```
__author__ = 'Robert Cope', 'Jochen Hoenicke'

import sys
import time
import usb1
import array
import select
import libusb1
import threading
from struct import pack
import os  #ji for os.name

isPython3 = '3' == sys.version[0]

from PyHT6022.HantekFirmware import custom_firmware_BE, custom_firmware_BL,
fx2_ihex_to_control_packets

class Oscilloscope(object):
    NO_FIRMWARE_VENDOR_ID = 0x04B4
    FIRMWARE_PRESENT_VENDOR_ID = 0x04B5
    MODEL_ID_BE = 0x6022
    MODEL_ID_BL = 0x602A

    RW_FIRMWARE_REQUEST = 0xa0
    RW_FIRMWARE_INDEX = 0x00

    RW_EEPROM_REQUEST = 0xa2
    RW_EEPROM_INDEX = 0x00

    SET_SAMPLE_RATE_REQUEST = 0xe2
    SET_SAMPLE_RATE_VALUE = 0x00
    SET_SAMPLE_RATE_INDEX = 0x00

    SET_CH1_VR_REQUEST = 0xe0
    SET_CH1_VR_VALUE = 0x00
    SET_CH1_VR_INDEX = 0x00

    SET_CH2_VR_REQUEST = 0xe1
    SET_CH2_VR_VALUE = 0x00
    SET_CH2_VR_INDEX = 0x00

    TRIGGER_REQUEST = 0xe3
    TRIGGER_VALUE = 0x00
    TRIGGER_INDEX = 0x00

    SET_NUMCH_REQUEST = 0xe4
    SET_NUMCH_VALUE = 0x00
    SET_NUMCH_INDEX = 0x00

    SET_AC_DC_REQUEST = 0xe5
```

```
SET_AC_DC_VALUE = 0x00
SET_AC_DC_INDEX = 0x00

AC = 0
DC = 1
AC_AC = 0x00
AC_DC = 0x01
DC_AC = 0x10
DC_DC = 0x11

SET_CAL_FREQ_REQUEST = 0xe6
SET_CAL_FREQ_VALUE = 0x00
SET_CAL_FREQ_INDEX = 0x00

CALIBRATION_EEPROM_OFFSET = 0x08

SAMPLE_RATES = {106: ( "60 KS/s",  60e3),
                110: ("100 KS/s", 100e3),
                120: ("200 KS/s", 200e3),
                150: ("500 KS/s", 500e3),
                  1: (  "1 MS/s",   1e6),
                  2: (  "2 MS/s",   2e6),
                  3: (  "3 MS/s",   3e6),
                  4: (  "4 MS/s",   4e6),
                  5: (  "5 MS/s",   5e6),
                  6: (  "6 MS/s",   6e6),
                  8: (  "8 MS/s",   8e6),
                 10: ( "10 MS/s",  10e6),
                 12: ( "12 MS/s",  12e6),
                 15: ( "15 MS/s",  15e6),
                 16: ( "16 MS/s",  16e6),
                 24: ( "24 MS/s",  24e6),
                 30: ( "30 MS/s",  30e6),
                 48: ( "48 MS/s",  48e6) }

VOLTAGE_RANGES = { 1: ('+/- 5V', 0.0390625, 2.5),
                   2: ('+/- 2.5V', 0.01953125, 1.25),
                   5: ('+/- 1V', 0.0078125, 0.5),
                  10: ('+/- 500mV', 0.00390625, 0.25) }

CAL_FREQUENCYS = {
                  105: ( " 50 Hz",     50 ),
                  110: ( "100 Hz",    100 ),
                  120: ( "200 Hz",    200 ),
                  150: ( "500 Hz",    500 ),
                    1: (  "1 kHz",   1000 ),
                    2: (  "2 kHz",   2000 ),
                    5: (  "5 kHz",   5000 ),
                   10: ( "10 kHz",  10000 ),
                   20: ( "20 kHz",  20000 ),
```

```
                     50: ( "50 kHz",  50000 ),
                    100: ("100 kHz", 100000 )
                   }



    def __init__(self, scope_id=0):
        self.device = None
        self.device_handle = None
        self.context = usb1.USBContext()
        self.is_device_firmware_present = False
        self.supports_single_channel = False
        self.is_iso = False
        self.packetsize = None
        self.num_channels = 2
        self.ac_dc_status = 0x11
        self.scope_id = scope_id


    def setup(self):
        """
        Attempt to find a suitable scope to run.
