*Vin + 0.1875$ was designed, to bias the guitar signal into a range of 0 V to 3 V. The output of the op-amp circuit is sent to Analog Input 0 of the Tiva. The Tiva SPI ports are connected to the appropriate ports of the DAC. A reference voltage (REFIN) is used to define the output range of voltage range of the DAC. The output voltage range is 2 * REFIN, where REFIN has a maximum value of 1 V. A voltage divider is used to provide approximately 1 V to the REFIN input. The output of the DAC is sent through a low-pass filter with a cutoff frequency of approximately 9 kHz (Determined by the equation $f_c = 1/2\pi RC$). The output of the low-pass filter is then sent to the speaker.

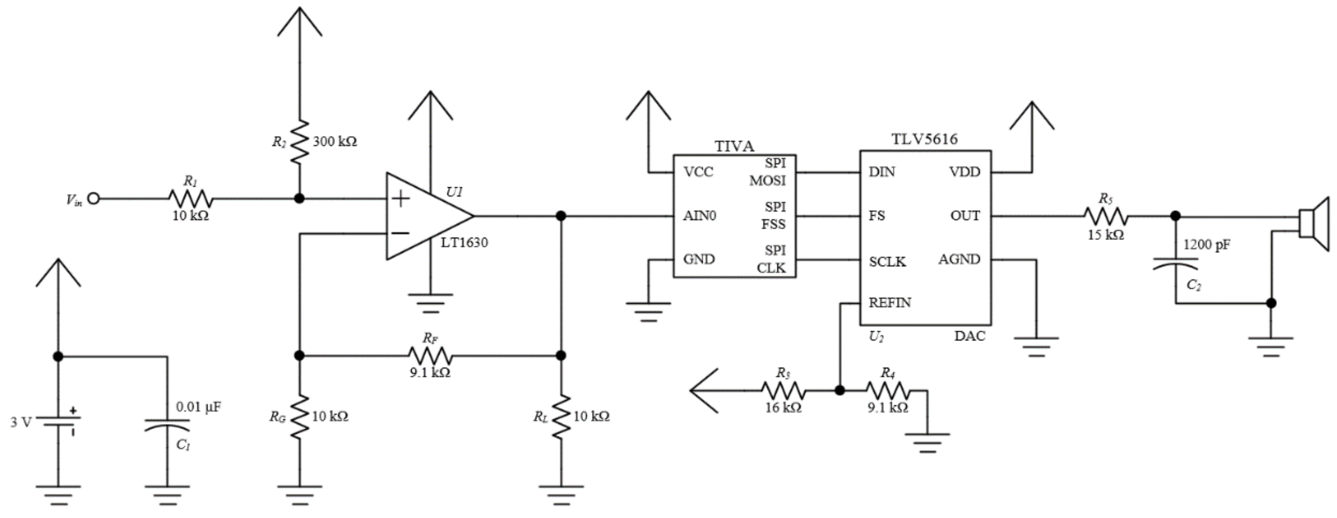

Fig. 3. Circuit Schematic

B. *Code Explanation*

The code for the Delay Processor can be seen in the Appendix. It can be seen that the three user-defined parameters are defined as global variables, along with the input buffer, the array to hold the samples, the array to hold the current address of each echo sample, and a variable used to trigger the rest of the operations after a sample has been taken.

The main function begins by setting the system clock to 80 MHz, the maximum frequency. The ADC is then configured to be timer-triggered, to take a single sample when triggered, and to set an interrupt flag when the sample conversion is complete. The Timer0 module is configured trigger the ADC on time out, and to time out at an approximate frequency set by the SAMPLE_RATE at the beginning of the program. The SSI0 module is configured to use Motorola SPI protocol, with a polarity of 1 and a phase of 1. The bit rate is set at the beginning of the program by DAC_BIT_RATE and has maximum value of 20 MHz as defined by the TLV5616 datasheet. The SSI is also configured to transfer 16 bits at a time.

Following configuration, several crucial local variables are initialized. The block_space variable defines how many elements apart in the data array each delayed sample is by multiplying the sample rate by the delay time. top_mem_address is the highest element in data to be used for the given decay and delay time parameters. It should be noted that not all of the elements of the data array may be used for given parameters. The data array is initialized to the largest array possible to give the most freedom with parameter selection. The InitBlockAddress is function is called, which initializes the g_block_address elements to their proper addresses for the given parameters.

