```
================================================================================
Name : Arpit Savarkar
================================================================================
================================================================================
Getting in Tune.c
================================================================================
/*
 * Copyright 2016-2020 NXP
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without modification,
 * are permitted provided that the following conditions are met:
 *
 * o Redistributions of source code must retain the above copyright notice, this
list
 *   of conditions and the following disclaimer.
 *
 * o Redistributions in binary form must reproduce the above copyright notice, this
 *   list of conditions and the following disclaimer in the documentation and/or
 *   other materials provided with the distribution.
 *
 * o Neither the name of NXP Semiconductor, Inc. nor the names of its
 *   contributors may be used to endorse or promote products derived from this
 *   software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR
 * ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
 * (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
 * ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

/**
 * @file    Getting In Tune.c
 * @brief   Application entry point.
 */
#include <stdio.h>
#include "board.h"
#include "peripherals.h"
#include "pin_mux.h"
#include "clock_config.h"
#include "MKL25Z4.h"
#include "fsl_debug_console.h"

#include "fp_sine.h"
#include "ADC.h"
#include "DAC.h"
#include "autocorrelate.h"
#include "util.h"
#include "stdint.h"
#include "test_sin.h"

#define ADC_FREQ 96000
#define SIZE 1024
```

```
/*
 * @brief   Application entry point.
 */
int main(void) {

      /* Init board hardware. */
    BOARD_InitBootPins();
    BOARD_InitBootClocks();
    BOARD_InitBootPeripherals();
#ifndef BOARD_INIT_DEBUG_CONSOLE_PERIPHERAL
    /* Init FSL debug console. */
    BOARD_InitDebugConsole();
#endif

    int samples, i=0;
    uint16_t output[SIZE], input[SIZE];
    int frequencies[] = {440, 587, 659, 880};
    DAC_Init_();
    DMA_Init_();
    TPM0_Init_();
    TPM1_Init_(ADC_FREQ);
    ADC_Init_();

//    test_sine();

    for(i = 0; i<=4; i++) {
      if(i==4) {
                  i=0;
            }
      PRINTF("Frequency: %d Hz\r\n", frequencies[i]);
      samples = tone_to_samples(frequencies[i], output, 1024);
      DAC_begin(output, samples);
      ADC_Buffer(input, 1024);
      analysis(input, 1024);

    }
    return 0 ;
}
```

