AN-1191
APPLICATION NOTE

One Technology Way • P.O. Box 9106 • Norwood, MA 02062-9106, U.S.A. • Tel: 781.329.4700 • Fax: 781.461.3113 • www.analog.com

ADV7182 CMRR Measurements Across Frequency Using ADSP-BF527

by Witold Kaczurba

INTRODUCTION

This application note describes techniques for measuring the noise spectrum for video ADCs using the Blackfin® ADSP-BF527. An external noise can have negative impact on video quality. Finding a source of noise might be difficult, especially in the complex systems using digital processing and/or radio frequency. The measurement is especially important in case of differential video using long cable runs (such as in automotive applications using reversing cameras with cables going from the head unit to the back of the car).

The method described in this application note allows analysis of noise across a frequency spectrum as well as measuring the noise floor level of video ADC. For this purpose, the ADV7182 is set into raw ADC mode that constantly converts all incoming analog signals into 10-bit digital codes output via pixel lines { P[7:0], HS, VS } with an accompanying clock. In this particular mode, the synchronization signal is not processed.

The ADC is connected to a 10 kHz differential signal generator providing a reference 1 V peak-to-peak signal.

The ADSP-BF527 (or similar) acts as a data grabber transferring samples straight to the SDRAM. Once collected, data can be transferred to a PC for further processing and to perform FFT.

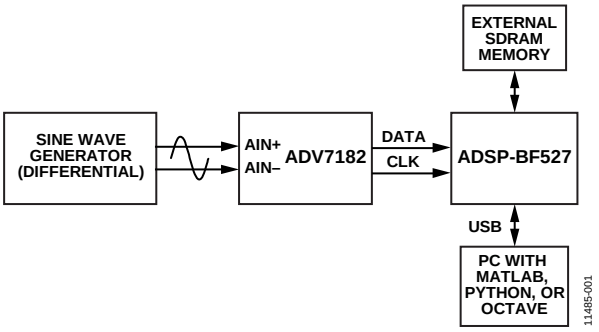

Figure 1. Simplified Setup

TABLE OF CONTENTS

REVISION HISTORY

6/13—Revision 0: Initial Version

CMRR MEASUREMENT

The measurements of CMRR must be performed across frequency. Factors that can have an impact on measurements are differential pairs that are not kept together, termination mismatch, and layout practices. The input resistor matching has a crucial effect on CMR measurement. Matching should be of 0.1% accuracy.

Test 1: Adjusting and Collecting Data for Single-Ended Signal

During the first measurement test, a sine wave of known frequency should be applied to one of the terminals (AIN+) whereas the other should be connected to ground. The amplitude of the sine wave should be adjusted to utilize the full range of the ADC, without saturating the ADC. The digital data should be collected using a digital grabber.

Test 2: Measuring

During the second measurement, a sine wave of the same frequency and amplitude as in the first test should be applied to both terminals. Common-mode rejection of the ADC amp will reject the sine wave.

Result

In order to quantify results, data collected from the digital output from the first and the second test should be plotted in a semi-logarithmic scale showing absolute FFT values across frequency. The comparison between plots from Test 1 and Test 2 should provide information on how common mode is rejected. Since this data is presented in a logarithmic scale—subtracting the result of Test 1 from the result from Test 2 for the particular frequency should result in the CMRR in dB. The code presented in Listing 3 has been adjusted to show the peak-to-peak sine wave at 0 dB. The example shown in Figure 2 and Figure 3 shows −59 dB CMRR attenuation for 1% resistors. Lab results with 0.1% matched resistors showed −70 dB attenuation.

The data used for these figures has been generated for illustration purposes.

