Authors: Eric Oberla (UChicago)

Contents

1 Overview

This document provides a brief overview of operating the SURFv5 circuit board based on the LAB4D waveform sampling ASIC. The SURFv5 board was designed by Patrick Allison (OSU) and Jarred Roberts (UHawai'i). The LAB4D ASIC was designed by Gary Varner (UHawai'i). P. Allison developed the FPGA firmware. Testing of this hardware began in June, 2016. This work is part of the ANITA project.

The SURFv5 board, pictured in Fig. 1, houses 12 LAB4D ASICs corresponding to 12 RF channels. Each ASIC has only a single channel per chip to minimize channel-to-channel crosstalk via the IC package bondwires and/or on-die coupling. Each RF channel is equipped with a bandpass filter, nominally $\sim$140 - 1000 MHz[1]. The SURFv5 front panel also has two 'SYNC' inputs which are split and coupled onto the RF signal lines after the bandpass filter. The SYNC inputs pass baseband signals up to $\sim$250 MHz, which allows *in situ* time synching between channels/SURFs and the parallelization of the LAB4D timebase calibrations.

For the ANITA mission, the plan is to run the LAB4D at 3.2 GSa/s with 1024 samples-per-event (event record window = 320 ns). The LAB4D has a total of 4096 samples, which allows the (analog) multi-buffering of four events and simultaneous read/write operation, reducing dead-time induced latency.

A key feature of the LAB4D ASIC is the ability to trim the inherently non-uniform time-base delays of a CMOS voltage-controlled delay line, reducing previously required calibration and processing overhead for these ASICs. A tuning procedure is included that has demonstrated a spread on the sampling intervals of a few picoseconds.

1.1 code respositories

- The firmware for the SURFv5 is available at P. Allison's github:
  https://github.com/barawn/firmware-surf5
- Python software for running a SURFv5 testbench can be found here:
  https://github.com/ejobe/surf-python

Notes about the Python sofware: In the main directory, `surf.py` and `surf_data.py` are the main SURFv5 drivers. Subdirectory `calibrations/` stores the `surf_calibrations.json` cal file that holds channel-specific LAB4D register values as well as pedestal data, `timing/` has the timebase-tuning modules. The file `utils/surf_constants.py` defines a number of variables for the DAQ system as well as default values for the LAB4D registers.

2 Getting Started

(Leaving out detailed instructions on how to program the FPGA. Either use a standard Xilinx cable and iMPACT loader or use P. Allison's nifty xcvd-anita daemon. The latter, however, is currently unable to load the SPI flash device.)

1. **Download the software linked above.** There's nothing really special here: You'll need some standard Python packages like numpy, json, matplotlib, and scipy installed.

[1] filters used: low pass is Mini-Circuits LFCN-1000+; high pass is Mini-Circuits HFCV-145+

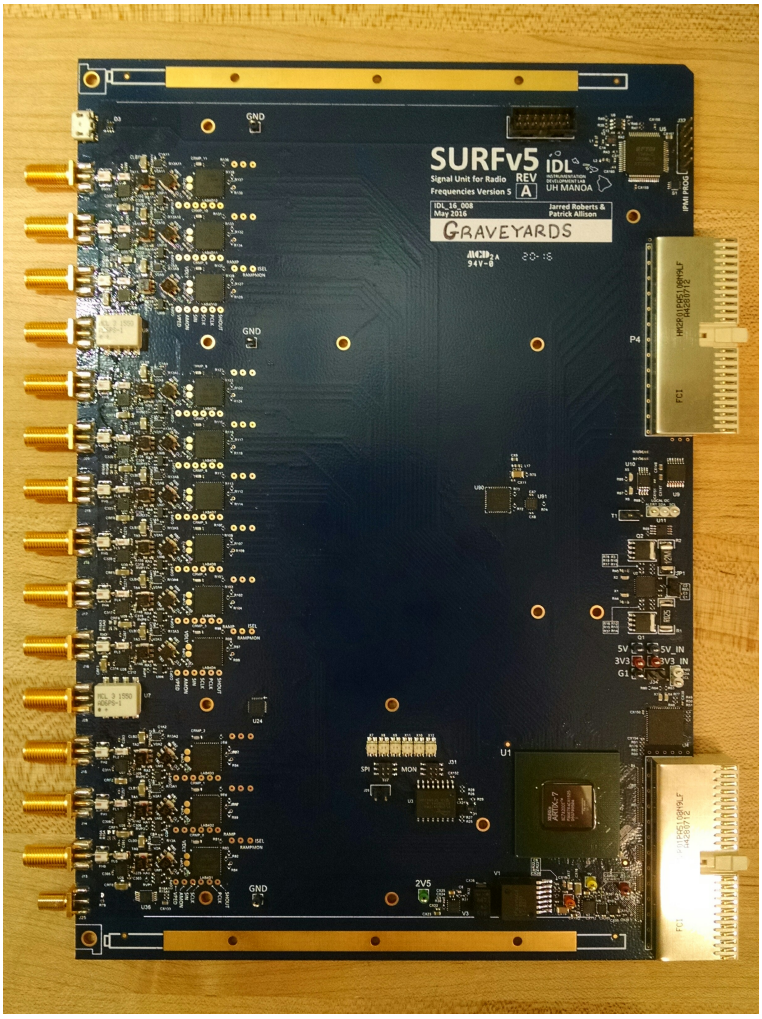


Figure 1: SURFv5 board ('GRAVEYARDS')

2. **Run surf_board.py** This module sets up default configurations, enables the DLL, and does some basic calibrations if it's the first time the board has been operated.

   To run: `python surf_board.py [BoardName]`

   Alternatively this can be run in the Python interpreter:

   ```
   dev=surf_board.do(BoardName) #dev returns an instance of the Surf class
   ```

   where BoardName is the human-readable all-caps 'surf break' identifier written on the board. This identifier is used to read/write to the surf_calibrations.json file stored in `calibrations/`.