==========

ADC.h

==========

```
/*
 * ADC.h
 *
 *  Created on: Nov 14, 2020
 *      Author: root
 */

#ifndef ADC_H_
#define ADC_H_

#define SYS_CLOCK       (48e6)

#define BEGIN_TPM1 TPM1->SC |= TPM_SC_CMOD(1);
#define STOP_TPM1  TPM1->SC &= ~TPM_SC_CMOD_MASK;

/*
 * Initializing the TPM1 Timer
 *
```

```
 * Parameters:
 *   sample: int Initializes the TPM Clock to the requested rate
 *
 * Returns:
 *   void
 */
void TPM1_Init_(int sample);

/*
 * ADC Initializer function to appropriate characteristics
 *
 * Parameters:
 *   void
 *
 * Returns:
 *   void
 */
void ADC_Init_();

/*
 * Buffer Setup to be retrieved at the frequency specified in the parameters
 * , 16 bits per sample
 *
 * Parameters:
 *   buffer : uint16_t* Buffer to store values
 *   sample_count: uint32_t Samples Count to transfer to buffer
 *
 * Returns:
 *   void
 */
void ADC_Buffer(uint16_t *buffer, uint32_t sample_count);


#endif /* ADC_H_ */
```

```
================================================================================
ADC.c
================================================================================
```

```
/*
 * ADC.c
 *
 *  Created on: Nov 14, 2020
 *      Author: root
 */

#include "MKL25Z4.h"
#include "stdio.h"
#include "stdbool.h"
#include "stdint.h"
#include "ADC.h"


/*
 * Initializing the TPM1 Timer
 *
 * Parameters:
 *   sample: int Initializes the TPM Clock to the requested rate
 *
 * Returns:
 *   void
```

```
 */
void TPM1_Init_(int sample) {

      // Clock Gating
      SIM->SCGC6 |= SIM_SCGC6_TPM1_MASK;

      // Disable TPM for config
      TPM1->SC = 0;

    // mod and counter
      TPM1->MOD = TPM_MOD_MOD(SYS_CLOCK / sample);
      TPM1->CNT = 0;

      // Prescalar Settings
      TPM1->SC = TPM_SC_PS(0) | TPM_SC_CPWMS(0);

}


/*
 * ADC Initializer function to appropriate characteristics
 *
 * Parameters:
 *   void
 *
 * Returns:
 *   void
 */
void ADC_Init_() {
      // Gating
      SIM->SCGC6 |= SIM_SCGC6_ADC0_MASK;

      // Shorter sample time, 16 bit single-ended
      ADC0->CFG1 = ADC_CFG1_ADLPC(0) | ADC_CFG1_ADIV(0) | ADC_CFG1_ADLSMP(0) |
ADC_CFG1_MODE(3) | ADC_CFG1_ADICLK(0);

      // Setting this to all default values
      ADC0->CFG2 = 0;

      // DMA initially disabled
      ADC0->SC2 = ADC_SC2_ADTRG(1) | ADC_SC2_ACFE(0) | ADC_SC2_DMAEN(0) |
ADC_SC2_REFSEL(0);

      // Input Channel for DAC is the input for ADC
      ADC0->SC1[0] = ADC_SC1_AIEN(0) | ADC_SC1_DIFF(0) | ADC_SC1_ADCH(23);

      // Just some TPM1 things
      SIM->SOPT7 = SIM_SOPT7_ADC0ALTTRGEN(1) | SIM_SOPT7_ADC0TRGSEL(9);

}


/*
 * Buffer Setup and analysis for ADC
 *
 * Parameters:
 *   buffer : uint16_t* Buffer to store values
 *   sample_count: uint32_t Samples Count to transfer to buffer
 *
```

```
 * Returns:
 *   void
 */
void ADC_Buffer(uint16_t *buffer, uint32_t sample_count) {

      int i =0;
      // Begin TPM
      BEGIN_TPM1

      for (i =0; i < sample_count; i++) {
            while (!(ADC0->SC1[0] & ADC_SC1_COCO_MASK))
                  ;

            buffer[i] = ADC0->R[0];
      }

      // Stop Sampling
      STOP_TPM1
}
```

autocorrelate.h

```
/*
 * autocorrelate.h: Detect the period of the fundamental frequency in
