CPE 350/450

User Manual

Cal Poly Computer Engineering Capstone

Joseph Alfredo
Michael Le
Aaron Nguyen
Kevin Yang

Fall 2017 – Winter 2018

Contents

List of Figures

Listings

1 System Overview

The overall flow of the system can be seen below.

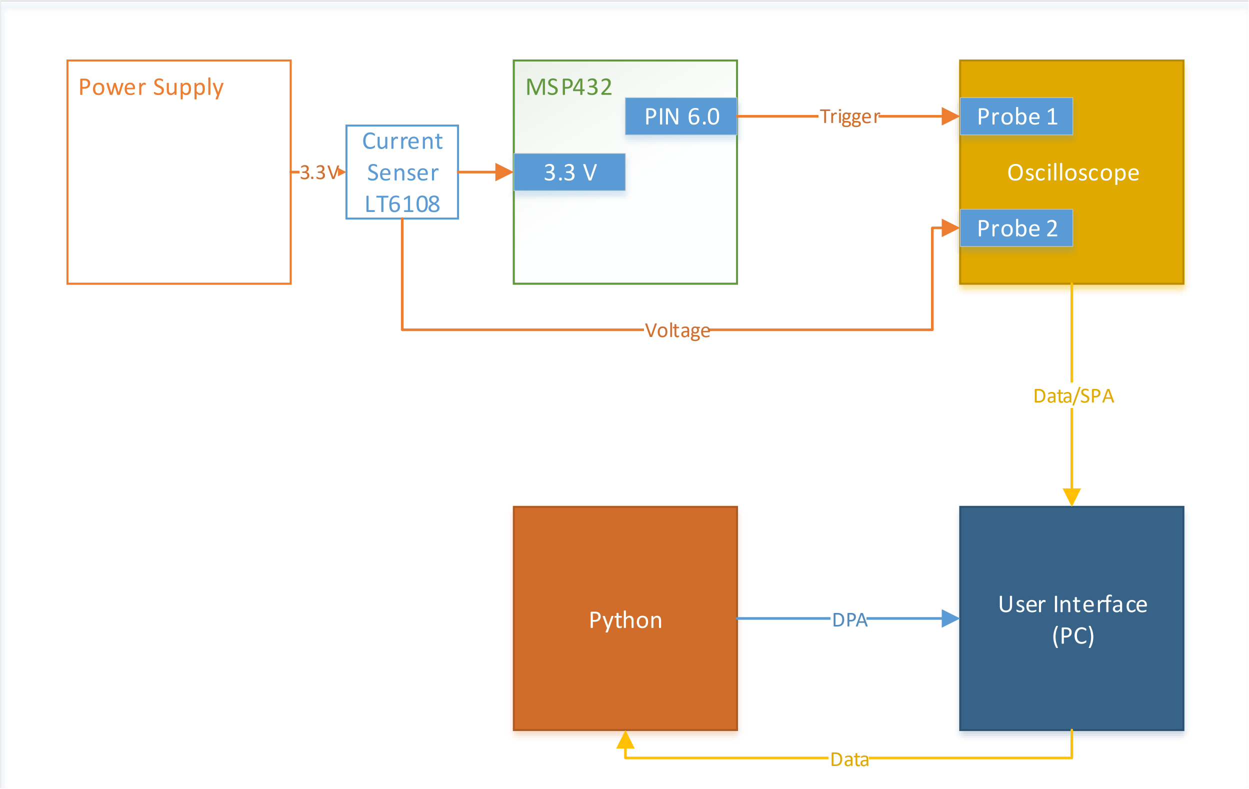

Figure 1: System Flowchart

To measure the power usage of the MSP432, a sense resistor, R_{sense} is placed in series in between the power supply and the board itself. The current drawn by the MSP432 causes a voltage drop across R_{sense}. This voltage drop is inputted into the differential inputs of the LT6108, which is a current sense IC. Two resistors, R_{in} and R_{out}, control the resulting output voltage seen at the output of the LT6108 IC; the ratio $\frac{R_{out}}{R_{out}}$ is the voltage gain of the circuit. This configuration amplifies R_{sense}'s voltage drop, V_{sense}.

A bypass capacitor is utilized across the LT6108's power input and ground to suppress noise caused by noise coupling in through the power rail.

To properly measure the MSP432's power usage, the oscilloscope must oversample the data. Ideally, a sampling rate of ten times the clock speed of the MSP432 should be utilized. Since the MSP432 was set to run at $375kHz$, the oscilloscope should therefore sample at approximately $3.75MHz$.

The power is measured by connecting a scope to the output of the LT6108 IC and to a trigger pin. Note that LEDs should not be used. Using LEDs for visual confirmation causes the MSP432 to "charge," which will rail the output voltage of the LT6108.

2 Setup

2.1 Hardware Setup

Remove all pins from the MSP432 except for the ground and JTAG pins. Set up the circuit seen below. Note that R_{sense}'s connections are tapped out from the MSP432's J101 isolation block.