3. **Take pedestal data**. Save the fixed-pattern pedestal data to the `surf_calibration.json` file. (Also see Appendix A)

   To run: `python surf_data.py pedestal`

   The pedestal data is also saved to an ascii file: `calibrations/peds.temp`

4. **Check baseline waveforms** Check to see if pedestal-subtracted baseline looks OK. Many ways to do this; one option to use the 'scope' option:

```
python surf_data.py scope [Channel]
```

where Channel is 0-11 (or if undefine will plot all channels). If data has spikes or an apparent RMS value above 10 ADC counts, re-run the pedestal calibration. (Also see Appendix A)

3 Basic Operations

A few DAQ operations are described here. These are shown in more detail within the IPython notebook attached in Appendix A:

setting the pedestal level

The pedestal voltage is controlled with an I2C DAC. The pedestal level is accessed within the Surf class.

```
dev=surf.Surf()
dev.vped  #prints pedestal level
dev.set_vped(level)  #sets pedestal level
```

where level is an integer between 0 and 4096.

saving/loading pedestal data

Pedestal data can be taken from the command line (make sure no active signaling on the input!):

To run: `python surf_data.py pedestal`

This runs the default 160 events to form the pedestal calibration data.
To read the pedestal data:

```
devData=surf_data.SurfData()
devData.loadPed() #this calls calibrations.surf_calibrations.read_pedestals()
ped = devData.pedestals #pedestals stored in class variable
```

logging data

Data are saved to a flat ascii file (12 columns, 1024*NumEvents rows). Data can be logged using:

```
python surf_data.py log [NumEvents] [filename]
```

pedestal scan

The DC transfer curve can be extracted by scanning the pedestal voltage generated by the on-board DAC:

```
python surf_data.py lin [Start] [Stop] [Interval]
```

where Start is the DAC start code, Stop is the DAC stop code, and Interval is the scan interval. If not provided, default values are used.

The SurfData class also has a function to create a channel-level LUT based on the mean transfer function of all the storage cells. TO-DO: implement per-storage-cell LUT (i.e. 4096 LUTs/LAB4D). Add option to apply this to readout.

4 Calibrating the LAB4D Timebase

The LAB4D ASIC has 128 primary sampling cells, which are toggled by an equivalent length voltage-controlled delay line. Due to the nature of chip fabrication, these 128 sample intervals will have some variance around the nominal unit delay. The LAB4D is equipped with a trim DAC on each delay cell to narrow the distribution of these sample intervals on-chip. Additionally, the LAB4D has an on-chip delay-locked loop (DLL) that enables exactly one input clock period to traverse those 128 sample cells. The DLL functionality enables synching between channels when broadcasting a common, low-jitter clock to all LAB4Ds.

In order to find the nominal parameters for tuning the timebase we first tune the DLL, which already provides quite good-looking waveform data. Further, we include a process to tune the trim DACs to minimize the variation of the sample intervals. The process of tuning the DLL is a few orders of magnitude quicker than tuning the sample intervals.

4.1 Tuning the DLL feedback trim

It is assumed that the DLL is enabled on the LAB4D, as it is in the `surf_board.py` initialization script. If not, call the `dll(lab, mode=True)` function [`lab`=15 to address all LABs on the board, otherwise specify 0-11] in the `LAB4Controller` class in `surf.py`.

There are primarily two parameters in the LAB4D register space that manage the tuning of the DLL: `sstoutfb` and `vtrimfb`. `sstoutfb` specifies the delay tap in the voltage-controlled delay line to phase-compare to the input reference clock. At the moment this is hard-coded to `sstoutfb` = 104. It is likely that this parameter needs to be uniquely set to 105 or 103 for specific LAB4D chips within the lot.

The DAC `vtrimfb` sets the total amount of delay in the DLL by finely tuning the the delay of the `sstoutfb` VCDL pick-off. The optimal `vtrimfb` can be found by scanning this parameter and fitting sine waves in groups of 128 cells corresponding to the primary sampling bank (i.e. excluding the DLL wraparound cells from the fit) at each `vtrimfb` setting. The value is then extracted by comparing the fit results to the known input frequency and nominal sampling rate. Lower frequencies are better for this tuning: $\sim$100-230 MHz and advisably excluding multiples of 25 MHz, the input clock frequency. To expediate the process, `vtrimfb` can be incremented at intervals of $\sim$10 counts and fit with a 3rd or 4th order polynomial.

A function to tune the `vtrimfb` parameter is given: `timing/tune_dll_trim.py`. An example on how to use this for tuning, saving, and loading trim values is given in Appendix B.

Figure 2 shows the `vtrimfb` tuning curves for `sstoutfb` set to 104 and 105 on a single LAB4D. We picked `sstoutfb`=104 as it appears to give the widest stable range of tuning around the nominal sampling rate, 3.2 GSa/s.

Figure 3 shows multiple `vtrimfb` taken during at differerent iterations during a VCDL-trim-DAC tuning run (described in next section). The plot shows that the optimal value for `vtrimfb` varies very little even when changing the VCDL trim-DAC values.

4.2 Tuning the VCDL trim DACs

A straightforwared Newton's Method minimzation is currently used to tune the VCDL trim DACs. The functions are defined in `timing/timing.py` and an executable to run the minimization is `timing/run_timing_cal.py`, which should be edited to specify the channels to tune, the number of iterations, the output filename, and the number of events per iteration.