 * a buffer full of samples
 *
 * Howdy Pierce, howdy@cardinalpeak.com
 */

#ifndef _AUTOCORRELATE_H_
#define _AUTOCORRELATE_H_

#include <stdint.h>

typedef enum {
  kAC_12bps_unsigned,   // 12 bits per sample, unsigned samples (stored in 16 bits)
  kAC_16bps_unsigned,   // 16 bits per sample, unsigned samples
  kAC_12bps_signed,     // 12 bits per sample, signed samples (stored in 16 bits)
  kAC_16bps_signed      // 16 bits per sample, signed samples
} autocorrelate_sample_format_t;


/*
 * Determine the fundamental period of a waveform using
 * autocorrelation
 *
 * Parameters:
 *   samples   Array of samples
 *   nsamp     Number of samples
 *   format    The format for the samples (see above)
 *
 * Returns:
 *   The recovered fundamental period of the waveform, expressed in
 *   number of samples, or -1 if no correlation was found
 */
int autocorrelate_detect_period(void *samples, uint32_t nsamp,
    autocorrelate_sample_format_t format);
```

```
#endif  //  _AUTOCORRELATE_H_

================================================================================
autocorrelate.c
================================================================================
/*
 * autocorrelate.c: Detect the period of the fundamental frequency in
 * a buffer full of samples
 *
 * Howdy Pierce, howdy@cardinalpeak.com
 *
 * Good explanation at
 * https://www.instructables.com/Reliable-Frequency-Detection-Using-DSP-Techniques/
 *
 *   (although his code is poorly written and has a bug, so I wrote this myself)
 */

#include <stdint.h>
#include <stdbool.h>
#include <assert.h>

#include "autocorrelate.h"


/*
 * See documentation in .h file
 */
int
autocorrelate_detect_period(void *samples, uint32_t nsamp,
    autocorrelate_sample_format_t format)
{
      int i=0, k=0;
  int32_t sum = 0;
  int prev_sum = 0;
  int32_t thresh = 0;
  bool slope_positive = false;

  int32_t s1 = 0;
  int32_t s2 = 0;

  sum = 0;
  for (i=0; i < nsamp; i++) {
    prev_sum = sum;
    sum = 0;

    for (k=0; k < nsamp - i; k++) {

      switch (format) {

      case kAC_12bps_unsigned:
        s1 = (int32_t)*((uint16_t*)samples + k) - (1 << 11);
        s2 = (int32_t)*((uint16_t*)samples + k+i) - (1 << 11);
        sum += (s1 * s2) >> 12;
        break;

      case kAC_16bps_unsigned:
        s1 = (int32_t)*((uint16_t*)samples + k) - (1 << 15);
        s2 = (int32_t)*((uint16_t*)samples + k+i) - (1 << 15);
        sum += (s1 * s2) >> 16;
```

```
        break;

      case kAC_12bps_signed:
      case kAC_16bps_signed:
        s1 = *((int16_t*)samples + k);
        s2 = *((int16_t*)samples + k+i);
        sum += (s1 * s2) >> (format == kAC_12bps_signed ? 12 : 16);
        break;
      }
    }

    if (i == 0) {
      thresh = sum / 2;

    } else if ((sum > thresh) && (sum - prev_sum > 0)) {
      // slope is positive, so now enter mode where we're looking for
      // negative slope
      slope_positive = true;

    } else if (slope_positive && (sum - prev_sum) <= 0) {
      // We have crested the peak and started down the other
      // side; actual peak was one sample back
      return i-1;
    }
  }

  // no correlation found
  return -1;
}
```