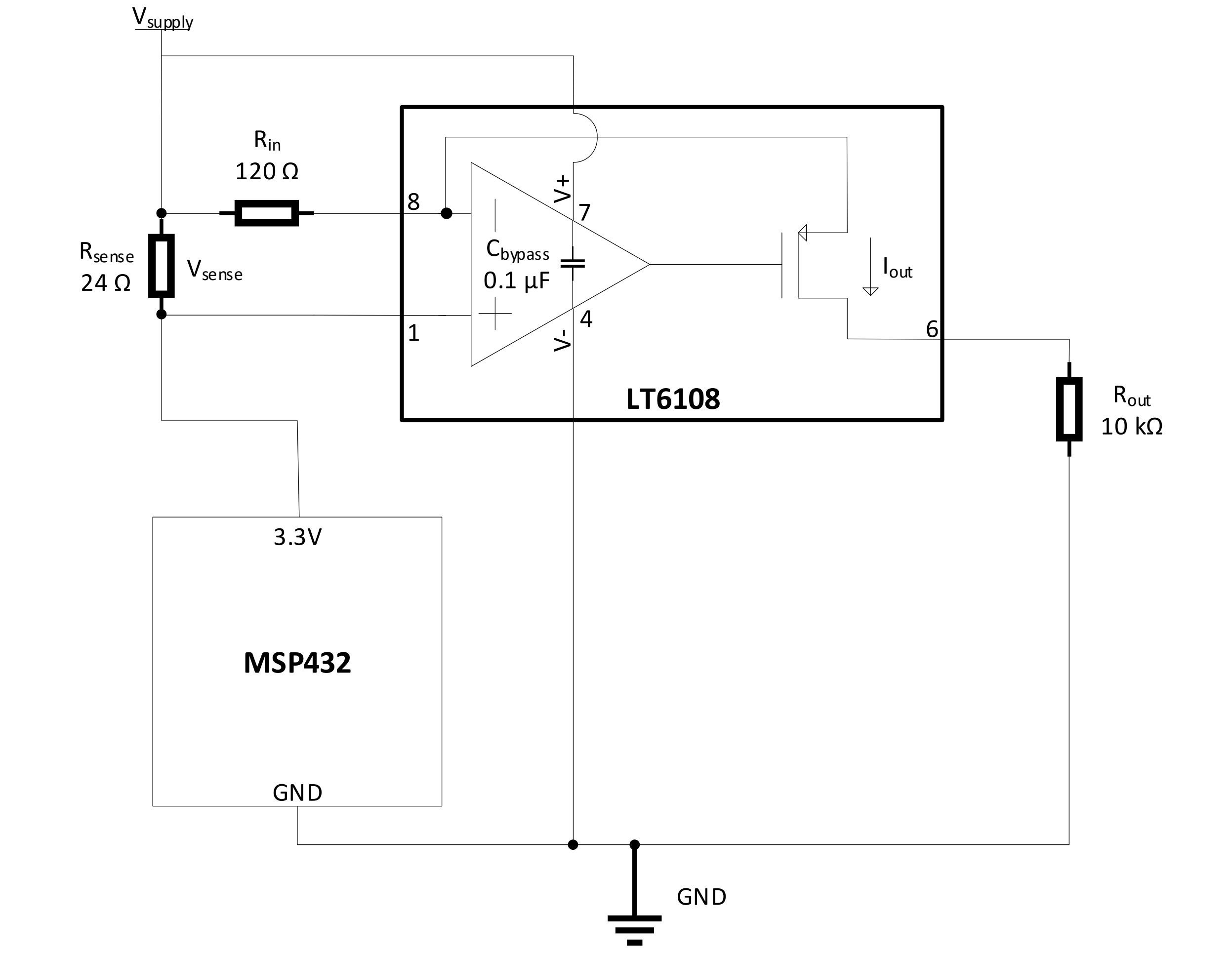

Figure 2: Wiring Diagram for MSP432

2.2 Software Setup

2.2.1 Model Creation

Configure the MSP432 to use the binary keys of a leading 1 with all 0 bits or all 1 bits to create the 0 or 1 bit models, respectively. Capture a large amount of these waveforms with DAQ.py and using bin.py, the 0 and 1 bit data will be put into two separate .csv files. With those files, an average can be taken of all the columns to produce a model for that bit.

2.2.2 Single-data Multiple-exponent Attack

With the models created, capture more data of an unknown key to compare the models to with DAQ.py once again. Average out the waveform once again to create a model for the entire exponent. This may not show much, but use an error function, such as mean-squared error, for every point of the exponent against each of the models. This should show where the areas of each model aligns with the exponent. We did not get very far in this process, so many more improvements could be made.

3 Functionality

This system contains four components that work together to enable power analysis. The components are a current sensor, microcontroller, data acquisition program, and power analysis.

3.1 Current Sensing

To use the current sensor, ensure it is configured as in Figure 1 of the system overview's hardware setup. Then connect the V_{sense} node to the MSP432's 3.3V pin. When the MSP432 runs, there should be about a gain of 87.

3.2 Microcontroller Encryption

The MSP432 microcontroller runs the RSA encryption algorithm. The plain text and encryption key are hard-coded on lines 28 and 29 respectively in the *main.c* file. As a result, these two inputs need to be manually set in the *main.c* to select the plain text and encryption key.

To run the RSA encryption, open Texas Instrument's (TI) Code Composer Studio (CCS) integrated development environment which can be downloaded and installed from the following URL: `http://processors.wiki.ti.com/index.php/Download_CCS`.

In CCS, the relevant source and header files are placed together in a project folder which the encryption program can then be compiled and transmitted to the MSP432. To do so, press the debug button on the top toolbar. Then, press the run button after compiling the program. The RSA encryption should now indefinitely loop.

To stop the encryption process, press the stop button at the toolbar. For further help using CCS, refer to TI's wiki page: `http://processors.wiki.ti.com/index.php/Main_Page`.

3.3 Data Acquisition

To use the data acquisition program, connect a computer to a Keysight InfiniiVision Oscilloscope. Then use a scope probe to connect the oscilloscope to the V_{sense} node.

Once the oscilloscope is configured, run the *plot-gui.py* using Python3. A graphical user interface should launch, where scope capture can be configured, with features including the filename to export the waveform data.

3.4 Template Power Analysis

Using a spreadsheet program like Microsoft excel, open the .csv files saved from the data acquisition program and the software setup in the earlier section.

$$\frac{1}{n}\sum_{t=1}^{n} e_t^2$$

Then using the spreadsheet program, the mean-squared error (MSE) can be calculated where e is the difference between points in the model waveform and the corresponding averaged captured waveform. The MSE is calculated multiple times by shifting the model waveform across the captured waveform. An MSE graph is then created with each point representing the MSE for a particular shift.

Since the MSP432 encrypts sequentially from the least significant bit (LSB) to the most significant bit (MSB), at each local minimum of the MSE graph represents where the model waveform matches the captured waveform. Repeating the templating, we can solve for the encryption key from LSB to MSB.

4 Troubleshooting

4.1 Unexpected waveform is captured

Make sure Trigger is connected to the determined pin (6.0 in this case) and power distribution on the other

4.2 Unable to find library

Install the required third party library for python using ”pip install”. These are required third party library, numpy, pandas, visa, PyQt5 and matplotlib.

4.3 Unable to detect oscilloscope

Make sure the oscilloscopes model are supported by the library, py visa. Also make sure the proper scope visa address is entered correctly in the DAQ.py file.

4.4 CSV file is not exported

The file name field does not required .csv extension.

A Code

A.1 C

Listing 1: `main.c`

```c
#include <stdio.h>
#include <stdlib.h>
#include "msp.h"
#include "imath.h" // https://github.com/creachadair/imath
#define BASE    16

void printNum(mpz_t num);

void main(void) {
    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;     // stop watchdog timer

    printf("Init\n");
    // set_DCO(freq);
    set_HFXT();
    printf("HFXT enabled\n");
    configure_unused_ports();
    printf("Disabled unused ports\n");
    /* imath stuff */
    mpz_t input, e, n, d, encrypt, decrypt;

    /* Initialize a new zero-valued mpz_t structure */
    mp_int_init(&n);
    mp_int_init(&d);
    mp_int_init(&encrypt);
    mp_int_init(&decrypt);

    /* Initialize a new mpz_t with a small integer value */
    mp_int_init_value(&input, 123456);
    mp_int_init_value(&e, 43690); /* 65537 = b10000000000000001 */

    /* N and d for different sizes
     * Works:
     * 64   N: 8ff3d1240677272eb239818d5a080b97
     * 64   d: 1b626019fa5416d3007fdf09c216b9
     * 128  N: 99766c280d75d2e1dcb3f5cc878e61a858fa60c6c2de2befa63c202a8b0fcf83
     * 128  d: a36ffa1d012c0c56e2c1997458a3cc5f74532248cc36759d346ee18ef80179d
     * 256  N: 8e7075a2a26c7dee7b67b63e62cad1a8d141dc18e4d6315e998147b56c2fdb788
     *         b49e73d90e16e4cc49f580825076054e6ee08e69fcad21491c2885522daf2a9
     * 256  d: a97de87cbe099b504c47fcf3ff5a9860e9a014e70a665618db509aa3a738cb4db
     *         e2123fde5ba07c081b0ebf8fc37e98b326ac61f7e1655c71cc18ead2ecec01
     * 512  N: 96072e1e996ae4ff4ced88b87ed2fd809d3759d4d674c7a64f08f04c456a6a2c2
     *         57b791cf9f8832510cc386002a8525aed665bce943ab548d760d8a7214ed0fadf
     *         334c5ee7f01f3288f017291faf5cb434a0d423475f4143b8f513b67df15e05ea1
     *         feee3e08f14463f0fc37869c0e1897f240c9d603bda544654785460adad51
     * 512  d: 1723d1def9da452a447701a6b536fd4f2450987c2dbcb2db89a6af0b908c56866
     *         b1c95845ce0d1f77a55095ff3a0fb1cd7af793b0e5d54bf40ab09179419583d90
     *         b5a07030a9377564335b447d7fe7f767efddcd24940482e0a5a1cece8f9e13d1f
     *         6b3bca4ac296f21d9a83b20e7fa51e97c3c0d92fa2c54746faafe4766593d
     * 1024 N: 880305c7a47899cd394ef9e802214ad996478b85077952b31f4250332d8b67518
     *         9bd398c21795f8f72dae11084d69ebe0bca34810f395d857a8fc49fc389e2d446
     *         7ab9404f019d56f246f4eab9e05acf42153ab6f1422758e6c92787324acfb0fc5
     *         b4d49c4cb5be2ed3e72e8511e86b132ef2db4c65fb7d5f367617752c171493756
     *         9add661be61f3cc37ffcb88b79640f4d5944d996a0f38e79ff9cb944d930e04ac
     *         0b13240f11e1fef842ac63c7713061d6e54ac8c5272da2efbe73c4f62e1af3df4
     *         b8016c1fc4bedb9c7ecf6bdd91f78c81b3882b4c976aadc356b68261311ec10e0
     *         58ab9795a3e2e545017fdd2966dec1e7fa7c559e2dc32d0b591734d65
     * 1024 d: 1cec079e9c6ac8c9cb15f02e55c59e95064fd06b495b932a63cb46229bdcb8eba
     *         dce7f1e3d4002020efa5c4196fdcc63bd3e124c1f60a3726ecd839235926c9997
     *         2321a17b2b6cb9c06b3649739d31b240eb22c1242c5d119a81cbd603ebc49e6e0
     *         b3c342394dac5368dc10185be6805e63ed6094ae5afc1df306c99630f9f77128c
     *         878f82ca0c6c410003aacfc6489d4582ce1c529af0f3cd9f9de6abc70391d37d4
     *          74e73da0a6c50d6a89540cf10d5a0d1f7c7ac6300ce6eb241fbb760bd74ed0680
     *          fe152f97e8b7dc351ee2e469e382461959e33d576fe64a3574414283ab37052db
     *          0ea1dc19dad14abb2eab2c6b5d768f4ace9d104c263c30576eb49d41
     */

    /* Initialize a new mpz_t with a string value in base BASE */
    mp_int_read_string(&n, BASE, "8ff3d1240677272eb239818d5a080b97");
    mp_int_read_string(&d, BASE, "1b626019fa5416d3007fdf09c216b9");

    /* Configure trigger pin */
    P6->SEL1 &= ~BIT0;
    P6->SEL0 &= ~BIT0;
    P6->DIR |= BIT0;

    while(1) {
        /* Trigger high, encrypt, trigger low, delay */
        P6->OUT |= BIT0;
        mp_int_exptmod(&input, &e, &n, &encrypt);
        P6->OUT &= ~BIT0;
        delayMs(250, 38);
    }
}

printNum(mpz_t num) {
    int len = mp_int_string_len(&num, BASE);
    char *buf = calloc(len, sizeof(*buf));
    mp_int_to_string(&num, BASE, buf, len);

    printf("Num: %s\n", buf);

    free(buf);
}
```