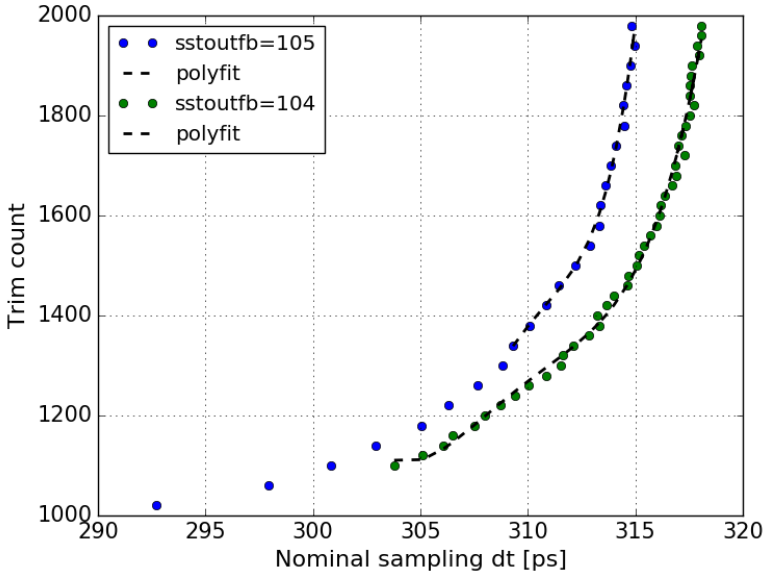


Figure 2: Scanning `vtrimfb` and fitting sine wave data to determine the optimal parameters for the DLL.

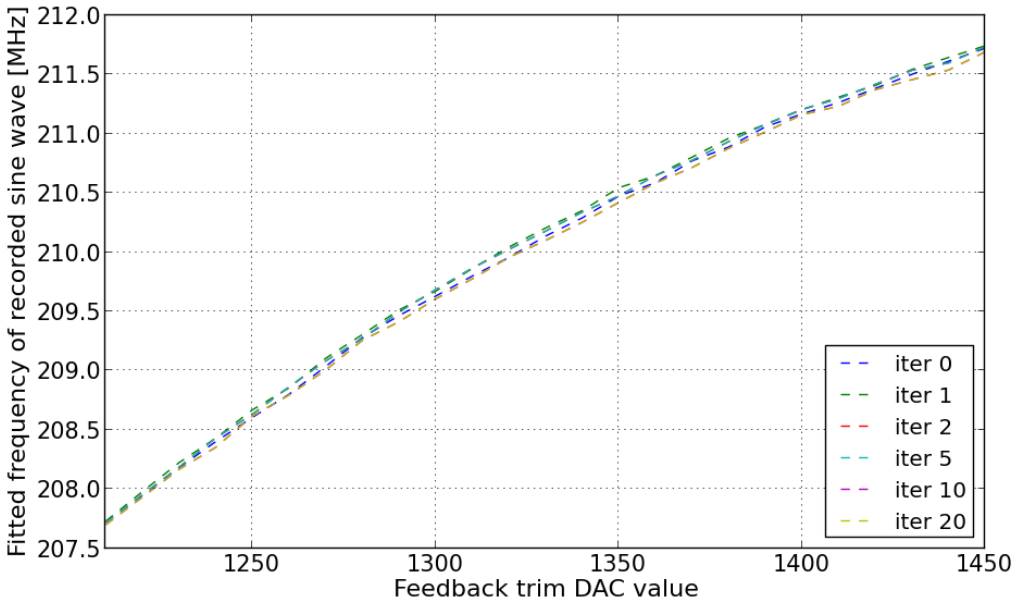


Figure 3: Fitted sine-wave frequency vs the feedback trim value using a 210.0 MHz sine-wave input on channel 0 of 'CANOES'. The curve is robust against adjuments to the time-base trim dac values (next section) so presumably can just be calibrated once. [This measurement shows the turning curve at several iterations in the trim-dac tuning procedure.]

The minimazation curve used in this procedure is defined in `timing/trim_minimization_curve.py`. It's mostly ad-hoc, but does the trick. The scheme is based on low-statistics measurements of the individual trim-dac-value vs. sample-interval-time curve shown in Figure 4

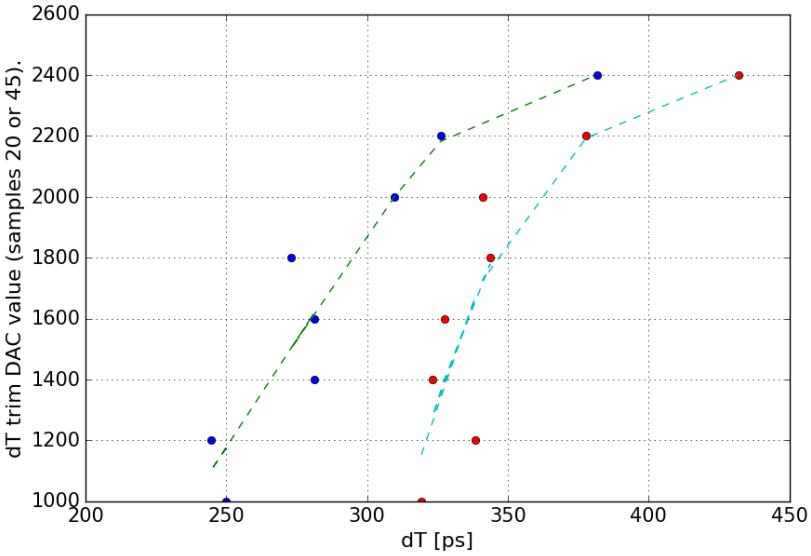


Figure 4: dT trim DAC value vs. measured sampling interval for an even and odd sampling cell.

The minimization method is based on counting the zero-crossings of sine-waves in the LAB4D data. The effective sampling interval is determined by the relative occupancy in each sample-bin. At each iteration, the sampling interval is computed and the trim DAC is adjusted based on the minimization curve. Because the zero-crossing method only uses a small amount of the total waveform around the zero-point, this method requires a large number of events. I have found that 4000-8000 events per iteration (=32,000-64,000 passes at the primary sampling cells) yield a fairly robust result. Of course, higher statistics are always desired and probably required to push down to the $\sim$ps level (though understanding the temperature dependence and the the linearity of the trim DACs become important issues at this point anyway).

