Simple Trainable Music Beats Generator - Copatone: SPA-32

Chris Muldoon
Justin Opraseuth
Mohammed Alam

EE4144

Date Due: 12/14/14

Abstract

The Copatone: SPA-32 is a simple trainable beat generator. In the initial startup state, the user can shake or tap the development board in rhythm to record a simple 4 measure beat. The device then plays back the rhythm, looping back to the start when it reaches the end. A low tone to simulate a bass drum is played back on the downbeat, while a high tone is played on the upbeat.

Introduction

Various parts of the STM32F4 development board were used to implement the program. The LIS3DSH accelerometer was utilized to generate the volume level for the waveform. The CS43L22 audio DAC was utilized to produce output for headphones.

In order to generate the magnitude of the headphone output, the combined accelerometer output was used. The three readings from the different axes were combined by taking the vector sum. The equation used for this was:

$$output = \sqrt{a_x^2 + a_y^2 + a_z^2}$$

By implementing the accelerometer data in this way, one only needed to shake the STM32F4 board to make a beat pattern.

The CS43L22 was interfaced to the microprocessor using the Integrated Interchip Sound (I2S) and Inter-Integrated Circuit (I2C) interfaces. Before the the protocols can be used, however, the general-purpose I/O GPIO pins must be initialized. Also, different clocks must be set.

The different tones generated by the board are created from a simple function of the <arm_math.h> library. The sine function is a periodic function that uses the current time to continuously send waveform values to the audio buffer.

The send_to_speaker function takes 16-bit audio samples and sends it to the audio output on the discovery board. One can think of these audio samples as single points on a sound wave. The more frequenly that samples are taken, the closer that the output sounds compared to the original.

Description and Implementation

Record State

The device starts in the record state after an initial two second delay. In the record state, the device can be shaken or tapped to develop a rhythm. The initial accelerometer readings at a steady state is used to normalize the readings in order to record changes from the initial position. This is done by taking the magnitude of the of the X-Y-Z axis readings on the accelerometer at the start of the program. The code for recording the initial magnitude of the accelerometer readings:

```
// magnitude of accelerometer readings
// function take the accelerometer axis data readings pointer
int get_magnitude( TM_LIS302DL_LIS3DSH_t *accel_data ) {

      return sqrt( accel_data->X*accel_data->X +
                   accel_data->Y*accel_data->Y +
                   accel_data->Z*accel_data->Z );
}
```

The subsequent readings are taken by subtracting the initial magnitude from the current magnitude at each interval period.

Each sampling period is 250 milliseconds. A sample is taken every 25 millisecond for 100 milliseconds for a total of four samples followed by a 150 millisecond delay. The average of the four samples is stored in an array of length 32 for a total of 8 seconds of sample data. The 100ms recording period followed by a 150ms delay period was used in order to allow the average magnitudes to vary in value. The following code implements the recording of the sample data:

```
// four samples have been collected
if (num_samples%4 == 0) {

        // get average of four samples and store in array
        sound_data[ num_samples/4 - 1] = accel_magnitude_sum/4;
        accel_magnitude_sum = 0; //reset sum to 0

        //wait for remaining 150ms of 250ms cycle
        while (1) {
                if ((msTicks - msPrev) >= 150) {
                        msPrev = msTicks;
                        break;
                }
        }
```

The array of recorded samples is adjusted in order to properly amplify the up or down beat and to set any low values to zero. All values below 50 are set to 0 and any value above gets amplified by 2.8x.

Playback State

Using the 32 value array generated in the record state, the device is able to playback the beat in a looping fashion. The function playArray takes a pointer to an array which holds integer values that represent the volume of the note to be played. Once entering the playArray function, the program will remain in the playback state until it is terminated. To accomplish this a while(1) loop is used as seen below:

```
            while(1){
                  if( lastPlay + 250 <= msTicks){
                        lastPlay = msTicks;
                        playSample(arr[count]);
                        count++;
                        if (count >=size) count = 0;  //if at end of arr,
      loopback

                  }//endif
                  else {send_to_speaker(0);}

            } //end while
```

Since our project specification calls for a tempo of 120 beats per minute, and quantization to an eighth note (one half of a beat), a note is played back once every 250 milliseconds. This is implemented with the lastPlay variable and the if statement seen above. The playSample() function is called and the counter is incremented once every 250 milliseconds. While waiting for the 250 milliseconds to pass after playing a sample, a sample of ‘0’ is sent to the speaker, in order to prevent the user from hearing white noise.

The playSample() function plays a short sample with a volume specified by its parameter. The frequency of the sample changes based on if it is on the downbeat or the upbeat (the notes in between beats). Since every other sample will be a downbeat, with the others upbeats, this can easily be implemented with a static variable for the frequency and a simple if else statement to change the frequency when needed. This functionality can be seen in the pseudo code snippet below:

```
void playSample(int volume){
      static int frequency = 500;

      //code to playback a note goes here

      if(frequency == 500) frequency = 3000;
      else frequency = 500;
}
```

The if else statement sets the frequency for the next function call: to 3000 if the frequency that was just played was 500, or to 500 of a note with a frequency of 3000 was just played. The frequency is set to 500 initially because we want to play a low tone the first time the function is called, because it will be a down beat.

To actually playback a sound, individual 16-bit samples are sent to the speaker using the send_to_speaker function. In this project, samples are generated in a sinusoidal pattern, so that the user hears a close approximation to a sine wave. The sample is calculated by multiplying the volume parameter by the return value of the sine function that takes in the frequency times the time as an argument. These samples are continuously generated and sent to the speaker of the 150 millisecond interval. A while loop is used to implement this continuous generation of samples.

Future Features to Implement

In the future, the Copatone: SPA-256 beat generator will be able to support the playback of prerecorded audio samples. The team hoped to have had this functionality implemented before the project delivery date but was unable to complete in time. The audio sample would have been pre-processed using MATLAB's wavread function. The samples returned by the function would be processed by a simple python script that would format the data into a form that can be pasted into a C file and compiled as an array. The array of samples was then to be played back in place of the samples generated by the sine wave.

Unfortunately, it is the playback of samples that held up the production of this feature. The audio sample used was compressed in audio editing software to a sample rate of 8000Hz. This meant that a sample must be played back once every 0.125 milliseconds or once every 125 microseconds. This time interval is smaller than what could represented by the msTicks variable, which as a resolution of only 1 millisecond. So in order to control the playback of samples, the team experimented with adjusting the

Systick interrupt to interrupt once per microsecond. Unfortunately this change was unable to fix the issue of white noise that is played back instead of the audio sample. Further experimentation must be done to progress further with the implementation of this feature.

Conclusion

The Copatone: SPA-32 beat generator is a small but capable board. When programming the device to create a beat generator, operation was less than ideal. This is due to the small amount of onboard RAM (192 KB). This is not enough to store a quality audio sample. As such, the beat generator utilized the sine function to create a basic tone. The Discovery board is capable of supporting external RAM, which could be used for storing samples in the next version, SPA-256.

Appendix

accel.c

```
#include <math.h>
#include <stdlib.h>

#include "accel.h"

//used for getting initial magnitude
int get_magnitude( TM_LIS302DL_LIS3DSH_t *accel_data ) {
	return sqrt( accel_data->X*accel_data->X + accel_data->Y*accel_data->Y + 
accel_data->Z*accel_data->Z );
}

//subtracts initial magnitude to obtain how fast sensor is moving
int get_computed_magnitude( TM_LIS302DL_LIS3DSH_t *accel_data, int orig_magnitude ) {
	return abs( sqrt( accel_data->X*accel_data->X + accel_data->Y*accel_data->Y + 
accel_data->Z*accel_data->Z ) - orig_magnitude );
}

//convert values for use by audio engine

void convert_array_for_audio( int *sound_data, int array_len ) {
	for (int i=0; i<array_len; ++i) {
		if ( (sound_data[i] <= 50) ) {
			sound_data[i] = 0;
		}
		else {
			sound_data[i] = (int)(sound_data[i]*2.8f);
		}
	}
}
```

playback.c

```
#include <stdio.h>
#include <stdlib.h>
#define ARM_MATH_CM4
#include <arm_math.h>
#include <math.h>

#include "stm32f4xx.h"
#include "speaker.h"

extern volatile uint32_t msTicks;



void playSample(int volume){
	static int frequency = 500;
	uint32_t initialT = msTicks;
	int16_t audio_sample;
	printf("play %i, at %i\n", volume, msTicks);

	//STATIC LED FLAG = 0

	if(frequency ==3000){
		while (initialT + 88 > msTicks){
			send_to_speaker(volume*(-.3f));
			float t = msTicks / 1000.0; // calculate time -- SysTick is configured
for 0.1 ms
			audio_sample = (int16_t) 30 + (.8f*volume*(.3f +
arm_sin_f32(frequency*t))); // calculate one sample for the speaker
			// the CMSIS arm_sin_f32 function from arm_math.h is typically faster
than sin() from math.h
			send_to_speaker(audio_sample);	// send one audio sample to the
audio output
		}
	}

	else{
		while (initialT + 150 > msTicks){
			float t = msTicks / 1000.0; // calculate time -- SysTick is configured
for 0.1 ms
			audio_sample = (int16_t) (volume*arm_sin_f32(frequency*t)); // calculate
one sample for the speaker
			// the CMSIS arm_sin_f32 function from arm_math.h is typically faster
than sin() from math.h
			send_to_speaker(audio_sample);	// send one audio sample to the
audio output
		}
	}

	if(frequency == 500) frequency = 3000;
	else frequency = 500;
}


void playArray(int *arr, int size){
  uint32_t lastPlay = msTicks;
	int count = 0;
	while(1){
		if( lastPlay + 250 <= msTicks){
			lastPlay = msTicks;
			printf("calling %i", count);
			playSample(arr[count]);
			count++;
```

```
                        if (count >=size) count = 0;  //if at end of arr, loopback

                }//endif
                else {send_to_speaker(0);}


        } //end while

}
```

main.c

```c
#include <stdio.h>
#include <stdlib.h>

#include "stm32f4xx.h"
#include "tm_stm32f4_lis302dl_lis3dsh.h"

#include "accel.h"
#include "speaker.h"
#include "playback.h"



volatile uint32_t msTicks = 0;                      /* counts 1ms timeTicks       */

// SysTick Handler (Interrupt Service Routine for the System Tick interrupt)
void SysTick_Handler(void){
  msTicks++;
}

// initialize the system tick
void InitSystick(void){
	SystemCoreClockUpdate();                      /* Get Core Clock Frequency   */
  if (SysTick_Config(SystemCoreClock / 1000)) { /* SysTick 1 msec interrupts  */
    while (1);                                  /* Capture error              */
  }
}

int32_t main( void ) {
	printf("#  1\n");
//	InitLEDPins();  //LED off intitally
	uint32_t msPrev = 0;
	TM_LIS302DL_LIS3DSH_t init_axes_data;
	TM_LIS302DL_LIS3DSH_t axes_data;
	int orig_magnitude;
	int magnitude;

	int sound_data[32];   //holds value used every 250ms for a total of 8 seconds of beats
	int accel_magnitude_sum = 0; //current sum used in calculating average
	int num_samples = 0;

	SystemInit();
	InitSystick();

	/* Initialize LIS302DL */
  TM_LIS302DL_LIS3DSH_Init(TM_LIS3DSH_Sensitivity_2G, TM_LIS3DSH_Filter_50Hz);
	TM_LIS302DL_LIS3DSH_ReadAxes(&init_axes_data);
	orig_magnitude = get_magnitude( &init_axes_data );


	//wait 2 seconds before getting beat data
	int countin = 1;
	uint32_t countintimer ;

	while (1) {
		if (msTicks >= 500 && countin ==1 ) {printf("#  2\n"); countin++;}
		if (msTicks >= 100 && countin ==2 ) {printf("#  3\n"); countin++;}
		if (msTicks >= 1500 && countin ==3 ) {printf("#  4\n"); countin++;}
		if (msTicks >= 2000) {
			break;
		}
	}
	//main loop for loading beat array
	while (1) {

		if ((msTicks - msPrev) >= 25) {
			//IF (NUM SAMPLES % 8 == 0) SET LED ON
		if (num_samples%8 ==0) printf("\n## [ %i ] ##,  ", (num_samples%32/8 +1));
			TM_LIS302DL_LIS3DSH_ReadAxes(&axes_data);
			magnitude = get_computed_magnitude( &axes_data, orig_magnitude );

			accel_magnitude_sum += magnitude;
			++num_samples;
			msPrev = msTicks;

			//after any multiple of 4 readings have been taken
			if (num_samples%4 == 0) {
				//TURN LED OFF
				sound_data[ num_samples/4 - 1] = accel_magnitude_sum/4;   //set
current beat to average sensor data
				accel_magnitude_sum = 0;     //reset sum to 0

				//wait for remaining 150ms of 250ms cycle
				while (1) {
					if ((msTicks - msPrev) >= 150) {
						msPrev = msTicks;
						break;
					}
				}

				printf( " %d", sound_data[ num_samples/4 - 1] );
			}
		}

		//if 128 samples gathered, 8 seconds of data is recorded, end loop
		if ( num_samples >= 128 ) {
			break;
		}
	}

	printf( "\n\n" );
	convert_array_for_audio( sound_data, 32 );
	for (int i=0; i<32; ++i) {
		printf( "%d\n", sound_data[i] );
	}

	init_speaker();
	playArray( sound_data, 32 );

}
```