        :return: True if a 6022{BE,BL} scope was found, False otherwise.
        """
        # 1st look for 6022BE devices
        self.device = (
            self.context.getByVendorIDAndProductID(
                self.FIRMWARE_PRESENT_VENDOR_ID, self.MODEL_ID_BE,
skip_on_error=True, skip_on_access_error=True)
            or self.context.getByVendorIDAndProductID(
                self.NO_FIRMWARE_VENDOR_ID, self.MODEL_ID_BE, skip_on_error=True,
skip_on_access_error=True)
            )
        if self.device:
            self.is_device_firmware_present = self.device.getVendorID() ==
self.FIRMWARE_PRESENT_VENDOR_ID
            return True

        # if not found look for 6022BL
        self.device = (
            self.context.getByVendorIDAndProductID(
                self.FIRMWARE_PRESENT_VENDOR_ID, self.MODEL_ID_BL,
skip_on_error=True, skip_on_access_error=True)
            or self.context.getByVendorIDAndProductID(
                self.NO_FIRMWARE_VENDOR_ID, self.MODEL_ID_BL, skip_on_error=True,
skip_on_access_error=True)
            )
        if self.device:
            self.is_device_firmware_present = self.device.getVendorID() ==
self.FIRMWARE_PRESENT_VENDOR_ID
```

```
            return True

        return False


    def open_handle(self):
        """
        Open a device handle for the scope. This needs to occur before sending any
commands.
        :return: True if successful, False otherwise. May raise various libusb
exceptions on fault.
        """
        if self.device_handle:
            return True
        if not self.device and not self.setup():
            return False
        self.device_handle = self.device.open()
        #ji 'kerneDriverActive' is not available in Windows
        #ji if self.device_handle.kernelDriverActive(0):
        if (os.name != 'nt') and self.device_handle.kernelDriverActive(0):
            self.device_handle.detachKernelDriver(0)
        self.device_handle.claimInterface(0)
        if self.is_device_firmware_present:
            self.set_num_channels(2)
            self.set_interface(0)
        return True


    def close_handle(self, release_interface=True):
        """
        Close the current scope device handle. This should always be called at
clean-up.
        :param release_interface: (OPTIONAL) Attempt to release the interface, if we
still have it.
        :return: True if successful. May assert or raise various libusb errors if
something went wrong.
        """
        if not self.device_handle:
            return True
        if release_interface:
            self.device_handle.releaseInterface(0)
        self.device_handle.close()
        self.device_handle = None
        return True


    def get_fw_version( self ):
        """
        Returns the BCD coded firmware/device version
        """
```

```
        if self.device:
            return self.device.getbcdDevice()
        else:
            return None


    def __del__(self):
        self.close_handle()


    def poll(self):
        self.context.handleEvents()


    def flash_firmware(self, firmware=None, supports_single_channel=True,
timeout=60):
        """
        Flash scope firmware to the target scope device. This needs to occur once
when the device is first attached,
        as the 6022BE does not have any persistant storage.
        :param firmware: (OPTIONAL) The firmware packets to send. Default: custom
firmware (either BE or BL).
        :param supports_single_channel: (OPTIONAL) Set this to false if loading the
stock firmware, as it does not
                                        support reducing the number of active
channels.
        :param timeout: (OPTIONAL) A timeout for each packet transfer on the
firmware upload. Default: 60 seconds.
        :return: True if the correct device vendor was found after flashing
firmware, False if the default Vendor ID
                 was present for the device. May assert or raise various libusb
errors if something went wrong.
        """
        if not self.device_handle:
            assert self.open_handle()
        if not firmware: # called without an explicit firmware parameter
            if self.device.getProductID() == self.MODEL_ID_BE:
                firmware = custom_firmware_BE
            elif self.device.getProductID() == self.MODEL_ID_BL:
                firmware = custom_firmware_BL
            else:
                return False
        for packet in firmware:
            bytes_written = self.device_handle.controlWrite(0x40,
self.RW_FIRMWARE_REQUEST,
                                                            packet.value,
self.RW_FIRMWARE_INDEX,
                                                            packet.data,
timeout=timeout)
            assert bytes_written == packet.size
```

```
        # After firmware is written, scope will typically show up again as a
different device, so scan again
        self.close_handle(release_interface=False)
        time.sleep(.5)
        while not self.setup():
            time.sleep(0.1)
        self.supports_single_channel = supports_single_channel
        self.open_handle()
        return self.is_device_firmware_present


    def flash_firmware_from_hex(self, hex_file, timeout=60):
        """
        Open an Intel hex file for the 8051 and try to flash it to the scope.