Listing 2: `delay.h`

```
#ifndef DELAY_H_
#define DELAY_H_
#include "msp.h"

void set_HFXT() // decrease this and decrease key size to allow for more accurate scope capture (maybe)
                //also should make sure LED doesnt pull from 3.3v line if tapping out
{
    /* Enable LDO high-power mode (3V, not DC-DC cause DC-DC has switching noise */
    /* Step 1: Transition to VCORE Level 1: AM0_LDO --> AM1_LDO */
    while ((PCM->CTL1 & PCM_CTL1_PMR_BUSY));
    PCM->CTL0 = PCM_CTL0_KEY_VAL | PCM_CTL0_AMR__AM_LDO_VCORE1;
    //PCM->CTL0 = PCM_CTL0_KEY_VAL | PCM_CTL0_AMR__AM_DCDC_VCORE1; // this is tapped out. doesnt work
    while ((PCM->CTL1 & PCM_CTL1_PMR_BUSY));

    /* Step 2: Configure Flash wait-state to 1 for both banks 0 & 1 */
    FLCTL->BANK0_RDCTL = (FLCTL->BANK0_RDCTL & ~(FLCTL_BANK0_RDCTL_WAIT_MASK)) |
            FLCTL_BANK0_RDCTL_WAIT_1;
    FLCTL->BANK1_RDCTL  = (FLCTL->BANK0_RDCTL & ~(FLCTL_BANK1_RDCTL_WAIT_MASK)) |
            FLCTL_BANK1_RDCTL_WAIT_1;

    /* Configure pins J.2/3 for HFXT function (HFXTIN, HFXTOUT) */
    PJ->SEL0 |= BIT2 | BIT3;
    PJ->SEL1 &= ~(BIT2 | BIT3);

    /* defines arent representative of the actual frequency but don't really care
     * since we'll be using this at 24 or 48 MHz.
     */
    CS->KEY = CS_KEY_VAL;
    // CS->CTL = 0;

    /* CS_CTL2_HFXTDRIVE required for HFTX higher than HFTXFREQ */
```

```c
    CS->CTL2 = CS_CTL2_HFXTFREQ_6 | CS_CTL2_HFXT_EN | CS_CTL2_HFXTDRIVE;

    while(CS->IFG & CS_IFG_HFXTIFG)
        CS->CLRIFG |= CS_CLRIFG_CLR_HFXTIFG;

    CS->CTL1 = CS_CTL1_SELM__HFXTCLK | CS_CTL1_DIVM__128; /* set MCLK as output. also need to output to pin to check freq. no division */
    CS->KEY = 0;
}

void configure_unused_ports()
{
    /* initialize all pins so power isn't wasted on possible floating pins.
     * sets all pins to output mode. output bit is don't care but initialized to 0
     * see section 12.3.2 revision H (Configuration of Unused Ports) for more information */
    P1->DIR = 0x00; /* set all as input */
    P1->REN = 0xFF; /* enable resistor pull up / pull down */
    P1->OUT = 0x00; /* pull down to ground */
    P2->DIR = 0x00; /* set all as input */
    P2->REN = 0xFF; /* enable resistor pull up / pull down */
    P2->OUT = 0x00; /* pull down to ground */
    P3->DIR = 0x00; /* set all as input */
    P3->REN = 0xFF; /* enable resistor pull up / pull down */
    P3->OUT = 0x00; /* pull down to ground */
    P4->DIR = 0x00; /* set all as input */
    P4->REN = 0xFF; /* enable resistor pull up / pull down */
    P4->OUT = 0x00; /* pull down to ground */
    P5->DIR = 0x00; /* set all as input */
    P5->REN = 0xFF; /* enable resistor pull up / pull down */
    P5->OUT = 0x00; /* pull down to ground */
    P6->DIR = 0x00; /* set all as input */
    P6->REN = 0xFF; /* enable resistor pull up / pull down */
    P6->OUT = 0x00; /* pull down to ground */
    P7->DIR = 0x00; /* set all as input */
    P7->REN = 0xFF; /* enable resistor pull up / pull down */
    P7->OUT = 0x00; /* pull down to ground */
    P8->DIR = 0x00; /* set all as input */
    P8->REN = 0xFF; /* enable resistor pull up / pull down */
    P8->OUT = 0x00; /* pull down to ground */
    P9->DIR = 0x00; /* set all as input */
    P9->REN = 0xFF; /* enable resistor pull up / pull down */
    P9->OUT = 0x00; /* pull down to ground */
    P10->DIR = 0x00; /* set all as input */
    P10->REN = 0xFF; /* enable resistor pull up / pull down */
    P10->OUT = 0x00; /* pull down to ground */
}

/* Arbitrary busy delay function recycled from previous projects */
void delayMs(int n,int f) {
    int i, j;

    for (j = 0; j < n; j++)
        for (i = f; i > 0; i--);
}

#endif /* DELAY_H_ */
```