Example results from a tuning run on Channel 0 of the SURFv5 called 'CANOES' is shown in Fig 5. The output of the tuning executable is a set of .json files. In this run, the standard deviation of the sampling intervals is pushed down to $\sim$2-3 ps (which is $< 1\%$ on the nominal sampling rate!). A script is included to make these plots: `timing/read_timing_json.py`

5 Analog Bandwidth

The analog bandwidth of the SURFv5 (with a LCFN-1200+ low pass filter installed) was measured using a 100 ps rise-and-fall time pulse. The width of this pulse was also about 100 ps. The SURFv5 waveforms were averaged and windowed. A measurement of the analog bandwidth compared to the impulse measured on a commercial 2 GHz oscilloscope is showsn in Fig. 6. The small signal bandwidth is approximately 1.1 GHz [2]

[2]It appears the SURFv5 bandwidth will be limited by the on-board filters not by the LAB4D input parasitics, so once the correct 1 GHz low pass is installed the high-end bandwidth will drop accordingly

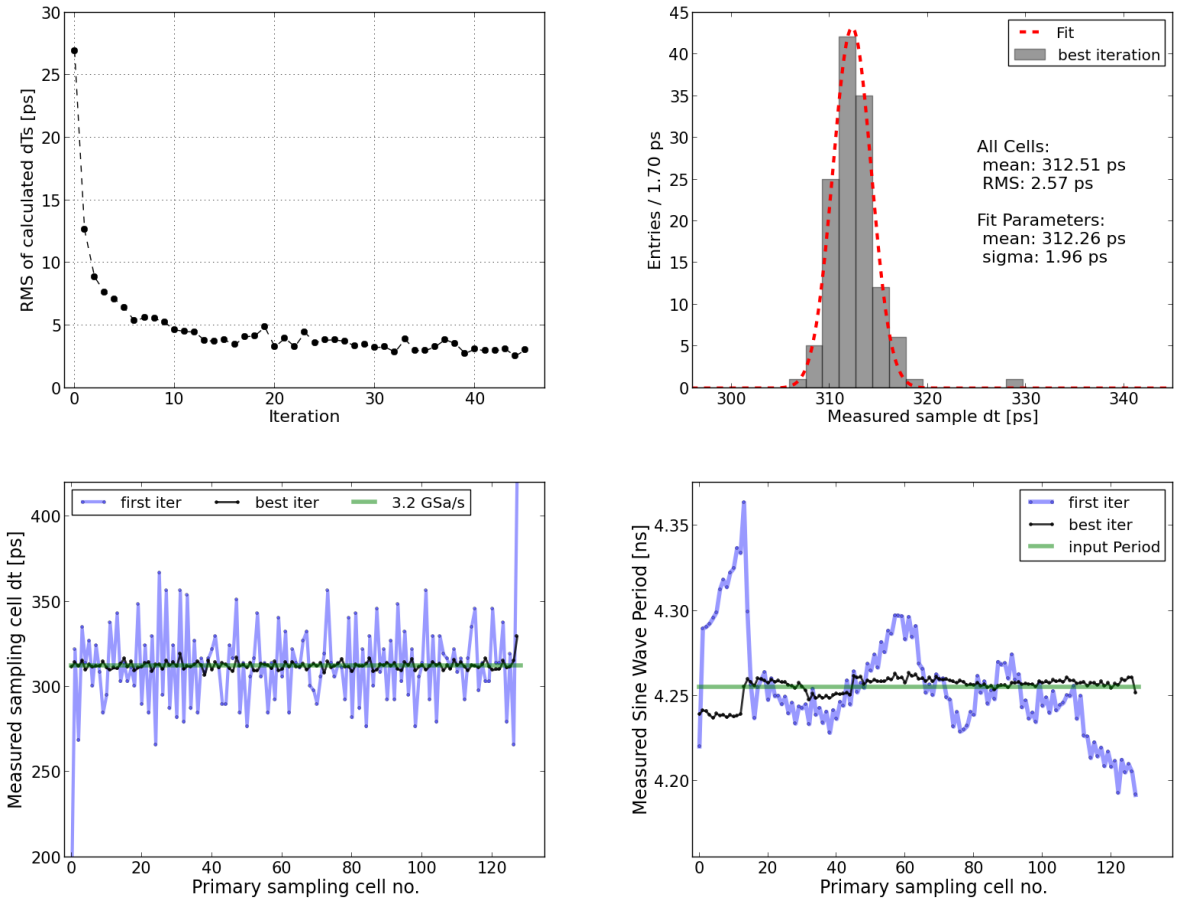


Figure 5: Timebase tuning results on Channel 0 of SURFv5 'CANOES' using 235.0 MHz sine wave. Upper left: RMS vs iteration number. Upper right: histogram of measured sampling intervals after tuning. Lower left: Sampling interval at each interval before and after tuning (i.e. the Differential non-linearity) Lower right: The measured periods using interpolated zero-crossings of before and after (i.e. getting at the integral non-linearity).

This should not be considered an exhaustive look at the frequency response of the SURFv5's or LAB4D chips.