        :param hex_file: The hex file to load for flashing.
        :param timeout: (OPTIONAL) A timeout for each packet transfer on the
firmware upload. Default: 60 seconds.
        :return: True if the correct device vendor was found after flashing
firmware, False if the default Vendor ID
                 was present for the device. May assert or raise various libusb
errors if something went wrong.
        """
        return self.flash_firmware(firmware=fx2_ihex_to_control_packets(hex_file),
timeout=timeout)


    def read_firmware(self, address=0, length=8192, to_ihex=True, chunk_len=16,
timeout=60):
        """
        Read the entire device RAM, and return a raw string.
        :param to_ihex: (OPTIONAL) Convert the firmware into the Intel hex format
after reading. Otherwise, return
                        the firmware is a raw byte string. Default: True
        :param chunk_len: (OPTIONAL) The length of RAM chunks to pull from the
device at a time. Default: 16 bytes.
        :param timeout: (OPTIONAL) A timeout for each packet transfer on the
firmware upload. Default: 60 seconds.
        :return: The raw device firmware, if successful.
                 May assert or raise various libusb errors if something went wrong.
        """
        if not self.device_handle:
            assert self.open_handle()

        bytes_written = self.device_handle.controlWrite(0x40,
self.RW_FIRMWARE_REQUEST,
                                                        0xe600,
self.RW_FIRMWARE_INDEX,
                                                        b'\x01', timeout=timeout)
        assert bytes_written == 1
        firmware_chunk_list = []
```

```
        # TODO: Fix this for when 8192 isn't divisible by chunk_len
        for packet in range(0, length//chunk_len):
            chunk = self.device_handle.controlRead(0x40, self.RW_FIRMWARE_REQUEST,
                                                   address + packet * chunk_len,
self.RW_FIRMWARE_INDEX,
                                                   chunk_len, timeout=timeout)
            firmware_chunk_list.append(chunk)
            assert len(chunk) == chunk_len
        bytes_written = self.device_handle.controlWrite(0x40,
self.RW_FIRMWARE_REQUEST,
                                                        0xe600,
self.RW_FIRMWARE_INDEX,
                                                        b'\x00', timeout=timeout)
        assert bytes_written == 1
        if not to_ihex:
            return ''.join(firmware_chunk_list)
        else:
            lines = []
            for i, chunk in enumerate(firmware_chunk_list):
                addr = address + i*chunk_len
                iterable_chunk = array.array('B', chunk)
                hex_data = "".join(["{:02x}".format(b) for b in iterable_chunk])
                total_sum = (sum(iterable_chunk) + chunk_len + (addr % 256) + (addr
>> 8)) % 256
                checksum = (((0xFF ^ total_sum) & 0xFF) + 0x01) % 256
                line =
":{nbytes:02x}{addr:04x}{itype:02x}{hex_data}{checksum:02x}".format(nbytes=chunk_len
,

      addr=addr,

      itype=0x00,

      hex_data=hex_data,

      checksum=checksum)
                lines.append(line)
            # Add stop record at the end.
            lines.append(":00000001ff")
            return "\n".join(lines)


    def get_calibration_values(self, size=32, timeout=0):
        """
        Retrieve the current calibration values from the oscilloscope.
        :param timeout: (OPTIONAL) A timeout for the transfer. Default: 0 (No
timeout)
        :return: A size size single byte int list of calibration values, if
successful.
                 May assert or raise various libusb errors if something went wrong.
```

```
        """
        return array.array('B', self.read_eeprom(self.CALIBRATION_EEPROM_OFFSET,
size, timeout=timeout))


    def set_calibration_values(self, cal_list, timeout=0):
        """
        Set the a calibration level for the oscilloscope.
        :param cal_list: The list of calibration values, should usually be 32 single
byte ints.
        :param timeout: (OPTIONAL) A timeout for the transfer. Default: 0 (No
timeout)
        :return: True if successful. May assert or raise various libusb errors if
something went wrong.
        """
        cal_list = cal_list if isinstance(cal_list, bytearray) else array.array('B',
cal_list).tostring()
        return self.write_eeprom(self.CALIBRATION_EEPROM_OFFSET, cal_list,
timeout=timeout)


    def read_eeprom(self, offset, length, timeout=0):
        """
        Retrieve values from the eeprom on the oscilloscope.