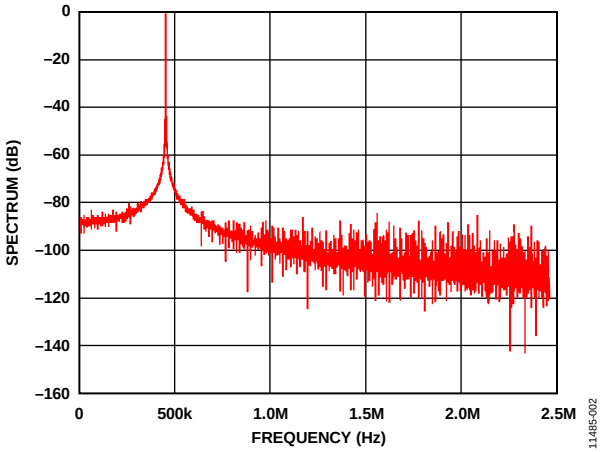

Figure 2. Sine Wave (45772 Hz) of Full 10-Bit Range Adjusted to 0 dB

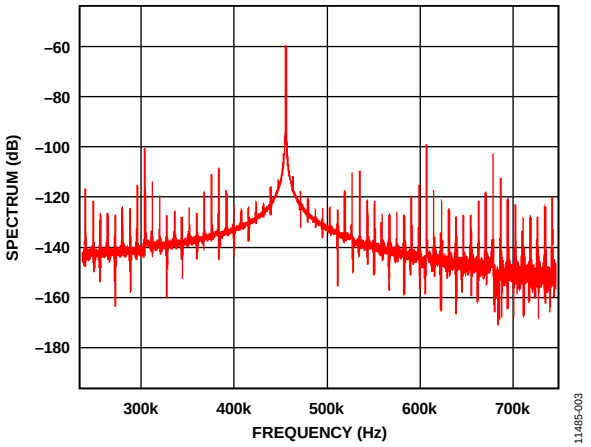

Figure 3. Frequency Spectrum Shows a −59 dB Spike at 45772 Hz (Synthesized Data)

ADV7182 SCRIPT AND SCHEMATICS

The following script sets the ADV7182 in ADC sampling mode, using CVBS_P on AIN1 and CVBS_N on AIN2. Figure 3 shows the schematic that includes input recommended termination for the ADV7182 as well as connection to ADSP-BF527 capturing the data (Port F and Port J).

Listing 1. Special ADV7182 I²C Writes

```
42 0F 00 ; Exit Power Down Mode
42 00 0E ; INSEL = CVBS_P in on Ain 1, CVBS_N in on Ain2
42 02 04 ; Analog Standard Selection
42 07 00 ; Auto-detect off
42 03 0C ; Enable Pixel & Sync output drivers
42 1D 40 ; Enable LLC output driver
42 13 00 ; Enable INTRQ output driver
42 64 10 ; Power up Xtal path
42 14 00 ; Special ADC test mode
42 52 C0 ; Special ADC test mode
42 5F 08 ; Special ADC test mode
42 6C 80 ; Special ADC test mode
42 60 A0 ; Special ADC test mode
42 28 80 ; Special ADC test mode
42 1D 40 ; Special ADC test mode
```