DAC.h

```
/*
 * DMA.h
 *
 *  Created on: Nov 13, 2020
 *      Author: root
 */

#ifndef DAC_H_
#define DAC_H_

#include "MKL25Z4.h"
#include "stdio.h"
#include "stdint.h"

/*
 * DAC Frequency
 */
#define FREQ                 (48000)
/*
 * System Clock Hz
 */
#define SYS_CLOCK       (48e6)

/*
 * Initializing the DAC Subsystem
 *
 * Parameters:
```

```
 *   void
 *
 * Returns:
 *   void
 */
void DAC_Init_();


/*
 * Initializing the DMA Functionalities
 *
 * Parameters:
 *   void
 *
 * Returns:
 *   void
 */
void DMA_Init_();


/*
 * Initializing the TPM0 Functionalities
 *
 * Parameters:
 *   void
 *
 * Returns:
 *   void
 */
void TPM0_Init_();


/*
 * Function to begin and end the playback of the freuencies,
 *
 * Parameters:
 *   frq: uint16_t* - Buffer to play back
 *   cnt: uint32_t - Sample Count
 *
 * Returns:
 *   void
 */
void DAC_begin(uint16_t *frq, uint32_t cnt);

//void DAC_end();

#endif /* DAC_H_ */
```

==========

DAC.c

==========

```
/*
 * DMA.c
 *
 *  Created on: Nov 13, 2020
 *      Author: root
 */

#include "DAC.h"
#include "MKL25Z4.h"
```

```
#include "stdio.h"
#include "string.h"

// Global Variables
static uint16_t buffer[1024];
static uint32_t count = 0;

#define DAC0_PIN 30


/*
 * Initializing the DAC Subsystem
 *
 * Parameters:
 *   void
 *
 * Returns:
 *   void
 */
void DAC_Init_() {
      SIM->SCGC6 |= SIM_SCGC6_DAC0_MASK; // DAC
      SIM->SCGC5 |= SIM_SCGC5_PORTE_MASK;

      // Pin and Port Setup
      PORTE->PCR[DAC0_PIN] &= ~PORT_PCR_MUX_MASK;
      PORTE->PCR[DAC0_PIN] |= PORT_PCR_MUX(3);

      // Basic Setup
    DAC0->C1 = 0;
    DAC0->C2 = 0;

    DAC0->C0 = DAC_C0_DACEN_MASK | DAC_C0_DACRFS_MASK;

}

/*
 * Initializing the DMA Functionalities
 *
 * Parameters:
 *   void
 *
 * Returns:
 *   void
 */
void DMA_Init_() {

      // Obviously gotta Gate Clock (Dummy)
      SIM->SCGC7 |= SIM_SCGC7_DMA_MASK;
      SIM->SCGC6 |= SIM_SCGC6_DMAMUX_MASK;

      // Disabbling for Configuration
      DMAMUX0->CHCFG[0] = 0;

      // Interrupt Setup
      DMA0->DMA[0].DCR = DMA_DCR_EINT_MASK | DMA_DCR_SINC_MASK | DMA_DCR_SSIZE(2) |
DMA_DCR_DSIZE(2) | DMA_DCR_ERQ_MASK | DMA_DCR_CS_MASK;

      // NVIC Config
      NVIC_SetPriority(DMA0_IRQn, 2);
```

```
      NVIC_ClearPendingIRQ(DMA0_IRQn);
      NVIC_EnableIRQ(DMA0_IRQn);

      // Enable DMA MUX channel with TPM0 overflow as trigger
      DMAMUX0->CHCFG[0] = DMAMUX_CHCFG_SOURCE(54);
}


/*
 * Initializing the TPM0 Functionalities
 *
 * Parameters:
 *   void
 *
 * Returns:
 *   void
 */
void TPM0_Init_() {

      // Gating
      SIM->SCGC6 |= SIM_SCGC6_TPM0_MASK;

      // 48 Mhz
      SIM->SOPT2 |= (SIM_SOPT2_TPMSRC(1) | SIM_SOPT2_PLLFLLSEL_MASK);

      // Begin Configuration
      TPM0->SC = 0;

      // Load the mod and counter
      TPM0->MOD = TPM_MOD_MOD(SYS_CLOCK / (SYS_CLOCK/1000));
      TPM0->CNT = 0;

      TPM0->SC = TPM_SC_PS(0) | TPM_SC_CPWMS(0) | TPM_SC_CMOD(1) | TPM_SC_DMA_MASK;
}


/*
 * Function to begin and end the playback of the freuencies,
 *
 * Parameters:
 *   frq: uint16_t* - Buffer to play back
 *   cnt: uint32_t - Sample Count
 *
 * Returns:
 *   void
 */
void DAC_begin(uint16_t *frq, uint32_t cnt){

      TPM0->SC &= ~TPM_SC_CMOD_MASK;
      // For interfacing with the global count function
      count = cnt;

      memcpy(buffer, frq, cnt*2);

      // Begin Configuration
      TPM0->SC |= TPM_SC_CMOD(1);

      DMA0->DMA[0].SAR = DMA_SAR_SAR((uint32_t) buffer);
      DMA0->DMA[0].DAR = DMA_DAR_DAR((uint32_t) (&(DAC0->DAT[0])));
```

```
      // Number of bytes to transfer
      DMA0->DMA[0].DSR_BCR = DMA_DSR_BCR_BCR(count * 2);
      DMAMUX0->CHCFG[0] |= DMAMUX_CHCFG_ENBL_MASK;
}

/*
 * IRQ Handler
 */
void DMA0_IRQHandler(void)
{
      // Goddamit, have to Clear done flag in IRQ
      DMA0->DMA[0].DSR_BCR |= DMA_DSR_BCR_DONE_MASK;