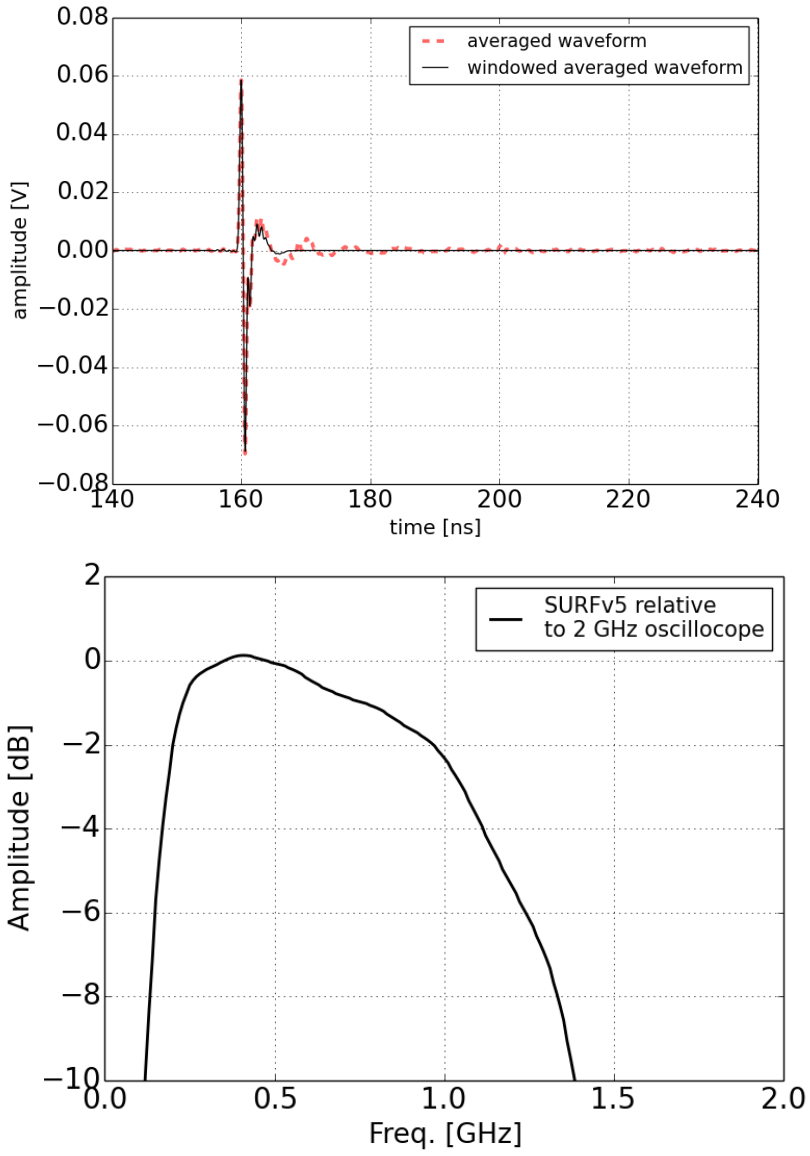


Figure 6: Averaged recorded waveform from a ∼100 ps rise-time impulse and resultant (small-signal) bandwidth measurement (Channel 0 of SURFv5 'CANOES')

A IPython Notebook: Basic Operations

In [107]:
```
%config InlineBackend.figure_format='svg'
import numpy
```

In [108]:
```
#load the surf module
import surf
#load the surf_data module
import surf_data

#create instances of these classes
dev=surf.Surf()
devData=surf_data.SurfData()
```

In [109]:
```
#check the pedestal level
#pedestal in mV = value * 2048 / 4096
dev.vped
```

Out[109]:
```
4000
```

In [110]:
```
#change the pedestal level
dev.set_vped(1800)
```

In [111]:
```
#re-check pedestal level
dev.vped
```

Out[111]:
```
1800
```

In [112]:
```
#the pedestal variable is set by reading the on-board DAC. You can access that directly like this
dev.i2c.read_dac()
```

```
Reading from MCP4728...
DAC channel A (RFP_VPED_0):  register is set to 0x9c4, EEPROM is set to 0x9c4
DAC channel B (RFP_VPED_1):  register is set to 0x578, EEPROM is set to 0x578
DAC channel C (RFP_VPED_2):  register is set to 0x578, EEPROM is set to 0x578
DAC channel D (VPED)      :  register is set to 0x708, EEPROM is set to 0x708
```

Out[112]:
```
1800
```

In [34]:
```
#re-take pedestal data
devData.pedestalRun()
```

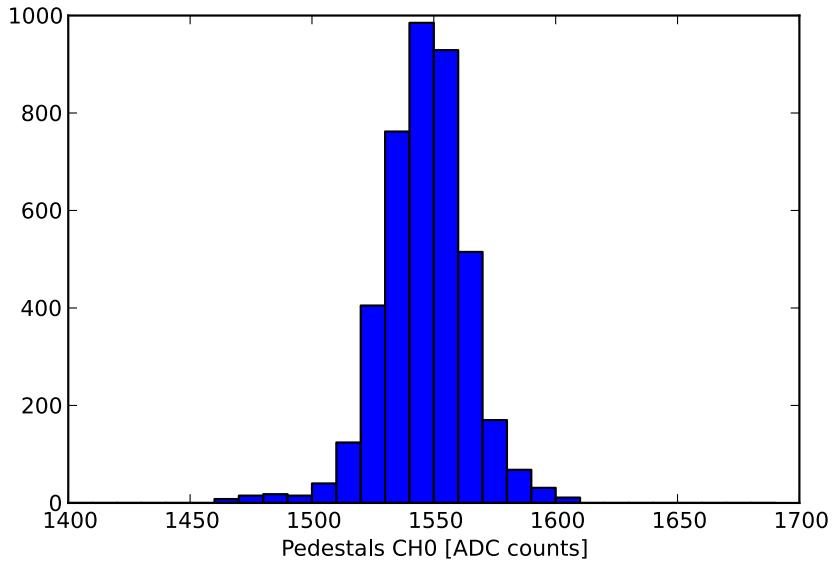
```
Saving pedestals for board CANOES...
```

Out[34]:
```
array([[1548, 1707, 1540, ..., 1625, 1559, 1623],
       [1555, 1724, 1546, ..., 1653, 1572, 1646],
       [1544, 1725, 1520, ..., 1627, 1571, 1643],
       ...,
       [1550, 1767, 1521, ..., 1639, 1599, 1546],
       [1529, 1757, 1549, ..., 1647, 1573, 1566],
       [1543, 1734, 1535, ..., 1602, 1573, 1467]])
```

In [113]:
```
#pedestals are saved to json file: calibrations/surf_calibrations.json
#they are also defined as a class variable:
devData.loadPed()

#histogram the pedestals on CANOES, channel=0
pylab.hist(devData.pedestals[:,0], bins=range(1400, 1700, 10))
pylab.xlabel('Pedestals CH0 [ADC counts]')
```