A.2 Python

Listing 3: `align.py`

```python
import numpy as np
import pandas as pd
import math

from scipy import signal
from sys import argv
from matplotlib import pyplot as plt
```

```python

# Only works for similar waveforms and only approximates
def libFun(wave1, wave2):
  mid = len(wave1)-1
  cor = np.correlate(wave1, wave2, 'full')
  print(cor, signal.correlate(wave1, wave2, 'full'))
  return np.argmax(cor) - mid

# Gets more accurate alignment but much more time consuming
def errorFun(wave1, wave2):
  wave_len = len(wave1)

  min_mse = math.inf
  min_shift = None

  for shift in range(-wave_len + 1, wave_len):
    # calculating upper and lower bound of shifted waveform
    lower_end = shift
    upper_end = lower_end + wave_len - 1

    if (upper_end >= 0 and lower_end < wave_len): # shifted wave contain points in base waveform's domain
      total_error = 0
      overlap = 0

      # calculating the total error for the shifted waveform
      if (upper_end < wave_len): # shifted wave inbound while its lower end out of bounds
        overlap = upper_end + 1
        for i in range(0, upper_end + 1):
          j = i - shift
          total_error += pow((wave2[j] - wave1[i]), 2)
      else: # shifted wave inbound while its upper end out of bounds
        overlap = wave_len - lower_end + 1
        for i in range(lower_end, wave_len):
          j = i - shift
          total_error += pow((wave2[j] - wave1[i]), 2)

      # calculating the mean of the SSE
      mse = total_error / overlap

      # tracking for min error
      if (mse < min_mse):
        min_mse = mse
        min_shift = shift

  return min_shift

# Align two waveforms
def main(argv):
  wave_data = pd.read_csv(argv[1])
  x = wave_data["time"]
  y1 = wave_data["base sinewave"]
  y2 = wave_data["base cosine"]

  dx = np.mean(np.diff(x))
  plt.plot(x, y1, x, y2)
  shift = libFun(y1, y2) * dx
  plt.plot(x, y1, x + shift, y2)
  plt.show()

if __name__=="__main__":
  main(argv)
```

Listing 4: `bin.py`

```python
import sys
import struct
import pandas as pd
import numpy as np
import csv

BIT = '1'
KEY = '1'
TRIG_VOLT = 3         # Cutoff voltage for a trigger high
BIT_1_THRESH = 100    # Cutoff threshold for number of points in a 0 bit model

# Converts waveform captured from each trigger of the encryption cycle
# into their own .csv to be later converted into a model
def main():
   if KEY == '1':
      result_csv = open('outputKey.csv', 'w')
   else:
      result_csv = open('outputBit' + BIT + '.csv', 'w')
      # Data for the delays are captured as well
      delay_csv = open('delayBit' + BIT + '.csv', 'w')

   if KEY == '1':
      # Key model
      for counter in range(31, 61):
         filename = 'Key/scope_' + str(counter) + '.csv'
         add_bit_data_to_csv(filename, result_csv, None)
   elif BIT == '0':
      # Bit 0 model
      for counter in range(41, 91):
         filename = 'Bit' + BIT + '/scope_' + str(counter) + '.csv'
         add_bit_data_to_csv(filename, result_csv, delay_csv)
   else:
      # Bit 1 model
      for counter in range(0, 41):
         filename = 'Bit' + BIT + '/scope_' + str(counter) + '.csv'
         add_bit_data_to_csv(filename, result_csv, delay_csv)

   result_csv.close()
   delay_csv.close()

def add_bit_data_to_csv(source_file, result_csv, delay_csv):
   src_csv = pd.read_csv(source_file)
   wr_result = csv.writer(result_csv, delimiter=',')
   if delay_csv:
   wr_delay = csv.writer(delay_csv, delimiter=',')

   src_vout = list(src_csv[”1”])[1:]
   src_trigger = list(src_csv[”2”])[1:]
   src_vout = [float(i) for i in src_vout]
   src_trigger = [float(i) for i in src_trigger]

   bit_data = []
   delay_data = []
   is_triggered = False
   for vout_data, trigger_data in zip(src_vout, src_trigger):
      # Start of a trigger
      if not is_triggered and trigger_data > TRIG_VOLT:
         is_triggered = True
         if KEY != '1':
            wr_delay.writerows([delay_data])
            delay_data = []
      # End of a trigger
      elif is_triggered and trigger_data < TRIG_VOLT:
         is_triggered = False
         if BIT == '1' or KEY == '1':
               if len(bit_data) > BIT_1_THRESH:
                  wr_result.writerows([bit_data])
         else:
               if len(bit_data) < BIT_1_THRESH:
                  wr_result.writerows([bit_data])
         bit_data = []

      # Trigger high means data for the bit
      if is_triggered:
         bit_data.append(vout_data)
      # Trigger low means data for the delay
      else:
         delay_data.append(vout_data)

if __name__== ”__main__”:
    main()
```

Listing 5: `DAQ.py`

```
import numpy as np
import pandas as pd
import visa, time
import sys
import struct
import matplotlib.pyplot as plt

SCOPE_VISA_ADDR = ”USB0::0x0957::0x1797::MY55460257::0::INSTR”

GLOBAL_TOUT = 10000 # 1000 = 1 second
TIME_TO_TRIGGER = 10
TIME_BTWN_TRIGGERS = 0.025
MHZ = 1e6

##############################################################################
## main
##############################################################################

def main():
    osc_daq = init_osc()

    results = pd.DataFrame()
    # osc_daq.write(”:RUN”)
    # osc_daq.write(”:SINGLE”) # acquires one waveform (pg. 790)

    for _ in range(2):
        wave_data = take_waveform(osc_daq)
        results = results.append(wave_data)

    first_waveform = results.iloc[0]
    sec_waveform = results.iloc[1]
    first_waveform.plot(kind=”line”)
    sec_waveform.plot(kind=”line”)

    plt.show()

##############################################################################
## Helper Functions
##############################################################################
def take_waveform(scope, to_trigger):
  scope.query(”:STOP;*CLS;*OPC?”)

  if to_trigger:
      scope.write(”:SINGLE”)

      i = 0
      while i <= GLOBAL_TOUT:
          value = scope.query(”:OPERegister:CONDition?”)
          time.sleep(1)
          i += 1000

          if not (int(value) & 8):
              break

      # trigger not found
      if i > GLOBAL_TOUT:
          print("Trigger not found")
          return
  else:
      scope.write(":RUN")
      scope.write(":AUTOSCALE")

  scope.write(":WAVeform:POINts:MODE RAW")
  scope.write(":WAVeform:POINts 7680")
  scope.write(":WAVeform:SOURce CHANnel1")
  scope.write(":WAVeform:FORMat ASCII")

  sData = scope.query(":WAVeform:DATA?")
  wave_results = format_wave_data(sData)
  df = pd.DataFrame([wave_results])
  return df

def init_osc():
    rm = visa.ResourceManager()
    resources = rm.list_resources()
    print("Resources: ", resources)

    try:
        device_addr = get_device_addr("USB0", resources) ## oscilloscope first address
        scope = rm.open_resource(device_addr)
    except Exception:
        print("Unable to connect to oscilloscope at " + str(SCOPE_VISA_ADDR) + ". Aborting script.\n")
        sys.exit()

    print(scope.query("*IDN?")) # what are you?
    print(scope.resource_info) # oscilloscope information

    ## Set Global Timeout
    ## This can be used wherever, but local timeouts are used for Arming, Triggering, and Finishing the acquisition... Thus it mostly handles IO timeouts
    scope.timeout = GLOBAL_TOUT

    trigger_mode_on(scope)

    return scope

def trigger_mode_on(scope):
    ## Clear the instrument bus
    scope.clear()

    ## Setup Triggering
    scope.write(":TRIGGER:MODE EDGE")
    scope.write(":TRIGger:EDGE:LEVel 2")

    ## Clear all registers and errors
    ## Always stop scope when making any changes.
    scope.query(":STOP;*CLS;*OPC?")

def do_command(command, scope, hide_params=False):
    if hide_params:
        (header, data) = string.split(command, " ", 1)

    scope.write("%s\n" % command)
    if hide_params:
        check_instrument_errors(header)
    else:
        check_instrument_errors(command)

# =========================================================
# Send a query, check for errors, return string:
# =========================================================
def do_query_string(query, InfiniiVision):
   result = InfiniiVision.query("%s\n" % query)
   check_instrument_errors(query, InfiniiVision)
   return result

# =========================================================
# Send a query, check for errors, return values:
# =========================================================
def do_query_values(query, InfiniiVision):
   results = InfiniiVision.ask_for_values("%s\n" % query)
   check_instrument_errors(query, InfiniiVision)
   return results

# =========================================================
# Check for instrument errors:
# =========================================================
def check_instrument_errors(command, InfiniiVision):
   while True:
      error_string = InfiniiVision.query(":SYSTem:ERRor?\n")
      if error_string: # If there is an error string value.
         if error_string.find("+0,", 0, 3) == -1: # Not "No error".
            print("ERROR: %s, command: '%s'" % (error_string, command))
            print("Exited because of error.")
            sys.exit(1)
         else: # "No error"
            break
      else: # :SYSTem:ERRor? should always return string.
         print ("ERROR: :SYSTem:ERRor? returned nothing, command: '%s'" % command)
         print ("Exited because of error.")
         sys.exit(1)

# =========================================================
# Returns data from definite-length block.
# =========================================================
def format_wave_data(sBlock):

   # First character should be "#".
   pound = sBlock[0:1]
   if pound != "#":
      print("PROBLEM: Invalid binary block format, pound char is '%s'." % pound)
      print("Exited because of problem.")
      sys.exit(1)

   # Second character is number of following digits for length value.
   digits = sBlock[1]

   # Get the data out of the block and return it.
   sData = sBlock[int(digits) + 2:]
   list_data = sData.split(",")
   list_data = [float(i) for i in list_data]
   return list_data

def get_device_addr(port_type, resource_list):
    device_addr = ""
    for device in resource_list:
        if port_type in device:
            device_addr = device
            return device_addr

if __name__== "__main__":
    main()
```

Listing 6: IV_2_Robust_Sync_Methods.py

```
   # -*- coding: utf-8 -*-

## DO NOT CHANGE ABOVE LINE

# Python for Test and Measurement
```

```
#
# Requires VISA installed on Control PC
# 'keysight.com/find/iosuite'
# Requires PyVisa to use VISA in Python
# 'http://PyVisa.sourceforge.net/PyVisa/'

## Keysight IO Libraries 17.1.19xxx
## Anaconda Python 2.7.7 64 bit
## PyVisa 1.6.3
## Windows 7 Enterprise, 64 bit

##""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
## Copyright    2015 Keysight Technologies Inc. All rights reserved.
##
## You have a royalty-free right to use, modify, reproduce and distribute this
## example files (and/or any modified version) in any way you find useful, provided
## that you agree that Keysight has no warranty, obligations or liability for any
## Sample Application Files.
##
##""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""

##############################################################################
## Import Python modules
##############################################################################
## Import python modules - Not all of these are used in this program; provided for reference
import sys
import visa # PyVisa info @ http://PyVisa.readthedocs.io/en/stable/
import time
import struct
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

##############################################################################
## Intro, general comments, and instructions
##############################################################################

## This example program is provided as is and without support. Keysight is not responsible for modifications.
## Standard Python style is not followed to allow for easier reading by non-Python programmers.

## Keysight IO Libraries 17.1.19xxx was used.
## Anaconda Python 2.7.7 64 bit is used - 64 bit is strongly recommended for all scope applications that can create lots of data.
## PyVisa 1.8 is used
## Windows 7 Enterprise, 64 bit (has implications for time.clock if ported to unix type machine, use time.time instead)

## HiSlip and Socket connections not supported

## DESCRIPTION OF FUNCTIONALITY
## This script shows the two best synchronization methods for all InfiniiVision and InfiniiVision-X scopes. Benefits and drawbacks of each method are described.
## Only trivial error handling is provided except in the actual synchronization methods, where it is exactly as needed, though modifiable.
## This script should work for all InfiniiVision and InfiniiVision-X oscilloscopes:
## DSO5000A, DSO/MSO6000A/L, DSO/MSO7000A/B, EDU/DSOX1000A/G, DSO/MSO-X2000A, DSO/MSO-X3000A/T, DSO/MSO-X4000A, DSO/MSO-X6000A, M924xA (PXIe scope)

##############################################################################
## DEFINE CONSTANTS
##############################################################################

## Initialization constants
SCOPE_VISA_ADDRESS = "USB0::0x0957::0x1783::MY47050006::0::INSTR" # Get this from Keysight IO Libraries Connection Expert
    ## Note: Sockets will not work for the blocking_method as there is now way to do a
    device clear over a socket. They are otherwise not tested in this script.
    ## Video: Connecting to Instruments Over LAN, USB, and GPIB in Keysight Connection Expert: https://youtu.be/sZz8bNHX5u4
GLOBAL_TOUT =  10000 # IO time out in milliseconds

TIME_TO_TRIGGER = 10 # Time in seconds
    ## This is the time until the FIRST trigger event.
    ## While the script calculates a general time out for the given setup, it cannot know when a trigger event will occur.  Thus, the user must still set this value.
    ## This time is in addition to the calculated minimum timeout... so, if a scope might take say, 1 us to arm and acquire data,
        ## the signal might take 100 seconds before it occurs... this accounts for that.
    ## The SCOPE_ACQUISITION_TIME_OUT calculation pads this by 1.1

TIME_BETWEEN_TRIGGERS = 0.025 # Time in seconds - for Average, Segmented, and Equivalent Time types/modes, else set to 0
    ## In Average and Segmented Acq. Types, and Equivalent Time mode, the scope makes repeated acquisitions.  This is similar to
        ## the above TIME_TO_TRIGGER, but it is the time BETWEEN triggers.  For example, it might take 10 seconds
        ## for the first trigger event, and then they might start occurring regularly at say, 1 ms intervals.  In
        ## that scenario, 15 seconds (a conservative number for 10s) would be good for TIME_TO_TRIGGER,
        ## and 2 ms (again conservative) would be good for TIME_BETWEEN_TRIGGERS.
    ## The default in this sample script is 0.025 seconds. This is to make the sample script work for the LINE trigger
        ## used in this script when the scope is in Average Acq. Type and Segmented mode, or Equivalent Time mode, to force a
        ## trigger off of the AC input line (:TRIGger:EDGE:SOURce LINE) which runs at 50 or 60 Hz in most
        ## of the world (1/50 Hz -> 20 ms, so use 25 ms to be conservative).
    ## The SCOPE_ACQUISITION_TIME_OUT calculation pads this by 1.1

##############################################################################################
## Define 2 functions to synchronize InfiiVision Oscilloscopes
##############################################################################################

##############################################################################################
## Define a simple and fast function utilizing the blocking :DIGitize command in conjunction with *OPC?
def blocking_method():

    KsInfiniiVisionX.timeout =  SCOPE_ACQUISITION_TIME_OUT # Time in milliseconds (PyVisa uses ms) to wait for the scope to arm, trigger, finish acquisition, and finish any processing.
        ## Note that this is a property of the device interface, KsInfiniiVisionX.
  ## If doing repeated acquisitions, this should be done BEFORE the loop, and changed again after the loop if the goal is to achieve best throughput.

    print "Acquiring signal(s)...\n"
    try: # Set up a try/except block to catch a possible timeout and exit.
        KsInfiniiVisionX.query(":DIGitize;*OPC?") # Acquire the signal(s) with :DIGItize (blocking) and wait until *OPC? comes back with a one. There is no need to issue a *CLS before issuing the :DIGitize command as :DIGitize actually takes care of this for you.
        print "Signal acquired.\n"
        KsInfiniiVisionX.timeout =  GLOBAL_TOUT # Reset timeout back to what it was, GLOBAL_TOUT.
    except Exception: # Catch a possible timeout and exit.
        print "The acquisition timed out, most likely due to no trigger, or improper setup causing no trigger. Properly closing scope connection and exiting script.\n"
        KsInfiniiVisionX.clear() # Clear scope communications interface; a device clear aborts a digitize and clears the scope's IO interface..
        ## Don't do a *CLS.  If you do, you won't be able to do a meaningful :SYSTem:ERRor? query as *CLS clears the error queue
        KsInfiniiVisionX.close() # Close communications interface to scope
        sys.exit("Exiting script.")

    ## Benefits of this method:
        ## Fastest, compact
        ## Only way for Average Acquisition type:
            ## The :SINGle command does not do a complete average.
            ## Counting triggers with :RUN is much too slow
        ## Allows for easy synchronization with math functions
        ## Don't have to deal with the status registers, which can be confusing
        ## Potentially faster than polling_method(), for better throughput
        ## Because it's faster one can retrieve more accurate acquisition times than with a polling method.
        ## Works best for segmented memory if any post processing is done on the scope, e.g. measurements, lister, math, as this does not come back until the processing is all done
            ## In this scenario, :DIGitize does not reduce the sample rate or memory depth.
    ## Drawbacks of this method:
        ## Usually does not fill acquisition memory tot eh maximum available, usually only on-screen data.
        ## May not be at the highest sample rate (compared with the polling_method)
        ## Requires a well-chosen, hard-set timeout that will cover the time to arm, trigger,
            ## and finish acquisition.
        ## Requires Exception handling and a device_clear() for a possible timeout (no trigger event).
            ## Socket connection cannot do device_clear()
        ## Since :DIGitize is a ”specialized form of the :RUN” command, on these scope, that results in:
            ## the sample rate MAY be reduced from using :SINGle - usually at longer time scales -
            ## typically only acquires what is on screen, though at the fastest time scales, more than on screen data may be acquired
            ## Thus, for max memory and max sample rate, use the polling_method(), which uses :SINGle.
    ## How it works:
        ## The :DIGitize command is a blocking command, and thus, all other SCPI commands are blocked until
            ## :DIGitize is completely done.  This includes any subsequent processing that is already set up,
            ## such as math, jitter separation, and measurements.  Key Point: When the *OPC? query is appended to
            ## :DIGitize with a semi-colon (;), which essentially ties it to the same thread in the parser,
            ## it is immediately dealt with when :DIGitize finishes and gives a  1  back to the script, allowing the script to move on.
    ## Other Notes:
        ## If you DO NOT know when a trigger will occur, you will need to (should) set a very long time out.
        ## The timeout will need to be (should be) adjusted before and after the :DIGitize operation,
            ## though this is not absolutely required.
        ## A :DIGitize can be aborted with a device clear, whcih also stops the scope: KsInfiniiVisionX.clear()
        ## :DIGItize disables the anti-aliasing feature (sample rate dither) on all InfiniiVision and InfiniiVision-X scopes.
        ## :DIGitize temporarily blocks the front panel, and all front panel presses are queued until :DIGitize is done.  So if you change the vertical scale, it will not happen until the acquisition is done.
            ## The exception is that the Run/Stop button on the front panel is NOT blocked (unless the front panel is otherwise locked by :SYSTem:LOCK 1).

################################################################################
## Define a function using the non-blocking :SINGle command and polling on the Operation Status Condition Register
def polling_method():

    MAX_TIME_TO_WAIT = SCOPE_ACQUISITION_TIME_OUT/float(1000) # Time in seconds to wait for the scope to arm, trigger, and finish acquisition.
        ## Note that this is NOT a property of the device interface, KsInfiniiVisionX, but rather some constant in the script to be used later with
            ## the Python module ”time,” and will be used as time.clock().

    ## Define "mask" bits and completion criterion.
    ## Mask condition for Run state in the Operation Status Condition (and Event) Register
         ## This can be confusing.  In general, refer to Programmer's Guide chapters on Status Reporting, and Synchronizing Acquisitions
         ## Also see the annotated screenshots included with this sample script.
    RUN_BIT = 3 # The run bit is the 4th bit (see next set of comments @ Completion Criteria).
    RUN_MASK = 1<<RUN_BIT  # This basically means:  2^3 = 8, or rather, in Python 2**3 (<< is a left shift; left shift is fastest); this is used later to
        ## "unmask" the result of the Operation Status Event Register as there is no direct access to the RUN bit.

    ## Completion criteria
    ACQ_DONE = 0 # Means the scope is stopped
    ACQ_NOT_DONE = 1<<RUN_BIT # Means the scope is running; value is 8
        ## This is the 4th bit of the Operation Status Condition (and Event) Register.
        ## The registers are binary and start counting at zero, thus the 4th bit (4th position in a binary representation of decimal 8 = 2^3 = (1 left shift 3).
        ## This is either High (running = 8) or low (stopped and therefore done with acquisition = 0).

    print "Acquiring signal(s)...\n"
    StartTime = time.clock() # Define acquisition start time; This is in seconds.
    KsInfiniiVisionX.write("*CLS;:SINGle") # Beigin Acquisition with *CLS and the non-blocking :SINGle command, concatenated together. The *CLS clears all (non-mask) registers & sets them to 0;

    ## Initialize the loop entry condition (assume Acq. is not done).
    Acq_State = ACQ_NOT_DONE

    ## Poll the scope until Acq_State is a one. (This is NOT a "Serial Poll.")
    while Acq_State == ACQ_NOT_DONE and (time.clock() - StartTime <= MAX_TIME_TO_WAIT):
        Status = int(KsInfiniiVisionX.query(":OPERegister:CONDition?")) # Ask scope if it is done with the acquisition via the Operation Status Condition (not Event) Register.
            ## The Condition register reflects the CURRENT state, while the EVENT register reflects the first event that occurred since it was cleared or read, thus the CONDITION register is used.
            ## DO NOT do: KsInfiniiVisionX.query("*CLS;SINGle;OPERegister:CONDition?") as putting :OPERegister:CONDition? rigth after :SINGle doesn't work reliably
                ## The scope SHOULD trigger, but it sits there with the Single hard key on the scope lit yellow; hitting this key causes a trigger.
        Acq_State = (Status & RUN_MASK) # Bitwise AND of the Status and RUN_MASK.  This exposes ONLY the 4th bit, which is either High (running = 8) or low (stopped and therefore done with acquisition = 0)
        if Acq_State == ACQ_DONE:
            break # Break out of while loop so that the 100 ms pause below is not incurred if done.
        time.sleep(.1) # Pause 100 ms to prevent excessive queries
            ## This can actually be set a little faster, at 0.045.  The point here is that
                ## 1. if there are other things being controlled, going too fast can tie up the bus.
                ## 2. going faster does not work on all scopes.  The symptom of this not working is:
                    ## The scope SHOULD trigger, but it sits there with the Single hard key on the scope lit yellow; hitting this key causes a trigger.
            ## The pause should be at the end of the loop, so that the scope is immediately asked if it is done.
        ## Loop exits when Acq_State != NOT_DONE, that is, it exits the loop when it is DONE or if the max wait time is exceeded.

    if Acq_State == ACQ_DONE: # Acquisition fully completed
        print "Signal acquired.\n"
    else: # Acquisition failed for some reason
        print "Max wait time exceeded."
        print "This happens if there was no trigger event."
        print "Adjust settings accordingly.\n"
        print "Properly closing scope connection and exiting script.\n"
        KsInfiniiVisionX.clear() # Clear scope communications interface
        KsInfiniiVisionX.query(":STOP;*OPC?") # Stop the scope
        KsInfiniiVisionX.close() # Close communications interface to scope
        sys.exit("Exiting script.")
        ## Or do something else...

    ## Benefits of this method:
        ## Don't have to worry about interface timeouts
        ## Easy to expand to know when scope is armed, and triggered
        ## MAY result in a higher sample rate than the blocking method
        ## Always fills max available memory
        ## Can use with a socket connection if desired
    ## Drawbacks of this method:
        ## Slow
        ## Does NOT work for Average Acquisition type
            ## :SINGle does not do a complete average
                ## It does a single acquisition as if it were in NORMal acq. type
                ## Counting triggers in :RUN is much too slow
        ## Works for Segmented Memory, BUT if any post processing is done on the scope, e.g. measurements, lister, math, as this reprotsa that the acquisition is done,
            ## which is correct, BUT  the processing is NOT done, and it willt ake an indefintie amoutn of time to wiat for that, though there is no way to tell if it is done.
            ## Use the blcoking_mthod for Segmented Memory.
        ## Can't be used effectively for synchronizing math functions
            ## It can be done by applying an additional hard coded wait after the acquisition is done.  At least 200 ms is suggested, more may be required.
            ## However, as long as the time out is not excessively short, the math happens fast enough that once :OPERegister:CONDition? comes back as done
                ## that one can just wait for it when it is time to pull the math waveform. The exception would be for eye or jitter mode on an X6000A, where the processing time can be long.
        ## Still need some maximum timeout (here MAX_TIME_TO_WAIT), ideally, or the script will sit in the while loop forever if there is no trigger event
            ## Max time out (here MAX_TIME_TO_WAIT) must also account for any processing done (see comments on math above)
            ## Max time out (here MAX_TIME_TO_WAIT) must also account for time to arm the scope and finish the acquisition
                ## This arm/trigger/finish part is accounted for in the main script.
    ## How it works:
        ## Pretty well explained in line; see annotated screenshots. Basically:
            ## What really matters is the RUN bit in the Operation Condition (not Event) Register.  This bit changes based on the scope state.
            ## If the scope is running, it is high (8), and low (0) if it is stopped.
            ## The only (best) way to get at this bit is with the :OPERation:CONDition? query.  The Operation Condition Register can reflect states
            ## for other scope properties, for example, if the scope is armed, thus it can produce values other than 0 (stopped) or 8 (running).
            ## To handle that, the result of :OPERation:Condition? is bitwise ANDed (& in Python) with an 8.  This is called "unmasking."
            ## Here, the "unmasking" is done in the script.  On the other hand, it is possible to "mask" which bits get passed to the
            ## summary bit to the next register below on the instrument itself.  However, this method it typically only used when working with the Status Byte,
            ## and not used here.
            ## Why 8 = running = not done?
                ## The Run bit is the 4th bit of the Operation Status Condition (and Event) Registers.
                ## The registers are binary and start counting at zero, thus the 4th bit is bit number 3, and 2^3 = 8, and thus it returns an 8 for high and a 0 for low.
            ## Why the CONDITION and NOT the EVENT register?
                ## The Condition register reflects the CURRENT state, while the EVENT register reflects the first event that occurred since it was cleared or read (as in: has it EVER happened?),
                ## thus the CONDITION register is used.
    ## Note that with this method using :SINGle, for InfiniiVision-X scopes only, :SINGle itself forces the trigger sweep mode into NORMal.
        ## This does not happen with the blocking method, using :DIGitize or on the
    InfiniiVsion notXs.

################################################################################
## Connect and initialize scope
################################################################################

## Define VISA Resource Manager & Install directory
## This directory will need to be changed if VISA was installed somewhere else.
rm = visa.ResourceManager() # this uses PyVisa
## This is more or less ok too: rm = visa.ResourceManager('C:\\Program Files (x86)\\IVI Foundation\\VISA\\WinNT\\agvisa\\agbin\\visa32.dll')
## In fact, it is generally not needed to call it explicitly: rm = visa.ResourceManager()

## Open Connection
## Define & open the scope by the VISA address ; # This uses PyVisa
try:
    KsInfiniiVisionX = rm.open_resource(SCOPE_VISA_ADDRESS)
except Exception:
    print "Unable to connect to oscilloscope at " + str(SCOPE_VISA_ADDRESS) + ". Aborting script.\n"
    sys.exit()

## Set Global Timeout
## This can be used wherever, but local timeouts are used for Arming, Triggering, and Finishing the acquisition... Thus it mostly handles IO timeouts
KsInfiniiVisionX.timeout = GLOBAL_TOUT

## Clear the instrument bus
KsInfiniiVisionX.clear()

## Clear all registers and errors
## Always stop scope when making any changes.
KsInfiniiVisionX.query(":STOP;*CLS;*OPC?")

################################################################################
## Main code
################################################################################

try:

    ############################################################################
    ## Setup scope

    ## Note that one would normally perform a reset (default setup) if one were to create the setup from scratch...
        ## But here we will use the scope "as is" for the most part.
    ## KsInfiniiumScope.query("*RST;*OPC?") # resets the scope

    KsInfiniiVisionX.query(":STOP;*OPC?") # Scope always should be stopped when making changes.

    ## Whatever is needed

    ## For this example, the scope will be forced to trigger on the (power) LINE voltage so something happens
    KsInfiniiVisionX.write(":TRIGger:SWEep NORMal") # Always use normal trigger sweep, never auto.
    KsInfiniiVisionX.query(":TRIGger:EDGE:SOURce LINE;*OPC?") # This line simply gives the scope something to trigger on

    ## Clear the display (only so the user can see the waveform being acquired, otherwise this is not needed at all)
    KsInfiniiVisionX.write(":CDISplay")

    ###############################################################################
    ## Calculate acquisition timeout/wait time by short, overestimate method

    ## Create some default variables
    N_AVERAGES = 1
    N_SEGMENTS = 1

    ## Get some info about the scope time base setup
    HO         = float(KsInfiniiVisionX.query(":TRIGger:HOLDoff?"))
    T_RANGE    = float(KsInfiniiVisionX.query(":TIMebase:RANGe?"))
    T_POSITION = float(KsInfiniiVisionX.query(":TIMebase:POSition?"))

    ## Determine Acquisition Type and Mode:
    ACQ_TYPE = str(KsInfiniiVisionX.query(":ACQuire:TYPE?").strip("\n"))
    ACQ_MODE = str(KsInfiniiVisionX.query(":ACQuire:MODE?").strip("\n"))

    if ACQ_MODE == "SEGM":
        N_SEGMENTS= float(KsInfiniiVisionX.query(":ACQuire:SEGMented:COUNt?"))
        ## Note that if there is a lot of analysis associated segments, e.g. serial data decode, the timeout will likely need to be longer than calculated.
            ## The user is encouraged to manually set up the scope in this case, as it will be used, and time it, and use that, with a little overhead.
            ## Blocking method is recommended for Segmented Memory mode.
    elif ACQ_TYPE == "AVER":
        N_AVERAGES = float(KsInfiniiVisionX.query(":ACQuire:COUNt?"))

    ## Calculate acuisition timeout by overestimate method:
    SCOPE_ACQUISITION_TIME_OUT = (float(TIME_TO_TRIGGER)*1.1 + (T_RANGE*2.0 + abs(T_POSITION)*2.0 + HO*1.1 + float(TIME_BETWEEN_TRIGGERS)*1.1)*N_SEGMENTS*N_AVERAGES)*1000.0 # Recall that PyVisa timeouts are in ms, so multiply by 1000

    ## Ensure the timeout is no less than 10 seconds
    if SCOPE_ACQUISITION_TIME_OUT < 10000.0:
        SCOPE_ACQUISITION_TIME_OUT = 10000.0

    ## What about Equivalent Time Mode and other odd modes such as Jitter or Eye (the last two only being found on the X6000A), and math functions?
        ## In most cases, the padding and 10 second minimum timeout will take care of this.
        ## Equivalent Time Mode only has an effects at the fastest time scales, so it really doesnt make a difference as long as a trigger signal is present. If trigger signal occurs rarely, adjust the TIME_BETWEEN_TRIGGERS constant accordingly.
        ## For math, the math will happen fast enough that the padding in the timeout calculation takes care of this.
        ## For jitter mode on the X6000A, the user can try this, method, and typically there is always s signal present, and the 10 second minimum should work out.  If not, make it bigger, or increase padding.
        ## For Eye mode on the X6000A, none of this works anyway, and you have to use :RUN (or :RTEYe:ACQuire) and :STOP.

    ##############################################################################
    ## Acquire Signal

    ## Choose blocking_method or polling_method
    ## There is no a-priori reason to do this as a Python function except that a user would probably want to use it repeatedly

    ## If Acquisition Type is Average, always use blocking_method() to get complete average
    polling_method()

    ##############################################################################
    ## Do Something with data... save, export, additional analysis...

    ## For example, make a peak-peak voltage measurement on channel 1:
    Vpp_Ch1 = str(KsInfiniiVisionX.query("MEASure:VPP? CHANnel1")).strip("\n") # The result comes back with a newline, so remove it with .strip("\n")
    print "Vpp Ch1 = " + Vpp_Ch1 + " V\n"

    ##############################################################################

    ##############################################################################
    ## Done - cleanup
    ##############################################################################

    KsInfiniiVisionX.clear() # Clear scope communications interface
    KsInfiniiVisionX.close() # Close communications interface to scope
except KeyboardInterrupt:
    KsInfiniiVisionX.clear()
    KsInfiniiVisionX.query(”:STOP;*OPC?”)
    KsInfiniiVisionX.write(”:SYSTem:LOCK 0”)
    KsInfiniiVisionX.clear()
    KsInfiniiVisionX.close()
    sys.exit(”User Interupt.  Properly closing scope and aborting script.”)
except Exception:
    KsInfiniiVisionX.clear()
    KsInfiniiVisionX.query(”:STOP;*OPC?”)
    KsInfiniiVisionX.write(”:SYSTem:LOCK 0”)
    KsInfiniiVisionX.clear()
    KsInfiniiVisionX.close()
    sys.exit(”Something went wrong.  Properly closing scope and aborting script.”)

print ”Done.”
```

Listing 7: `plot-gui.py`

```
import sys
from PyQt5.QtWidgets import QLabel, QApplication, QMainWindow, QMenu, QVBoxLayout,
    QSizePolicy, QMessageBox, QAction, QLineEdit, QWidget, QPushButton
from PyQt5.QtGui import QIcon
from PyQt5.QtGui import *
#**************************************************************************
import matplotlib
matplotlib.use(”Qt5Agg”)
#**************************************************************************
from PyQt5 import QtCore
from PyQt5.QtCore import *
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
from matplotlib.figure import Figure
import matplotlib.pyplot as plt
import pandas as pd
import random
import DAQ
#**************************************************************************

osc_daq = DAQ.init_osc()
overall_df = pd.DataFrame()

def main():
    app = QApplication(sys.argv)
    ex = App()
    sys.exit(app.exec_())

class App(QMainWindow):

    def __init__(self):
        super().__init__()
        # Panel setup option
        self.left = 100
        self.top = 100
        self.title = 'Secure Our System'
        self.width = 1440
        self.height = 960
        self.is_trigger = True

        self.m = self.initUI()

    # params
    def _initButton(self,button,tooltip,x,y,sizex,sizey):
        button.setToolTip(tooltip)
        button.move(x,y)
        button.resize(sizex,sizey)



    def initUI(self):
        # actualy set the panel
        self.setWindowTitle(self.title)
        self.setGeometry(self.left, self.top, self.width, self.height)

        # the graph
        m = PlotCanvas(self, width=5, height=4)
        m.move(10,10)

        # start button
        self.button_start = QPushButton('Start', self)
        self._initButton(self.button_start,'Start Recording',550,70,120,50)
        self.button_start.clicked.connect(self.on_click)

        # Create save button
        self.button_save = QPushButton('Export data', self)
        self._initButton(self.button_save,'Save data as CSV file',550,120,120,50)
        self.button_save.clicked.connect(self.on_export)

        # Create toggle Button
        self.button_toggle = QPushButton('Trigger/Auto', self)
        self._initButton(self.button_toggle,'Toggle Trigger Mode On and OFF',550,170,120,50)
        self.button_toggle.clicked.connect(self.on_switch_mode)

        # Create textbox
        self.textbox = QLineEdit(self)
        self.textbox.move(670, 130)
        self.textbox.resize(140, 20)
        self.textbox.setPlaceholderText('Enter file name here')
        self.textbox.setText('default');
        self.label_ext = QLabel(".csv", self);
        self.label_ext.move(820,125)

        self.input_times = QLineEdit(self)
        self.input_times.move(670, 70)
        self.input_times.resize(140, 20)
        self.input_times.setPlaceholderText('X TIMES')
        self.input_times.setText('1');

        self.show()
        return m

    @pyqtSlot()
    def on_click(self):
        print('PyQt5 button click')
        for num_capture in range (int(self.input_times.text())):
            print('is Trigger' + str(self.is_trigger))
            global overall_df
            wave_data = DAQ.take_waveform(osc_daq, self.is_trigger)
            placeholder_data = pd.DataFrame([random.random() for i in range(25)])
            overall_df = overall_df.append(wave_data, ignore_index=True)
            print("Shape: " + str(overall_df.shape))

            self.m.setData(placeholder_data.iloc[:,0])
            self.on_export()


    @pyqtSlot()
    def on_export(self):
        overall_df.to_csv(self.textbox.text() + '.csv')

    @pyqtSlot()
    def on_switch_mode(self):
        self.is_trigger = not self.is_trigger
        print(self.is_trigger)
```

```


class PlotCanvas(FigureCanvas):

    def __init__(self, parent=None, width=5, height=4, dpi=100):
        fig = Figure(figsize=(width, height), dpi=dpi)

        FigureCanvas.__init__(self, fig)
        self.setParent(parent)

        FigureCanvas.setSizePolicy(self,
                QSizePolicy.Expanding,
                QSizePolicy.Expanding)
        FigureCanvas.updateGeometry(self)
        self.plot([random.random() for i in range(25)])


    def plot(self, data):
        self.figure.clear()
        ax = self.figure.add_subplot(1,1,1)
        ax.set_title('Waveform')
        self.lines = ax.plot(data, 'r-')
        ax.set_xlabel('Time (ms)')
        ax.set_ylabel('Voltage (V)')
        self.draw()

    def setData(self, data):
        self.lines.pop(0).remove()
        self.plot(data)

if __name__ == '__main__':
    main()
```