      // DMA playback
      DMA0->DMA[0].SAR = DMA_SAR_SAR((uint32_t) buffer);
      DMA0->DMA[0].DAR = DMA_DAR_DAR((uint32_t) (&(DAC0->DAT[0])));

      // Bytes to transfer
      DMA0->DMA[0].DSR_BCR = DMA_DSR_BCR_BCR(count * 2);
      DMAMUX0->CHCFG[0] |= DMAMUX_CHCFG_ENBL_MASK;
}
```

```
================================================================================
fp_sine.h
================================================================================
/*
 * fp_sine.h
 *
 *  Created on: Nov 12, 2020
 *      Author: root
 */

#ifndef FP_SINE_H_
#define FP_SINE_H_


#include <stdint.h>

#define TRIG_SCALE_FACTOR    (2037)
#define HALF_PI                    (3200)
#define PI                         (6399)
#define TWO_PI                     (12799)


/*
 * Computes sine of x within max ranges
 *
 * Parameters:
 *    x     Expressed in radians * TRIG_SCALE_FACTOR
 *
 * Returns:
 *    sin(x) * SCALE_FACTOR
 *
 */
int32_t fp_sin(int32_t x);


/*
```

```
 * Performs linear interpolation using fixed-point math
 *
 * Parameters:
 *    x       Input x value
 *    (x1, y1), (x2, y2):  First pt, Second pt
 *
 * Returns:
 *    Interpolated value
 */
int32_t interpolate(int32_t x, int32_t x1, int32_t y1, int32_t x2, int32_t y2);


/*
 * Converts from degrees (unscaled) into radians (scaled)
 *
 * Parameters:
 *    val : angle in degrees
 *
 * Returns:
 *    Radian Equivalent of degrees
 */
int32_t convert_to_radians(int val);

#endif /* FP_SINE_H_ */
```

```
================================================================================
fp_sine.c
================================================================================
```

```
/*
 * fp_sine.c
 *
 *  Created on: Nov 12, 2020
 *      Author: root
 */

#include "fp_sine.h"
#include "MKL25Z4.h"
#include "stdio.h"
#include "stdint.h"
#include "assert.h"

#define TRIG_TABLE_STEPS     (32)
#define TRIG_TABLE_STEP_SIZE (HALF_PI/TRIG_TABLE_STEPS)

static const int16_t sin_lookup[TRIG_TABLE_STEPS+1] =
  {0, 100, 200, 299, 397, 495, 591, 686, 780, 871, 960, 1047,
   1132, 1214, 1292, 1368, 1440, 1509, 1575, 1636, 1694, 1747,
   1797, 1842, 1882, 1918, 1949, 1976, 1998, 2015, 2027, 2035,
   2037};


/*
 * Computes sine of x within max ranges
 *
 * Parameters:
 *    x     Expressed in radians * TRIG_SCALE_FACTOR
 *
 * Returns:
 *    sin(x) * SCALE_FACTOR
 *
```

```
 */
int32_t fp_sin(int32_t x)
{
  int32_t idx;
  int sign = 1;

  // If x < -PI, add 2*PI repeatedly until -PI <= x <= PI
  while (x < -PI)
    x += TWO_PI;

  // If x > PI, subtract 2*PI repeatedly until -PI <= x <= PI
  while (x > PI)
    x -= TWO_PI;

  assert(-PI <= x && x <= PI);

  // Fold the range [-PI, 0] into [0, PI]
  if (x < 0) {
    x = -x;
    sign = -1;
  }

  // Fold the range (HALF_PI, PI] into the range [0, HALF_PI]
  if (x > HALF_PI) {
        x = PI - x;
  }

//  assert(0 <= x && x <= HALF_PI);

  idx = TRIG_TABLE_STEPS * x / HALF_PI;
  int32_t x1 = idx * TRIG_TABLE_STEP_SIZE;

  // exact match: no interpolation needed
  if (x1 == x)
    return sign * sin_lookup[idx];

  // else, interpolate
  int32_t x2 = (idx+1) * TRIG_TABLE_STEP_SIZE;
  int32_t interp = interpolate(x, x1, sin_lookup[idx], x2, sin_lookup[idx+1]);

  return sign * interp;
}


/*
 * Converts from degrees (unscaled) into radians (scaled)
 *
 * Parameters:
 *    val : angle in degrees
 *
 * Returns:
 *    Radian Equivalent of degrees
 */
int32_t fp_radians(int val)
{
  return val * PI / 180;
}

/*
```

```
 * Performs linear interpolation using fixed-point math
 *
 * Parameters:
 *    x       Input x value
 *    (x1, y1), (x2, y2):  First pt, Second pt
 *
 * Returns:
 *    Interpolated value
 */
int32_t interpolate(int32_t x, int32_t x1, int32_t y1, int32_t x2, int32_t y2)
{
  return ( (((x2 - x1)/2) + ((x - x1) * (y2 - y1))) / (x2 - x1) + y1);
}
```