        :param offset: start address into the eeprom.
        :param length: number of bytes to retrieve.
        :param timeout: (OPTIONAL) A timeout for the transfer. Default: 0 (No
timeout)
        :return: The raw eeprom data.
        """
        if not self.device_handle:
            assert self.open_handle()
        data = self.device_handle.controlRead(0x40, self.RW_EEPROM_REQUEST, offset,
                                              self.RW_EEPROM_INDEX, length,
timeout=timeout)
        return data


    def write_eeprom(self, offset, data, timeout=0):
        """
        Set the a calibration level for the oscilloscope.
        :param offset: The start address into the eeprom.
        :param data: The raw string of data to write.
        :param timeout: (OPTIONAL) A timeout for the transfer. Default: 0 (No
timeout)
        :return: True if successful. May assert or raise various libusb errors if
something went wrong.
        """
        if not self.device_handle:
            assert self.open_handle()
```

```
        data_len = self.device_handle.controlWrite(0x40, self.RW_EEPROM_REQUEST,
offset,
                                                   self.RW_EEPROM_INDEX, data,
timeout=timeout)
        assert data_len == len(data)
        return True


    def start_capture(self, timeout=0):
        """
        Tell the device to start capturing samples.
        :param timeout: (OPTIONAL) A timeout for the transfer. Default: 0 (No
timeout)
        :return: True if successful. May assert or raise various libusb errors if
something went wrong.
        """
        bytes_written = self.device_handle.controlWrite(0x40, self.TRIGGER_REQUEST,
                                                        self.TRIGGER_VALUE,
self.TRIGGER_INDEX,
                                                        b'\x01', timeout=timeout)
        return bytes_written == 1


    def stop_capture(self, timeout=0):
        """
        Tell the device to stop capturing samples.  This is only supported
        with the custom firmware.
        :param timeout: (OPTIONAL) A timeout for the transfer. Default: 0 (No
timeout)
        :return: True if successful. May assert or raise various libusb errors if
something went wrong.
        """
        bytes_written = self.device_handle.controlWrite(0x40, self.TRIGGER_REQUEST,
                                                        self.TRIGGER_VALUE,
self.TRIGGER_INDEX,
                                                        b'\x00', timeout=timeout)
        return bytes_written == 1


    def read_data(self, data_size=0x400, raw=False, timeout=0):
        """
        Read both channel's ADC data from the device. No trigger support, you need
to do this in software.
        :param data_size: (OPTIONAL) The number of data points for each channel to
retrieve. Default: 0x400 points.
        :param raw: (OPTIONAL) Return the raw bytestrings from the scope. Default:
Off
        :param timeout: (OPTIONAL) The timeout for each bulk transfer from the
scope. Default: 0 (No timeout)
        :return: If raw, two bytestrings are returned, the first for CH1, the second
```

```
for CH2. If raw is off, two
                lists are returned (by iterating over the bytestrings and
converting to ordinals). The lists contain
                the ADC value measured at that time, which should be between 0 -
255.

                If you'd like nicely scaled data, just dump the return lists into
the scale_read_data method with
                your current voltage range setting.

                This method may assert or raise various libusb errors if something
went wrong.
        """
        data_size *= self.num_channels
        if not self.device_handle:
            assert self.open_handle()

        self.start_capture()
        data = self.device_handle.bulkRead(0x86, data_size, timeout=timeout)
        self.stop_capture()
        if self.num_channels == 2:
            chdata = data[::2], data[1::2]
        else:
            chdata = data, ''
        if raw:
            return chdata
        else:
            return array.array('B', chdata[0]), array.array('B', chdata[1])


    def build_data_reader(self, raw=False):
        """
        Build a (slightly) more optimized reader closure, for (slightly) better
performance.
        :param raw: (OPTIONAL) Return the raw bytestrings from the scope. Default:
Off
        :return: A fast_read_data function, which behaves much like the read_data
function. The fast_read_data
                function returned takes two parameters:
                :param data_size: Number of data points to return (1 point <-> 1
byte).
                :param timeout: (OPTIONAL) The timeout for each bulk transfer from
the scope. Default: 0 (No timeout)
                :return:  If raw, two bytestrings are returned, the first for CH1,
the second for CH2. If raw is off,
                two lists are returned (by iterating over the bytestrings and
converting to ordinals).
                The lists contain the ADC value measured at that time, which should
be between 0 - 255.