Out[113]:
```
<matplotlib.text.Text at 0x150ced0c>
```



In [43]:
```
#log some data to check the baseline
#take 50 events and don't save to a file, otherwise make save=True and specify a filename by filen
ame='xxxxxx'
data = devData.log(50, save=False)

#this automatically subtracts the pedestals from the raw data. To not subtract pedestals, specify
subtract_ped=False
#this automatically unwraps the data given the trigger position. To not unwrap, specify unwrap=Fal
se
```

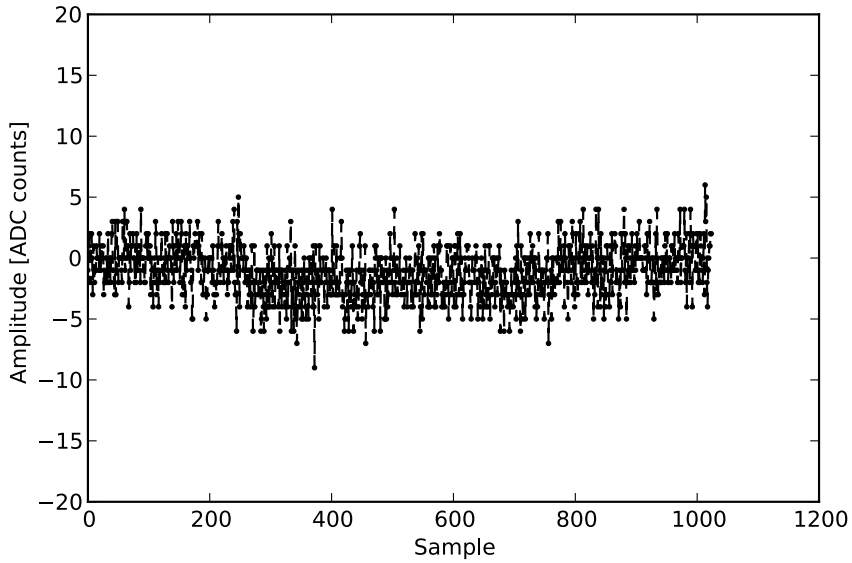
```
logging event...50
```

In [49]:
```
#the data is stored in a list of lists
print 'number of events', len(data), ' -- number of channels', len(data[0]), '-- number of samples
 per channel per event', len(data[0][0])
```

```
number of events 50  -- number of channels 12 -- number of samples per channel per event 1024
```

In [114]:
```
#plot event 5 on channel 0
pylab.plot(data[5][0][:], 'o--', color='black', ms=2)
pylab.xlabel('Sample')
pylab.ylabel('Amplitude [ADC counts]')
pylab.ylim([-20,20])
```

Out[114]: (-20, 20)

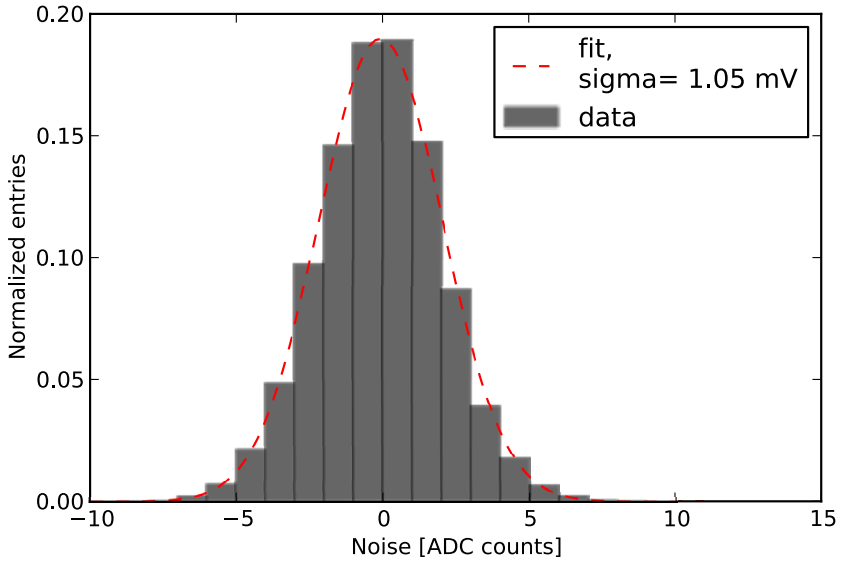


In [115]:
```
#look at the electronics noise on channel 0:
data=numpy.array(data)
pylab.hist(data[:,0,:].flatten(), bins=range(-10, 11, 1), normed=True, color='black', alpha=0.6, l
abel='data')
pylab.xlabel('Noise [ADC counts]')
pylab.ylabel('Normalized entries')

from scipy.stats import norm
mu, std=norm.fit(data[:,0,:].flatten())
x=numpy.linspace(-10, 11, 100)
pdf=norm.pdf(x, mu+0.5, std) #+0.5 to take into account bin edge -> bin center
pylab.plot(x, pdf, '--', color='red', label='fit,\nsigma= {:.2f} mV'.format(std/2)) #just about ex
actly 2 ADC counts per mV with current configuration

pylab.legend()
```

Out[115]: <matplotlib.legend.Legend at 0x14f834ec>



In [83]:
```
#do a pedestal scan, start/stop are pedstal DAC values
#saves to file pedscan.temp (can redefine filename, with filename='xxxxx' argument)

#this takes a few minutes to run
dac_values, lab_values = devData.pedestalScan(start=0, stop=4096, incr=100)
```