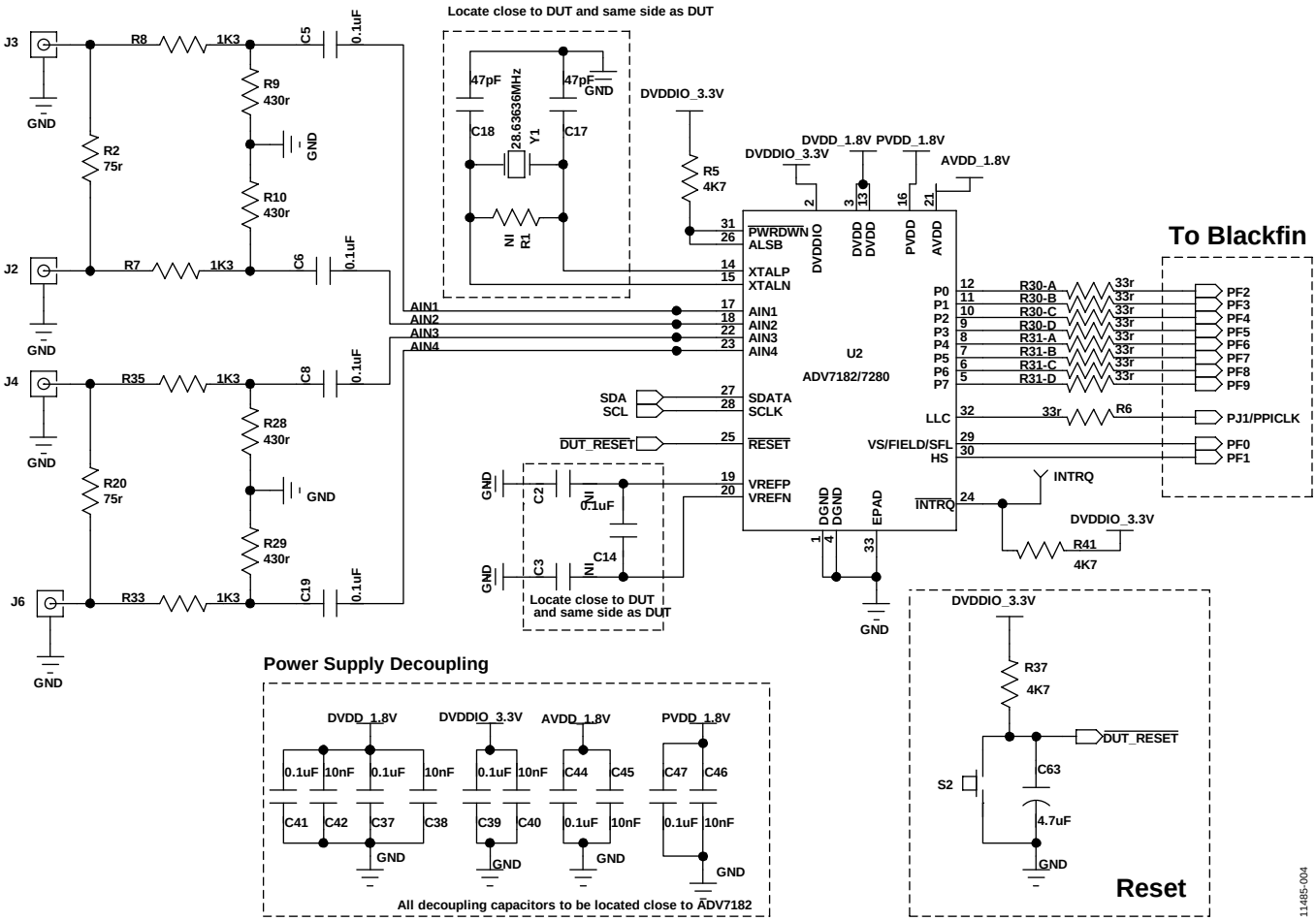

Figure 4. ADV7182 Connection to Blackfin

BLACKFIN AS A DATA GRABBER

The Blackfin family features a parallel port interface (PPI) allowing for bidirectional parallel data transfers of various types. Those include video transfers such as BT656, raw data with and without additional external synchronization signals. Since the ADV7182 is configured in raw ADC mode (without processing synchronization signals), the Blackfin must be configured to receive incoming raw data, without external synchronization signals, with a clock rate 27.0 MHz or 28.6363 MHz. The built-in PPI interface is configured to perform data transfers facilitating DMA mechanism allowing for direct transfer to external SDRAM memory. The Blackfin's core does not participate therefore in receiving data and can be occupied to execute other code.

Displaying and Analyzing Data on the PC

One of the easiest ways of performing FFT and displaying graphical representation without buying an expensive mathematical package is Python(x, y). This software package contains tools and libraries for mathematical calculations and provides visual representation (charts, plots, and so on). The package allows the performance of fast Fourier transforms and displays this information in a very quick and convenient way on a PC.

Blackfin Connection to PC over USB

The Blackfin DSP processor family offers a USB interface and a UART connectivity that can be used for data transfer to the PC and to maintain link. Blackfin's USB interface. It can be programmed to work in a number of various modes, including CDC, HID, mass storage, or general bulk.

This application note outlines how to modify existing Visual DSP++ 5.0 USB examples into general bulk mode allowing for data transfer with a PC. The example is based on the ADSP-BF527, but can be easily transferred to suit other USB interfaces.

Blackfin Code

The original source code covered in the VDSP++ examples allows for a number of certain USB commands. In order to facilitate ADV7182 programming via an I^2C interface and setting Blackfin's PPI mode that grabs the data, three additional commands were added into the code: CONFIGURE_ADV, GRAB_DATA, and READ_VIDEO_DATA. The first one configures the ADV part using the I^2C interface (See Listing 1) into RAW-ADC mode in which data is sampled at the XTAL clock frequency and output to the pixel output port without processing syncs. The next function configures the PPI interface to read data from the pixel port (refer to Listing 2). The last one (READ_VIDEO_DATA) allows data to be sent to the host computer.

Listing 2. PPI Configuration

```
//Configuring PPI pins PF0..PF15 to PPI
*pPORTF_MUX = 0x0000;
*pPORTF_MUX |= (1 << 12);   // Enable PPICLK pin
*pPORTF_FER = 0xFFFF;         // PF0..PF15 to PPI func.
//Zeroing registers in case the hold previous values
*pPPI_CONTROL = 0;
ssync();
*pDMA0_CONFIG = 0;
ssync();
//Configuring PPI and DMA:
// X_COUNT: 1024 samples * 2 bytes = 2048 bytes per line
// Y_COUNT: 1024 lines * 2048 bytes = 2097152 bytes total
*pDMA0_START_ADDR = frame_raw_buffer1;
*pDMA0_X_COUNT = 1024;
*pDMA0_Y_COUNT = 1024;
*pDMA0_X_MODIFY = 2;     // 2 byte data
*pDMA0_Y_MODIFY = 2;
*pDMA0_CONFIG=FLOW_STOP|DMA2D|WDSIZE16| WNR;
ssync();
// PPI Configuration (uses only PPICLK, no frame syncs)
// 16-bit data, 0 framesync with internal trigger, PPI receiver
*pPPI_CONTROL=DLEN_16| FLD_SEL|PORT_CFG| XFR_TYPE;
 *pPPI_DELAY = 0;
*pPPI_COUNT = 1024 - 1;
ssync ();
// Enabling DMA and PPI
*pDMA0_CONFIG |= DMAEN; //Enable DMA0
ssync();
*pPPI_CONTROL |=  PORT_EN; //Enable PPI
ssync();
```

The VDSP++ project can be downloaded from Analog Devices EngineeringZone®.

Host Application

The host application has been simply extended to match additional functions that Blackfin offers. CONFIGURE_ADV, GRAB_DATA require no additional data to be transferred, whereas function READ_VIDEO_DATA splits the 2 megabytes buffer into a number of 65536-byte long transmission bulks.

All functions were implemented in Visual C++ in a manner allowing for execution from the command line.

`hostapp.exe -b`	Configures ADV7182 part
`hostapp.exe -g`	Configures Blackfin for grabbing data
`hostapp –I FILE START COUNT`	Dumps the COUNT-bytes from the device to FILE at START address

The received file contains 16-bit data grouped into 2-byte codes, with the first byte representing the less significant byte (PF7 to PF0) and then the most significant byte (PF15 to PF8).

Processing the Received Data

Once received, data can be processed and its frequency spectrum can be shown using the PC. Listing 3 shows the simplified processing of a file containing captured data and displaying its frequency spectrum. The code is written in Python(x,y). For simplicity, the example does not apply to windowing or any filtering.

Listing 3. Example Python(x,y) Script

```
import time
from pylab import plot, show, title, xlabel, ylabel, subplot
from scipy import fft, arrange
from math import log10

Fs = 28.6363E6
pix_scale = 4

def newlog10(x):
    # Returns -90dB for log10(0) - in case FFT outputs zeros
    if x == 0:
       return -90 # -90dB
    return log10(x)

def scale_down(data, pix_scale):
    output = []
    if pix_scale <> 1:
        for i in range(0, len(data), pix_scale):
            output.append(round(sum(data[i:i+pix_scale])/float(pix_scale)))
        return output
    return data

def show_plots(data, t, Fs=28.6363E6, fctr=242):
    global pix_scale
    subplot(2,1,1) # plotting signal in time-domain
    plot(t[::pix_scale],scale_down(data, pix_scale))
    xlabel('Time')
    ylabel('Amplitude')
    subplot(2,1,2)
    Y = abs(fft(data))
    for i in range(0, len(Y)/2):
        Y[i] = ((float(Y[i]))/(len(Y)))/fctr
    Y = Y[range(len(Y)/2)] # trimminig to first half
    frq = arange(len(Y))/float(len(Y)) * (Fs / 2)
    for i in range(0, len(Y)): # displaying in semi-log-scale
        Y[i] = 20 * newlog10(Y[i])
    plot(frq,Y,'r') # plotting signal in frequency-domain
    xlabel('Freq (Hz)')
    ylabel('log10|Y(freq)|')
    show()


def read_file(filename, nbits=10):
    f = open(filename,'rb')
    bin_data = f.read()
    data = []
    for i in range(0, len(bin_data)-1, 2):
       bit_mask = (2<<(nbits-1)) – 1
       cur_value = ((ord(bin_data[i]) + (ord(bin_data[i+1]) *\ 256)) & bit_mask)
       data.append(cur_value)
    return data


# Main part:
data = read_file('data.bin', 10)
for i in range(0, len(data)):  # remove DC component
     data[i] = data[i] – 512
t = (arange(0, len(data)))/(Fs) # 1Msamples
show_plots(data, t)
```

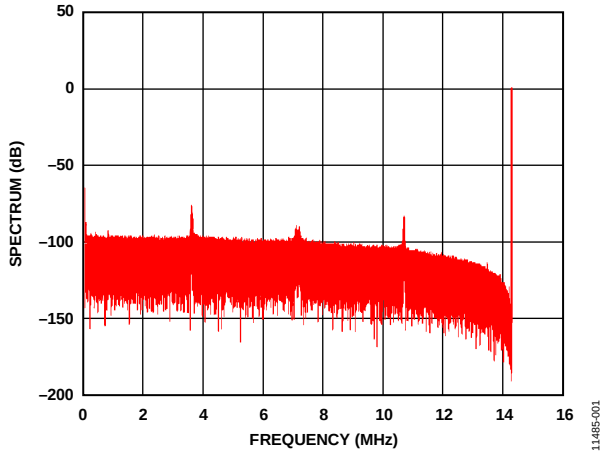

Figure 5. Frequency Spectrum 0 MHz to 14.31 MHz for 10 kHz Sine Wave

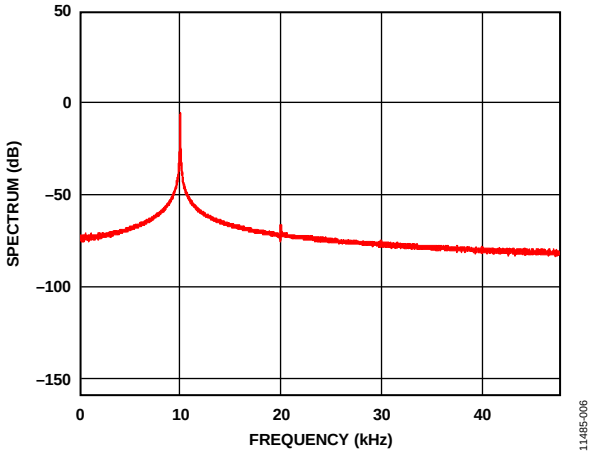

Figure 6. Frequency Spectrum 0 kHz to 50 kHz for 10 kHz Sine Wave

I²C refers to a communications protocol originally developed by Philips Semiconductors (now NXP Semiconductors).