```

```
        This method and the closure may assert or raise various libusb errors if
something went/goes wrong.
        """
        if not self.device_handle:
            assert self.open_handle()
        scope_bulk_read = self.device_handle.bulkRead
        array_builder = array.array
        if self.num_channels == 1 and raw:
            def fast_read_data(data_size, timeout=0):
                data = scope_bulk_read(0x86, data_size, timeout)
                return data, ''
        elif self.num_channels == 1 and not raw:
            def fast_read_data(data_size, timeout=0):
                data = scope_bulk_read(0x86, data_size, timeout)
                return array_builder('B', data), array_builder('B', '')
        elif self.num_channels == 2 and raw:
            def fast_read_data(data_size, timeout=0):
                data_size <<= 0x1
                data = scope_bulk_read(0x86, data_size, timeout)
                return data[::2], data[1::2]
        elif self.num_channels == 2 and not raw:
            def fast_read_data(data_size, timeout=0):
                data_size <<= 0x1
                data = scope_bulk_read(0x86, data_size, timeout)
                return array_builder('B', data[::2]), array_builder('B', data[1::2])
        else:
            # Should never be here.
            assert False
        return fast_read_data


    def set_interface(self, alt):
        """
        Set the alternative interface (bulk or iso) to use.  This is only
        supported with the custom firmware.  Iso stands for isochronous usb
        transfer and this guarantees a bandwidth and therefore make gapless
        sampling more likely.
        :param int alt: The interface to use.  Values are
                            0 <-> Bulk transfer
                            1 <-> Fastest Iso transfer
                            2 <-> Slower Iso transfer (needs less usb bandwidth)
                            3 <-> Even slower Iso transfer
        :return: True if successful. May assert or raise various libusb errors if
something went wrong.
        """
        if not self.device_handle:
            assert self.open_handle()
        self.device_handle.setInterfaceAltSetting(0, alt)
        endpoint_info = self.device[0][0][alt][0]
        self.is_iso = ((endpoint_info.getAttributes() &
```

```
libusb1.LIBUSB_TRANSFER_TYPE_MASK)
                        == libusb1.LIBUSB_TRANSFER_TYPE_ISOCHRONOUS)
        maxpacketsize = endpoint_info.getMaxPacketSize()
        self.packetsize = ((maxpacketsize >> 11)+1) * (maxpacketsize & 0x7ff)
        return True


    def read_async_iso(self, callback, packets, outstanding_transfers, raw):
        """
        Internal function to read from isochronous channel.  External
        users should call read_async.
        """
        array_builder = array.array
        shutdown_event = threading.Event()
        shutdown_is_set = shutdown_event.is_set
        if self.num_channels == 1 and raw:
            def transfer_callback(iso_transfer):
                for (status, data) in iso_transfer.iterISO():
                    callback(data, '')
                if not shutdown_is_set():
                    iso_transfer.submit()
        elif self.num_channels == 1 and not raw:
            def transfer_callback(iso_transfer):
                for (status, data) in iso_transfer.iterISO():
                    callback(array_builder('B', data), [])
                if not shutdown_is_set():
                    iso_transfer.submit()
        elif self.num_channels == 2 and raw:
            def transfer_callback(iso_transfer):
                for (status, data) in iso_transfer.iterISO():
                    callback(data[::2], data[1::2])
                if not shutdown_is_set():
                    iso_transfer.submit()
        elif self.num_channels == 2 and not raw:
            def transfer_callback(iso_transfer):
                for (status, data) in iso_transfer.iterISO():
                    callback(array_builder('B', data[::2]), array_builder('B',
data[1::2]))
                if not shutdown_is_set():
                    iso_transfer.submit()
        else:
            assert False
        for _ in range(outstanding_transfers):
            transfer = self.device_handle.getTransfer(iso_packets=packets)
            transfer.setIsochronous(0x82, (packets*self.packetsize),
callback=transfer_callback)
            transfer.submit()
        return shutdown_event
```

```
    def read_async_bulk(self, callback, packets, outstanding_transfers, raw):
        """
        Internal function to read from bulk channel.  External
        users should call read_async.
        """
        array_builder = array.array
        shutdown_event = threading.Event()
        shutdown_is_set = shutdown_event.is_set
        if self.num_channels == 1 and raw:
            def transfer_callback(bulk_transfer):
                data = bulk_transfer.getBuffer()[0:bulk_transfer.getActualLength()]
                callback(data, '')
                if not shutdown_is_set():
                    bulk_transfer.submit()
        elif self.num_channels == 1 and not raw:
            def transfer_callback(bulk_transfer):
                data = bulk_transfer.getBuffer()[0:bulk_transfer.getActualLength()]
                callback(array_builder('B', data), [])
                if not shutdown_is_set():
                    bulk_transfer.submit()
        elif self.num_channels == 2 and raw:
            def transfer_callback(bulk_transfer):
                data = bulk_transfer.getBuffer()[0:bulk_transfer.getActualLength()]
                callback(data[::2], data[1::2])
                if not shutdown_is_set():
                    bulk_transfer.submit()
        elif self.num_channels == 2 and not raw:
            def transfer_callback(bulk_transfer):
                data = bulk_transfer.getBuffer()[0:bulk_transfer.getActualLength()]
                callback(array_builder('B', data[::2]), array_builder('B',
data[1::2]))
                if not shutdown_is_set():
                    bulk_transfer.submit()
        else:
            assert False
        for _ in range(outstanding_transfers):
            transfer = self.device_handle.getTransfer(iso_packets=packets)
            transfer.setBulk(0x86, (packets*self.packetsize),
callback=transfer_callback)
            transfer.submit()
        return shutdown_event


    def read_async(self, callback, data_size, outstanding_transfers=3, raw=False):
        """
        Read both channel's ADC data from the device asynchronously. No trigger
support, you need to do this in software.
        The function returns immediately but the data is then sent asynchronously to
the callback function whenever it
        receives new samples.
```

```
        :param callback: A function with two arguments that take the samples for the
first and second channel.
        :param data_size: The block size for each sample.  This is automatically
rounded up to the nearest multiple of
                          the native block size.
        :param int outstanding_transfers: (OPTIONAL) The number of transfers sent to
the kernel at the same time to
                improve gapless sampling.  The higher, the more likely it works, but
the more resources it will take.
        :param raw: (OPTIONAL) Whether the samples should be returned as raw string
(8-bit data) or as an array of bytes.
        :return: Returns a shutdown event handle if successful (and then calls the
callback asynchronously).
                 Call set() on the returned event to stop sampling.
        """
        # data_size to packets
        packets = (data_size + self.packetsize-1)//self.packetsize
        if self.is_iso:
            return self.read_async_iso(callback, packets, outstanding_transfers,
raw)
        else:
            return self.read_async_bulk(callback, packets, outstanding_transfers,
raw)


    @staticmethod
    def scale_read_data(read_data, voltage_range, probe_multiplier=1):
        """
        Convenience function for converting data read from the scope to nicely
scaled voltages.
        :param list read_data: The list of points returned from the read_data
functions.
        :param int voltage_range: The voltage range current set for the channel.
        :param int probe_multiplier: (OPTIONAL) An additonal multiplictive factor
for changing the probe impedance.
                                     Default: 1
        :return: A list of correctly scaled voltages for the data.
        """
        scale_factor = (5.0 * probe_multiplier)/(voltage_range << 7)
        return [(datum - 128)*scale_factor for datum in read_data]


    @staticmethod
    def voltage_to_adc(voltage, voltage_range, probe_multiplier=1):
        """
        Convenience function for analog voltages into the ADC count the scope would
see.
        :param float voltage: The analog voltage to convert.
        :param int voltage_range: The voltage range current set for the channel.
        :param int probe_multiplier: (OPTIONAL) An additonal multiplictive factor
```

```
for changing the probe impedance.
                                Default: 1
        :return: The corresponding ADC count.
        """
        return voltage*(voltage_range << 7)/(5.0 * probe_multiplier) + 128


    @staticmethod
    def adc_to_voltage(adc_count, voltage_range, probe_multiplier=1):
        """
        Convenience function for converting an ADC count from the scope to a nicely
scaled voltage.
        :param int adc_count: The scope ADC count.
        :param int voltage_range: The voltage range current set for the channel.
        :param int probe_multiplier: (OPTIONAL) An additonal multiplictive factor
for changing the probe impedance.
                                Default: 1
        :return: The analog voltage corresponding to that ADC count.
        """
        return (adc_count - 128)*(5.0 * probe_multiplier)/(voltage_range << 7)


    def set_sample_rate(self, rate_index, timeout=0):
        """
        Set the sample rate index for the scope to sample at. This determines the
time between each point the scope
        returns.
        :param rate_index: The rate_index. These are the keys for the SAMPLE_RATES
dict for the Oscilloscope object.