==========

test_sin.c

==========

```
/*
 * test_sin.c
 *
 *  Created on: Nov 14, 2020
 *      Author: root
 */


#include "stdio.h"
#include "fp_sine.h"
#include "MKL25Z4.h"
#include "fsl_debug_console.h"
#include "math.h"

void test_sine()
{
double act_sin;
double exp_sin;
double err;
double sum_sq = 0;
double max_err = 0;
int i= 0;
for (i=-TWO_PI; i <= TWO_PI; i++) {
exp_sin = sin( (double)i / TRIG_SCALE_FACTOR) * TRIG_SCALE_FACTOR;
act_sin = fp_sin(i);
err = act_sin - exp_sin;
if (err < 0)
err = -err;
if (err > max_err)
max_err = err;
sum_sq += err*err;
}
PRINTF("max_err=%f sum_sq=%f\n", max_err, sum_sq);
}
```

==========

test_sin.h

==========

```
/*
 * test_sin.h
 *
 *  Created on: Nov 14, 2020
```

```
 *      Author: root
 */

#ifndef TEST_SIN_H_
#define TEST_SIN_H_

#include "stdint.h"

void test_sine();

#endif /* TEST_SIN_H_ */
```

util.h

```
/*
 * util.h
 *
 *  Created on: Nov 14, 2020
 *      Author: root
 */

#ifndef UTIL_H_
#define UTIL_H_

#include "MKL25Z4.h"

/*
 * Fill a buffer with 12-bit unsigned samples, representing the
 * specified tone
 *
 * Parameters:
 *    input_freq    The frequency of the tone to be played
 *    buffer       The buffer to store the samples into
 *    size     The size of the buffer, in number of samples
 *
 * Returns:
 *    The number of samples actually computed
 *
 * This function pre-computes the samples for the given tone. Used
 * during DAC. function fills up the buffer.
 */
size_t tone_to_samples(int input_freq, uint16_t *buffer, size_t size);


/*
 * Analyzes the Input Samples
 *
 * Parameters:
 *   buffer    Buffer to analyze
 *   count    Sample Size
 *
 * Return:
 * void
 *
 */
void analysis(uint16_t *buffer, uint32_t count);

#endif /* UTIL_H_ */
```

util.c

```
================================================================================
/*
 * util.c
 *
 *  Created on: Nov 14, 2020
 *      Author: root
 */


#include "util.h"
#include "autocorrelate.h"
#include "assert.h"
#include "stdio.h"
#include "stdint.h"
#include "DAC.h"
#include "fp_sine.h"
#include "fsl_debug_console.h"


/*
 * Fill a buffer with 12-bit unsigned samples, representing the
 * specified tone
 *
 * Parameters:
 *    input_freq    The frequency of the tone to be played
 *    buffer       The buffer to store the samples into
 *    size     The size of the buffer, in number of samples
 *
 * Returns:
 *    The number of samples actually computed
 *
 * This function pre-computes the samples for the given tone. Used
 * during DAC. function fills up the buffer.
 */
size_t tone_to_samples(int input_freq, uint16_t *buffer, size_t size) {

      int32_t temp;
      int i=0;
      int cycles = size / ((FREQ + (input_freq/2)) / input_freq);
      int samples = ((FREQ + (input_freq/2)) / input_freq) * cycles;

      for (i=0; i < samples; i++) {
            // Needs to be converted appropriately
            temp = fp_sin(i * TWO_PI / ((FREQ + (input_freq/2)) / input_freq) ) +
TRIG_SCALE_FACTOR;
            buffer[i] = temp;
      }

      PRINTF("Generated %d samples at %d Hz; computed period=%d, observed=%d\r\n",
                  samples, input_freq, ((FREQ + (input_freq/2)) / input_freq),
        autocorrelate_detect_period(buffer, samples, 0));

   return samples;
}


/*
 * Analyzes the Input Samples
```

```
 *
 * Parameters:
 *   buffer    Buffer to analyze
 *   count    Sample Size
 *
 * Return:
 * void
 *
 */
void analysis(uint16_t *buffer, uint32_t count) {

      int i=0;
      uint32_t max=0, sum = 0;
      int min = -1;

      // Keeps to within limit
      for (i=0; i < count; i++) {
            if (buffer[i] > max) {
                  max = buffer[i];
            }
            if (buffer[i] < min) {
                  min = buffer[i];
            }

            sum+= buffer[i];

      }

      int temp = autocorrelate_detect_period(buffer, count, 1);
      PRINTF("min=%u max=%u avg=%u period=%d frequency=%d Hz\r\n",
            min, max, sum / count, temp, 96000 / temp);

}
```