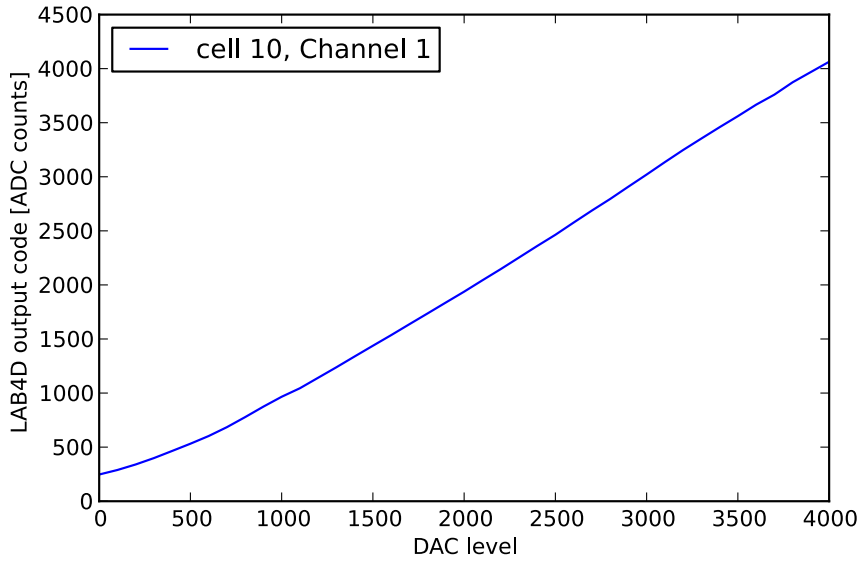
```
logging event...120
```

In [116]:
```
#dac_values = list of values of the pedestal DAC
#lab_values = list of array of pedestal values
print 'size of pedestal scan output array (number of scan points, number of cells per channel, num
ber of channels) =', numpy.array(lab_values).shape

#plot the tranfser curve for storage cell 10 on Channel 1:
pylab.plot(dac_values, numpy.array(lab_values)[:,10,1], label='cell 10, Channel 1')
pylab.xlabel('DAC level')
pylab.ylabel('LAB4D output code [ADC counts]')
pylab.legend(loc='upper left')
```

```
size of pedestal scan output array (number of scan points, number of cells per channel, number o
f channels) = (41, 4096, 12)
```

Out[116]:
```
<matplotlib.legend.Legend at 0x14a5c48c>
```



In [101]:
```
#code exists to generate a linear-interpolated LUT based on the output file from the pedestal scan
#however, it is not yet implemented in the readout or handled to a calibration file...TO DO!

#it can be generated using this:
surf_lut = devData.makeSurfLUT('calibrations/pedscan.temp', pedscan_start=0, pedscan_interval=100)
```

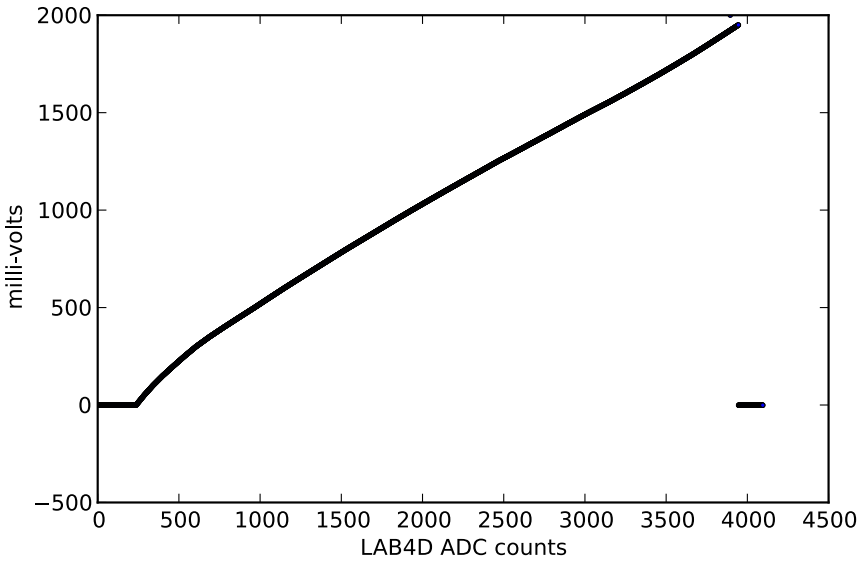
In [102]:
```
#this LUT is made by taking the mean response of each channel.
#It does not generate a LUT for each storage cell, which is probably the way to do this in the end
.
surf_lut.shape
```

Out[102]:
```
(12, 4096)
```

In [117]:
```
#plot the LUT for channel 1:
pylab.plot(surf_lut[1], 'o', ms=2)
pylab.xlabel('LAB4D ADC counts')
pylab.ylabel('milli-volts')

#note: values of -1 indicate the LUT is poorly defined (no mapping)
```

Out[117]: <matplotlib.text.Text at 0x14f8fe4c>



In [ ]:

B IPython Notebook: Tuning DLL feedback trim

In [49]:
```
%matplotlib inline
```

In [41]:
```
import timing.tune_dll_trim as tuneTrim
import surf
dev=surf.Surf()
```