                           Common rate_index values and actual sample rate per
channel:
                           106 <->  60 kS/s
                           110 <> 100 kS/s
                           120 <> 200 kS/s
                           150 <> 500 kS/s
                             1 <->   1 MS/s
                             2 <->   2 MS/s
                             3 <->   3 MS/s
                             4 <->   4 MS/s
                             5 <->   5 MS/s
                             6 <->   6 MS/s
                             8 <->   8 MS/s
                            10 <>  10 MS/s
                            12 <>  12 MS/s
                            15 <>  15 MS/s
                            16 <>  16 MS/s
                            24 <>  24 MS/s
                            30 <>  30 MS/s
                            48 <>  48 MS/s
```

```
                            Other values are not supported.
        :param timeout:
        :return: True if successful. This method may assert or raise various libusb
errors if something went wrong.
        """
        if not self.device_handle:
            assert self.open_handle()
        bytes_written = self.device_handle.controlWrite(0x40,
self.SET_SAMPLE_RATE_REQUEST,
                                                        self.SET_SAMPLE_RATE_VALUE,
                                                        self.SET_SAMPLE_RATE_INDEX,
                                                        pack("B", rate_index),
timeout=timeout)
        assert bytes_written == 0x01
        return True


    def convert_sampling_rate_to_measurement_times(self, num_points, rate_index):
        """
        Convenience method for converting a sampling rate index into a list of times
from beginning of data collection
        and getting human-readable sampling rate string.
        :param num_points: The number of data points.
        :param rate_index: The sampling rate index used for data collection.
        :return: A list of times in seconds from beginning of data collection, and
the nice human readable rate label.
        """
        rate_label, rate = self.SAMPLE_RATES.get(rate_index, ("? MS/s", 1.0))
        return [i/rate for i in range(num_points)], rate_label


    def set_num_channels(self, nchannels, timeout=0):
        """
        Set the number of active channels.  Either we sample only CH1 or we
        sample CH1 and CH2.
        :param nchannels: The number of active channels.  This is 1 or 2.
        :param timeout: (OPTIONAL).
        :return: True if successful. This method may assert or raise various libusb
errors if something went wrong.
        """
        if self.supports_single_channel:
            assert nchannels == 1 or nchannels == 2
            if not self.device_handle:
                assert self.open_handle()
            bytes_written = self.device_handle.controlWrite(0x40,
self.SET_NUMCH_REQUEST,
                                                            self.SET_NUMCH_VALUE,
                                                            self.SET_NUMCH_INDEX,
                                                            pack("B", nchannels),
timeout=timeout)
```

```
            assert bytes_written == 0x01
            self.num_channels = nchannels
            return True
        else:
            return False


    def set_ch1_voltage_range(self, range_index, timeout=0):
        """
        Set the voltage scaling factor at the scope for channel 1 (CH1).
        :param range_index: A numerical constant, which determines the devices range
by the following formula:
                            range := +/- 5.0 V / (range_indeX).

                            The stock software only typically uses the range
indicies 0x01, 0x02, 0x05, and 0x0a,
                            but others, such as 0x08 and 0x0b seem to work
correctly.

                            This same range_index is given to the scale_read_data
method to get nicely scaled
                            data in voltages returned from the scope.

        :param timeout: (OPTIONAL).
        :return: True if successful. This method may assert or raise various libusb
errors if something went wrong.
        """
        if not self.device_handle:
            assert self.open_handle()
        bytes_written = self.device_handle.controlWrite(0x40,
self.SET_CH1_VR_REQUEST,
                                                        self.SET_CH1_VR_VALUE,
                                                        self.SET_CH1_VR_INDEX,
                                                        pack("B", range_index),
timeout=timeout)
        assert bytes_written == 0x01
        return True


    def set_ch2_voltage_range(self, range_index, timeout=0):
        """
        Set the voltage scaling factor at the scope for channel 1 (CH1).
        :param range_index: A numerical constant, which determines the devices range
by the following formula:
                            range := +/- 5.0 V / (range_indeX).

                            The stock software only typically uses the range
indicies 0x01, 0x02, 0x05, and 0x0a,
                            but others, such as 0x08 and 0x0b seem to work
correctly.
```

```
                            This same range_index is given to the scale_read_data
method to get nicely scaled
                            data in voltages returned from the scope.

        :param timeout: (OPTIONAL).
        :return: True if successful. This method may assert or raise various libusb
errors if something went wrong.
        """
        if not self.device_handle:
            assert self.open_handle()
        bytes_written = self.device_handle.controlWrite(0x40,
self.SET_CH2_VR_REQUEST,
                                                        self.SET_CH2_VR_VALUE,
                                                        self.SET_CH2_VR_INDEX,
                                                        pack("B", range_index),
timeout=timeout)
        assert bytes_written == 0x01
        return True


    def set_calibration_frequency(self, freq_index, timeout=0):
        """
        Set the frequency of the calibration output.
        :param rate_index: The rate_index. These are the keys for the CAL_FREQUNCY
dict for the Oscilloscope object.
                            Common rate_index values and actual sample rate per
channel:
                           105 <->  50 Hz
                           110 <-> 100 Hz
                           120 <-> 200 Hz
                           150 <-> 500 Hz
                             1 <->   1 kHz
                             2 <->   2 kHz
                             5 <->   5 kHz
                            10 <->  10 kHz
                            20 <->  20 kHz
                            50 <->  50 kHz
                           100 <-> 100 kHz
                            Other values are not supported.
        :param timeout: (OPTIONAL).
        :return: True if successful. This method may assert or raise various libusb
errors if something went wrong.
        """
        if not self.device_handle:
            assert self.open_handle()

        bytes_written = self.device_handle.controlWrite(0x40,
self.SET_CAL_FREQ_REQUEST,
                                                        self.SET_CAL_FREQ_VALUE,
```

```
                                                        self.SET_CAL_FREQ_INDEX,
                                                        pack("B", freq_index),
timeout=timeout)
        assert bytes_written == 0x01
        return True


    def set_ch1_ch2_ac_dc(self, ac_dc, timeout=0):
        """
        set AC_DC status of both channels
        :param ac_dc: the value to send:
        0x00: CH2 AC, CH1 AC
        0x01: CH2 AC, CH1 DC
        0x10: CH2 DC, CH1 AC
        0x11: CH2 DC, CH1 DC
        :param timeout: (OPTIONAL).
        :return: True if successful. This method may assert or raise various libusb
errors if something went wrong.
        """
        assert ac_dc == self.AC_AC or ac_dc == self.AC_DC or ac_dc == self.DC_AC or
ac_dc == self.DC_DC
        if not self.device_handle:
            assert self.open_handle()
        bytes_written = self.device_handle.controlWrite(0x40,
self.SET_AC_DC_REQUEST,
                                                        self.SET_AC_DC_VALUE,
                                                        self.SET_AC_DC_INDEX,
                                                        pack("B", ac_dc),
timeout=timeout)
        assert bytes_written == 0x01
        self.ac_dc_status = ac_dc
        return True


    def set_ch1_ac_dc(self, ac_dc, timeout=0):
        """
        set AC_DC status of channel 1
        :param ac_dc: the value to set:
        0x00: AC
        0x01: DC
        :param timeout: (OPTIONAL).
        :return: True if successful. This method may assert or raise various libusb
errors if something went wrong.
        """
        assert ac_dc == self.AC or ac_dc == self.DC
        if not self.device_handle:
            assert self.open_handle()
        ac_dc_new = ( self.ac_dc_status & 0xF0 ) | ac_dc
        bytes_written = self.device_handle.controlWrite(0x40,
self.SET_AC_DC_REQUEST,
```

```
                                                        self.SET_AC_DC_VALUE,
                                                        self.SET_AC_DC_INDEX,
                                                        pack("B", ac_dc_new),
timeout=timeout)
        assert bytes_written == 0x01
        self.ac_dc_status = ac_dc_new # remember the latest status
        return True


    def set_ch2_ac_dc(self, ac_dc, timeout=0):
        """
        set AC_DC status of channel 2
        :param ac_dc: the value to set:
        0x00: AC
        0x01: DC
        :param timeout: (OPTIONAL).
        :return: True if successful. This method may assert or raise various libusb
errors if something went wrong.
        """
        assert ac_dc == self.AC or ac_dc == self.DC
        if not self.device_handle:
            assert self.open_handle()
        ac_dc_new = ( self.ac_dc_status & 0x0F ) | ac_dc << 4
        bytes_written = self.device_handle.controlWrite(0x40,
self.SET_AC_DC_REQUEST,
                                                        self.SET_AC_DC_VALUE,
                                                        self.SET_AC_DC_INDEX,
                                                        pack("B", ac_dc_new),
timeout=timeout)
        assert bytes_written == 0x01
        self.ac_dc_status = ac_dc_new # remember the latest status
        return True
```