Within the while(1) loop is the event loop. A switch statement dependent upon the g_sample_complete variable is seen. The g_sample_complete was initialized to 0, thus the nothing happens to begin with. However, in the ADC0 interrupt handler function, once the sample has been put into the input buffer variable, the g_sample_complete variable is set to 1, thus sending the program into case 1. Within case 1, the most recent sample (contained in g_input_buffer[0]) is converted from a 32 bit integer to a 16 bit integer, then assigned to the current_sample variable. The current sample is then written to the element of the data array signified by g_block_address[0].

Next, the delay output is calculated using the GetDelayOutput function. This function uses a for loop to multiply the appropriate echo samples (signified by the g_block_address variables) by the factor defined by the DECAY parameter. After the for loop is complete, a switch statement is called dependent upon the DECAY parameter. The switch statement contains various multipliers used to normalize the output to ensure it is not greater than 12 bits.

The mix output is then calculated using the GetMixOutput function. This function simply multiplies the current sample (g_data[g_block_address[0]]) by the mix parameter and divides by 10, then adds the result to the delay_output multiplied by 10 - mix divided by 10. The conceptual result is that the real-time signal is either made louder than the delay signal or vice versa, or they are made equal in volume.

The mix output obtained is then passed by reference into the SendSampleToDAC function, which simply sends the sample by SPI to the DAC and waits until it is completely sent to finish.

Each block_address variable is then incremented and reset to 0 if it reaches the top memory address. Finally, g_sample_complete is set to 0 to wait for the next ADC sample to be taken.

C. *Challenges and Achievements*

The originally-proposed features versus the achieved features are seen in Table I below.

Table I. Feature Achievement

Feature	Achieved?
Guitar Signal Input	Yes
Delay Signal Output	Yes
User-defined parameters of mix, tempo, note, and decay	Almost (mix and decay: yes, tempo and note: combined into delay time)
External controls of mix, tempo, not, and decay	No

As shown in Table I, the goals of this project were in large part achieved. The guitar signal was successfully read into the Tiva microcontroller, and a delay signal was output. The user-defined parameters were successfully implemented for the most part. The mix and decay parameters were implemented, while the tempo and note parameters were combined into the single parameter delay time for simplification. At this point, user-defined parameters are only able to be modified in the code. However, as explained above, each parameter is simply a global variable, thus implementing external controls should not be extremely difficult.

One of the primary challenges faced in this project was lack of memory. The Tiva does not have a large amount of internal memory, especially when dealing with storing audio constantly. The original plan was to use an external SRAM chip. The chip was a 32 pin, 8-bit parallel I/O. After working hard to get everything set to be able to read and write to the chip easily, timing was found to be an issue. Performing the same operations listed in the event loop above with writing to an external memory chip took much longer, so much so that it may not have been possible to complete all the tasks within sample triggers. Finally, it was decided that the best decision was to forego the memory chip and attempt to achieve delay output with just internal memory. Once the memory chip was out of the picture, achieving delay was surprisingly simple. While the memory chip did increase the complexity greatly, the freedom in parameter adjustment it would have provided would have been very desirable. Using the memory chip is not out of the question for future improvements.

III. Results

The project successfully creates a continuous echo on a guitar signal, with user-defined parameters of mix, delay time, and decay.

By experimenting with the parameters, several different impressions can be made with the delay processor. A longer delay can be used to create percussive effects on a guitar signal, while a shorter delay creates more of a reverb effect, making the guitar sound as if it's in a large room or a small room.

Example audio can be heard here:

- https://drive.google.com/file/d/0B2BZi7Fegb_WblIwem5hZnVtNkU/view?usp=sharing (Percussive)
- https://drive.google.com/file/d/0B2BZi7Fegb_WLWxKdTQtWkd1dU0/view?usp=sharing (Reverb)

The Bill of Materials is shown in the Appendix, listing each necessary part to make this project possible.

IV. Conclusions

A delay processor was created using a Tiva C Series microprocessor. The result was the ability to play a guitar plugged into the processor and have the output echoed continuously. The processor has three user-definable parameters: mix, delay time, and decay. The parameters are currently modifiable only in code, but future plans are to add external controls to adjust the parameters.

An immense amount of valuable knowledge was gained through this project. Generally, I learned a great deal about data types, pointers, and the technical aspects of audio, which was extremely interesting to me. More specifically, I learned the importance of diving into projects early. It's easy to be almost intimidated at just starting a project, especially a big one, but once you get into it, you begin to get a better view what there is to be done, and how you need to do it. Furthermore, if there are other things that need your attention, you will be able to plan your time accordingly. Possibly the most valuable lesson was to start small. I began the project by trying to get the delay working with the external memory, when in reality, I could have tested without the external memory first, then added the memory into my already working algorithm. It's easy to be ambitious and try to just shoot for getting a huge project knocked out in one go, but it's almost always a better idea to take it piece by piece, getting the most essential parts working first, then adding in other parts one at a time.

Overall this project was extremely enjoyable for me, and the feeling of the getting the project working was wonderful. I hope to be able to improve upon this project and add to its capabilities in the coming days.

Appendix

Code

```
#include <stdint.h>
#include <stdbool.h>
#include "inc/tm4c123gh6pm.h"
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_gpio.h"
#include "driverlib/adc.h"
#include "driverlib/debug.h"
#include "driverlib/fpu.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/sysctl.h"
#include "driverlib/timer.h"
#include "driverlib/ssi.h"

#define SAMPLE_RATE  13000 // approximately 16kHz (by experimentation)
#define DAC_BIT_RATE    15000000 // DAC baud of 15MHz (TLV5616 max is 20MHz)

//
// Configures Timer0 module by setting Timer0A to time out at a frequency of
// SAMPLE_RATE. Set to trigger ADC0 on time out. Timer enabled upon completion.
// CAVEAT: SAMPLE_RATE frequency is approximate. Tends to be higher than value.
// NOTE: Must be called before ConfigureADC0.
//
void ConfigureTimer0();

//
// Configures ADC0 module to retrieve a single sample triggered by the time out
// of Timer0A. Set to receive input from PE3 (AIN0). Interrupt flag set and
// ADC0IntHandler called after conversion is complete. ADC enabled upon
// completion.
//
void ConfigureADC0();

//
// Configures SSI0 module to master mode, Motorola format, 16-bit transfer.
// Bit rate set by DAC_BIT_RATE. Polarity = 1, phase = 1. SSI enabled upon
// completion.
// NOTE: Only SCLK, FS, and TX pins are enabled. RX pin is not.
//
void ConfigureSSI();

//
// Initializes g_block_address variable by setting 0th element equal to 0 and
// each following component (up to DECAY - 1 components) a distance of
// block_space apart.
//
void InitBlockAddress(uint16_t block_space);

//
// Returns the delayed output for the particular point in time. Gets each sample
// signified by elements of g_block_address, multiplies by appropriate factor,
// and sums. Finally, the sum is multiplied by a factor based on DECAY to scale
// the sum down to 12 bits.
//
uint16_t GetDelayOutput();

//
// Returns real-time output added to delayed output with MIX factored in.
//
uint16_t GetMixOutput(volatile uint16_t delay_output);

//
// Adds control bits to sample pointed to by data, then sends sample to DAC at
// baud rate of DAC_BIT_RATE. Function completed after all bits have been sent.
// NOTE: *data must hold data in 12 LSBs only. DAC may not function correctly
// otherwise.
//
void SendSampleToDAC(volatile uint16_t *data);

//
// Called when ADC0 conversion is completed. Stores most recent sample in
// g_input_buffer[0], clears interrupt flag, and sets g_sample_complete to
// trigger other operations.
//
void ADC0IntHandler(void);

//
// Returns 16-bit int obtained from LSBs of value. 16 MSBs are discarded.
//
uint16_t Convert32to16(uint32_t value);

//
// Increments each value contained in elements of g_block_address array.
// If value exceeds top_mem_address, value is set to 0.
//
void IncBlockAddress(uint16_t top_mem_address);

// The sample rate in kHz. Used to compute block_space.
const uint16_t SAMPLE_RATE_kHz = 16;

// Used to compute the number of echoes as well as the amplitude of echoes.
// The number of echoes is equal to DECAY - 1
const uint8_t DECAY = 3;

// The length of each echo in milliseconds.
const uint16_t DELAY_TIME_ms = 25;

// Balance of real-time signal with delayed signal. A value between 0 and 10.
// Low values mean more of the delayed signal is heard.
const uint8_t MIX = 4;

// Stores sample value retrieved from ADC0.
uint32_t g_input_buffer[1];

// Stores retrieved samples cyclically.
// Not all elements may be used depending upon the parameters.
volatile uint16_t g_data[16000] = { 0 };

// Stores addresses (indices) of current sample and delayed samples.
volatile uint16_t g_block_address[6];
```

```
// Set by ADC0IntHandler. Triggers other operations to take place in main.
volatile uint8_t g_sample_complete = 0;



void main(void)
{
   // 80MHz Clock
   SysCtlClockSet(SYSCTL_SYSDIV_2_5 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
                        SYSCTL_XTAL_16MHZ);
   ConfigureADC0();
   ConfigureTimer0();
   ConfigureSSI();

   uint32_t block_space = SAMPLE_RATE_kHz * DELAY_TIME_ms;
   uint16_t top_mem_address = DECAY * block_space - 1;
   uint16_t current_sample = 0;
   volatile uint16_t delay_output, mix_output;

    InitBlockAddress(block_space);
   IntMasterEnable(); // Enable processor interrupts

    while(1)
    {
      switch(g_sample_complete)
      {
         case 1:
         {
            // convert most recent sample to 16 bit int
            current_sample = Convert32to16(g_input_buffer[0]);

            // write most recent sample to current mem address
            g_data[g_block_address[0]] = current_sample;

            // calculate delay output
            delay_output = GetDelayOutput();

            // calculate delay output mixed with real-time output
            mix_output = GetMixOutput(delay_output);

            // send to DAC
            SendSampleToDAC(&mix_output);

            // increment addresses
            IncBlockAddress(top_mem_address);

            g_sample_complete = 0;
            break;
         }
         default:
            break;
      }
    }
}

void ConfigureTimer0()
```

```
{
   SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
   TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);

    // Value loaded in causes time out at a frequency of SAMPLE_RATE
    TimerLoadSet(TIMER0_BASE, TIMER_A, SysCtlClockGet()/SAMPLE_RATE);
   TimerEnable(TIMER0_BASE, TIMER_A);
    TimerControlTrigger(TIMER0_BASE, TIMER_A, true);    // Trigger ADC0
    TimerEnable(TIMER0_BASE, TIMER_A);
}

void ConfigureADC0()
{
   SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
   SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
   SysCtlDelay(10);                               // Allow clock to reach ADC0
    GPIOPinTypeADC(GPIO_PORTE_BASE, GPIO_PIN_3);   // PE3 -> AIN0
    IntDisable(INT_ADC0SS3);                       // Disabled for configuration
    ADCIntDisable(ADC0_BASE, 3);
    ADCSequenceDisable(ADC0_BASE, 3);             // Disabled for configuration
    ADCSequenceConfigure(ADC0_BASE, 3, ADC_TRIGGER_TIMER, 0);

    // Single sample, AIN0, interrupt flag set upon completion
    ADCSequenceStepConfigure(ADC0_BASE, 3, 0, ADC_CTL_CH0 |
         ADC_CTL_IE | ADC_CTL_END);
    ADCSequenceEnable(ADC0_BASE, 3);
    ADCIntClear(ADC0_BASE, 3);
   ADCIntEnable(ADC0_BASE, 3);
   IntEnable(INT_ADC0SS3);
}

void ConfigureSSI()
{
   SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
   SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI0);

    // PA2 -> CLK, PA3 -> FS, PA5 -> TX
   GPIOPinTypeSSI(GPIO_PORTA_BASE, GPIO_PIN_2 | GPIO_PIN_3 | GPIO_PIN_5);

    // Polarity = 1, Phase = 1, Master mode, Baud rate = DAC_BIT_RATE,
    // 16-bit transfers
   SSIConfigSetExpClk(SSI0_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE_3,
         SSI_MODE_MASTER, DAC_BIT_RATE, 16);
   SSIEnable(SSI0_BASE);
}

void InitBlockAddress(uint16_t block_space)
{
   uint8_t n;

   g_block_address[0] = 0; // real-time signal

   for(n = 1; n < DECAY; n++)
      g_block_address[n] = block_space * n - 1;
}


uint16_t GetDelayOutput()
```

```
{
   uint16_t output = 0;
   uint8_t n;

    // Each delayed sample is multiplied by a factor then added to the total.
    // More recent samples receive a larger multiplying factor than older
    // samples.
   for(n = 1; n < DECAY; n++)
      output = output + ((g_data[g_block_address[n]] * n) / DECAY);

    // Normalize output
   switch(DECAY)
   {
   case 4:
      output *= 0.6;
      break;
   case 5:
      output *= 0.5;
      break;
   case 6:
      output *= 0.4;
      break;
   case 7:
      output *= 0.3;
      break;
   case 8:
      output *= 0.28;
      break;
   case 9:
      output *= 0.25;
      break;
   case 10:
      output *= 0.22;
      break;
   default:
      break;
   }

   return output;
}

uint16_t GetMixOutput(volatile uint16_t delay_output)
{
   uint16_t output;

    // Real-time signal and delay_output weighted based on MIX.
   output = ((MIX * g_data[g_block_address[0]]) / 10) +
        (((10 - MIX) * delay_output) / 10);

   return output;
}


void SendSampleToDAC(volatile uint16_t *data)
{
   uint32_t send_data;

   // Sets control bits for DAC & converts to 32-bit int.
   // 0x4000 for fast mode
   // 0x0000 for slow mode
   send_data = (*data | 0x4000) & 0xFFFF;

   SSIDataPut(SSI0_BASE, send_data);   // Transmit send_data over SSI.
   while(SSIBusy(SSI0_BASE));         // Wait until SSI is done transmitting.
}

void ADC0IntHandler(void)
{
    ADCIntClear(ADC0_BASE, 3);   // Clear the interrupt status flag.

    // Place sample value into g_input_buffer.
    ADCSequenceDataGet(ADC0_BASE, 3, g_input_buffer);
    g_sample_complete = 1;          // Trigger other operations to begin.
}

uint16_t Convert32to16(uint32_t value)
{
   uint16_t converted;
   converted = value & 0xFFFF;
   return converted;
}

void IncBlockAddress(uint16_t top_mem_address)
{
   uint8_t n;

   for(n = 0; n < DECAY; n++)
   {
      g_block_address[n]++;

      if(g_block_address[n] > top_mem_address) // Cyclic
         g_block_address[n] = 0;
   }
}
```

Bill of Materials

QTY	DESCRIPTION	MANUFACTURER	MAN P/N	VENDOR	VENDOR P/N	PRICE/UNIT	PRICE
1	EVAL KIT TM4C123GXL LAUNCHPAD	TEXAS INSTRUMENTS	EK-TM4C123GXL	DIGI-KEY	296-35760-ND	$13.49	$13.49
2	CONN JACK PHONE 1/4" 2POS OPEN	SWITCHCRAFT INC.	11	DIGI-KEY	SC1085-ND	$3.05	$6.10
1	HOLDER BATT 2-AA CELLS WIRE LDS	MPD	BC22AAW	DIGI-KEY	BC22AAW-ND	$0.92	$0.92
1	BREADBOARD SOLDERLESS 600TIE	BUD INDUSTRIES	BB-32620	DIGI-KEY	377-2091-ND	$8.40	$8.40
1	RioRand 3 x 40P 20cm Dupont Wire Jumpers	RioRand	RRBBJW75PCSP	Amazon	-	$8.00	$8.00
3	RES 10K OHM 1/4W 5% CARBON FILM	STACKPOLE ELECTRONICS INC	CF14JT10K0	DIGI-KEY	CF14JT10K0CT-ND	0.1	$0.30
1	CAP CER 10000PF 50V 10% AXIAL	VISHAY BC COMPONENTS	A103K15X7RF5TAA	DIGI-KEY	1103PHCT-ND	0.32	$0.32
1	RES 9.1K OHM 1/4W 5% CARBON FILM	STACKPOLE ELECTRONICS INC	CF14JT9K10	DIGI-KEY	CF14JT9K10CT-ND	0.1	$0.10
1	RES 300K OHM 1W 5% AXIAL	VISHAY BC COMPONENTS	PR01000103003JR500	DIGI-KEY	PPC300KW-1CT-ND	0.37	$0.37
1	RES 16K OHM 1W 5% AXIAL	VISHAY BC COMPONENTS	PR01000101602JR500	DIGI-KEY	PPC16KW-1CT-ND	0.37	$0.37
1	RES 15K OHM 1/2W 1% AXIAL	VISHAY BC COMPONENTS	SFR16S0001502FR500	DIGI-KEY	PPC15.0KXCT-ND	0.31	$0.31
1	CAP CER 1200PF 50V 5% RADIAL	VISHAY BC COMPONENTS	K122J15C0GF5TL2	DIGI-KEY	BC1026CT-ND	0.47	$0.47
1	IC OPAMP GP 30MHZ RRO 8DIP	LINEAR TECHNOLOGY	LT1630CN8#PBF	DIGI-KEY	LT1630CN8#PBF-ND	$6.53	$6.53