In [42]:
```
#Tune the DLL feed back trim on channels 1, and 2 (signal plugged into lower SYNC input)
channels=[1,2]
#using 235 MHz sine wave
freq=235.e6
#sine wave amplitude is 300 ADC counts
amp=300
#run the scans:
scan_dict = tuneTrim.do(channels, freq, amp)
```

```
--------------
tuning LAB 1
--------------
difference: input freq - fitted freq = (kHz) 3574.50560894 trim 1160
difference: input freq - fitted freq = (kHz) 3270.63440554 trim 1170
difference: input freq - fitted freq = (kHz) 2934.17415815 trim 1180
difference: input freq - fitted freq = (kHz) 2624.66449163 trim 1190
difference: input freq - fitted freq = (kHz) 2353.97052626 trim 1200
difference: input freq - fitted freq = (kHz) 2060.55531639 trim 1210
difference: input freq - fitted freq = (kHz) 1826.74284741 trim 1220
difference: input freq - fitted freq = (kHz) 1574.80306888 trim 1230
difference: input freq - fitted freq = (kHz) 1316.71343222 trim 1240
difference: input freq - fitted freq = (kHz) 1099.39301319 trim 1250
difference: input freq - fitted freq = (kHz) 879.81695646 trim 1260
difference: input freq - fitted freq = (kHz) 669.871229278 trim 1270
difference: input freq - fitted freq = (kHz) 404.662354176 trim 1280
difference: input freq - fitted freq = (kHz) 232.963115856 trim 1290
difference: input freq - fitted freq = (kHz) 1.12193670201 trim 1300
difference: input freq - fitted freq = (kHz) -162.570496907 trim 1310
difference: input freq - fitted freq = (kHz) -323.182074147 trim 1320
difference: input freq - fitted freq = (kHz) -498.608411765 trim 1330
difference: input freq - fitted freq = (kHz) -654.690190063 trim 1340
difference: input freq - fitted freq = (kHz) -777.773189665 trim 1350
difference: input freq - fitted freq = (kHz) -954.694544789 trim 1360
difference: input freq - fitted freq = (kHz) -1118.59966445 trim 1370
difference: input freq - fitted freq = (kHz) -1235.22552152 trim 1380
difference: input freq - fitted freq = (kHz) -1343.4466097 trim 1390
difference: input freq - fitted freq = (kHz) -1455.68463342 trim 1400
difference: input freq - fitted freq = (kHz) -1559.15622442 trim 1410
lab: 1 , VtrimFB dac for 3.2 GSPS from fit: 1301.51593077  derivative, counts/kHz 0.0528342822115
setting VtrimFB to.. 1301
--------------
tuning LAB 2
--------------
difference: input freq - fitted freq = (kHz) 2873.75695344 trim 1160
difference: input freq - fitted freq = (kHz) 2514.17619835 trim 1170
difference: input freq - fitted freq = (kHz) 2181.84865133 trim 1180
difference: input freq - fitted freq = (kHz) 1824.30866525 trim 1190
difference: input freq - fitted freq = (kHz) 1537.98147527 trim 1200
difference: input freq - fitted freq = (kHz) 1252.32457857 trim 1210
difference: input freq - fitted freq = (kHz) 970.634682401 trim 1220
difference: input freq - fitted freq = (kHz) 673.859552052 trim 1230
difference: input freq - fitted freq = (kHz) 444.124474736 trim 1240
difference: input freq - fitted freq = (kHz) 186.596880604 trim 1250
difference: input freq - fitted freq = (kHz) -61.4818576448 trim 1260
difference: input freq - fitted freq = (kHz) -281.663108929 trim 1270
difference: input freq - fitted freq = (kHz) -444.039411026 trim 1280
difference: input freq - fitted freq = (kHz) -605.958254856 trim 1290
difference: input freq - fitted freq = (kHz) -789.066577433 trim 1300
difference: input freq - fitted freq = (kHz) -959.104970967 trim 1310
difference: input freq - fitted freq = (kHz) -1130.3092458 trim 1320
difference: input freq - fitted freq = (kHz) -1298.97390495 trim 1330
difference: input freq - fitted freq = (kHz) -1437.86201966 trim 1340
difference: input freq - fitted freq = (kHz) -1576.86205504 trim 1350
difference: input freq - fitted freq = (kHz) -1706.17000518 trim 1360
difference: input freq - fitted freq = (kHz) -1804.15107841 trim 1370
difference: input freq - fitted freq = (kHz) -1956.32480676 trim 1380
difference: input freq - fitted freq = (kHz) -2091.14232609 trim 1390
difference: input freq - fitted freq = (kHz) -2194.0652695 trim 1400
difference: input freq - fitted freq = (kHz) -2316.76948501 trim 1410
lab: 2 , VtrimFB dac for 3.2 GSPS from fit: 1258.37333123  derivative, counts/kHz 0.0460514695459
setting VtrimFB to.. 1258
```

In [43]:
```
#show the fitted register values
scan_dict
```

Out[43]:
```
{'1': 1301, '2': 1258}
```

In [44]:
```
#save to cal file
tuneTrim.save(scan_dict, dev.dna())
```

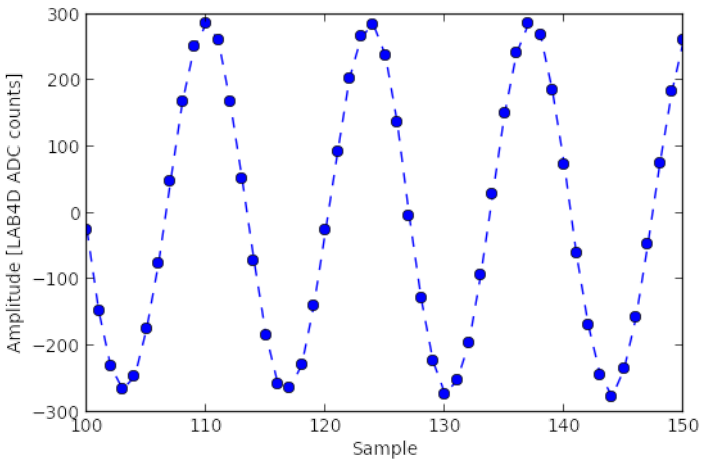
```
1 1301
1
2 1258
2
reading back cal file:
vtrimfb {u'1': 1301, u'0': 1325, u'3': 1268, u'2': 1258, u'5': 1391, u'4': 1236}
```

In [45]:
```
#to load the feedback trim values for the board use this function:
#it assigns the default value defined in utils/surf_constants.py if not defined in the cal file
tuneTrim.load(dev.dna())
```

Out[45]:
```
[1325, 1301, 1258, 1268, 1236, 1391, 1350, 1350, 1350, 1350, 1350, 1350]
```

In [58]:
```
#examine at the DLL wraparound seam sample 127->128
import surf_data
import matplotlib
import matplotlib.pyplot as plt
devData=surf_data.SurfData()
plt.plot(devData.log(1, save=False)[0][0], 'o--')
plt.xlim([100,150])
plt.xlabel('Sample')
plt.ylabel('Amplitude [LAB4D ADC counts]')
```

Out[58]:
```
<matplotlib.text.Text at 0xab6574c>
```



In [ ]: