• Remember to register your attendance using the UoB Check-In app. Either

1. download, install, and use the native app[a] available for Android and iOS, or
2. directly use the web-based app available at

https://check-in.bristol.ac.uk

noting the latter is also linked to via the `Attendance` menu item on the left-hand side of the Blackboard-based unit web-site.

• The hardware *and* software resources located in the MVB Linux lab(s). (e.g., MVB-1.15 or MVB-2.11) are managed by the Faculty IT Support Team, a subset of IT Services. If you encounter a problem (e.g., a workstation that fails to boot, an error when you try to use some software, or you just cannot log into your account), they can help: you can contact them, to report then resolve said problem, via

https://www.bristol.ac.uk/it-support

• The lab. worksheet is written *assuming* you work in the lab. using UoB-managed and thus *supported* equipment. If you need or prefer to use your own equipment, however, various *unsupported*[b] alternatives available: for example, *you* could 1) manually install any software dependencies yourself, *or* 2) use the unit-specific Vagrant[c] box by following instructions at

https://cs-uob.github.io/COMS30048/vm

• The purpose of the worksheet is to provide a) a tutorial-style introduction to selected technologies or concepts, and/or b) a means to explore them via hands-on tasks and challenges. Note that the worksheet is not assessed *at all*: if you are confident that you already understand the content, there is no problem with nor penalty for totally ignoring it.

• Keep in mind that various mechanisms exist to get support with and/or feedback on your work; these include both in-person (e.g., the lab. slot itself) *and* online (e.g., the unit forum, accessible via the unit web-site) instances.

[a] https://www.bristol.ac.uk/students/support/it/software-and-online-resources/registering-attendance

[b] The implication here is that such alternatives are provided in a best-effort attempt to help you: they are experimental, and so *no* guarantees about nor support for their use will be offered.

[c] https://www.vagrantup.com

COMS30048 lab. worksheet #1.2

Before you start work, download (and, if need be, unarchive[a]) the file

`https://assets.phoo.org/COMS30048_2025_TB-2/csdsp/sheet/lab-01-2.tar.gz`

somewhere secure[b] in your file system; from here on, we assume `${ARCHIVE}` denotes a path to the resulting, unarchived content. The archive content is intended to act as a starting point for your work, and will be referred to in what follows.

[a]For example, you could 1) use `tar`, e.g., by issuing the command `tar xvfz lab-01-2.tar.gz` in a terminal window, 2) use `ark` directly: use the `Activities` desktop menu item, search for and execute `ark`, use the `Archive`→`Open` menu item to open `lab-01-2.tar.gz`, then extract the contents via the `Extract` button, or 3) use `ark` indirectly: use the `Activities` desktop menu item, search for and execute `dolphin`, right-click on `lab-01-2.tar.gz`, select `Open with`, select `ark`, then extract the contents via the `Extract` button.

[b]For example, the `Private` sub-directory within your home directory (which, by default, cannot be read by another user).

1. Introduction

In essence, an oscilloscope[1] (or "scope") is an example of electronic test equipment: such devices allow inspection of signals as they change over time, and so represent an important way to test, analyse, and debug electronic systems.

Given the remit of this unit, an in-depth understanding of oscilloscopes (including electronics-related terminology and concepts) is definitively *not* required. However, EEVblog[2] offers a great high-level, and typically high-energy, introduction at

`http://www.youtube.com/watch?v=Iq4QlfH-oqk`

In short, and somewhat roughly, there are two main types of oscilloscope, namely the Cathode-Ray Oscilloscope (CRO) and the now more common Digital Storage Oscilloscope (DSO)[3], which differ significantly in terms of their design and thus function. Focusing on DSOs, the idea is that a signal of interest will be sampled periodically via an Analogue to Digital Converter (ADC); the samples are stored as time series data for later analysis. As a result, we can use the DSO to acquire a trace, or sequence, e.g.,

$$\Lambda = \langle \Lambda_0, \Lambda_1, \ldots, \Lambda_{l-1} \rangle,$$

where each Λ_i is a sample. Various characteristics of a given DSO impact this process, and therefore the resulting trace. For example, the maximum sampling rate limits how rapidly the ADC can sample the signal (i.e., the gap in time between a given Λ_i and Λ_{i+1}), whereas the ADC resolution relates to the accuracy of each sample (i.e., the range of values any Λ_i can take after quantisation[4]).

The SCALE kit includes a DSO, which can be used to analyse behaviour of the SCALE development board: the goal of this lab. worksheet is to explain how, i.e, to a) introduce the PicoScope 2206B oscilloscope, then b) explore how to make use of it, both manually and programmatically.

Note that although the lab. worksheet does conclude with a set of hands-on tasks and challenges, at this point they are intentionally biased towards reading and understanding the material (vs. more active alternatives, e.g., programming). It is *crucial* not to view this as optional effort: carefully working through the admittedly detailed content should allow you to more easily and rapidly engage with longer-term challenges (e.g., those relating to a given coursework assignment).

2. An overview: the PicoScope 2206B oscilloscope

2.1. Hardware

The SCALE kit includes a Pico Technology[5] PicoScope 2206B oscilloscope. As a low(er)-end DSO, it a) has a limited specification with respect to criteria such as maximum sampling rate, b) omits various more advanced

[1]`http://en.wikipedia.org/wiki/Oscilloscope`

[2]See the web-page at `http://www.eevblog.com` or YouTube channel at `http://www.youtube.com/channel/UC2DjFE7Xf11URZqWBigcVOQ`. In particular, the playlist `http://www.youtube.com/playlist?list=PL3C5D963B695411B6` offers an excellent primer on various topics electronics *if* you are interested in more detail.

[3]`http://en.wikipedia.org/wiki/Digital_storage_oscilloscope`

[4]`http://en.wikipedia.org/wiki/Quantization_(signal_processing)`

[5]`http://www.picotech.com`, see also `http://www.youtube.com/channel/UC7ZYc00CrHgaxeHZGp3aQ9Q`

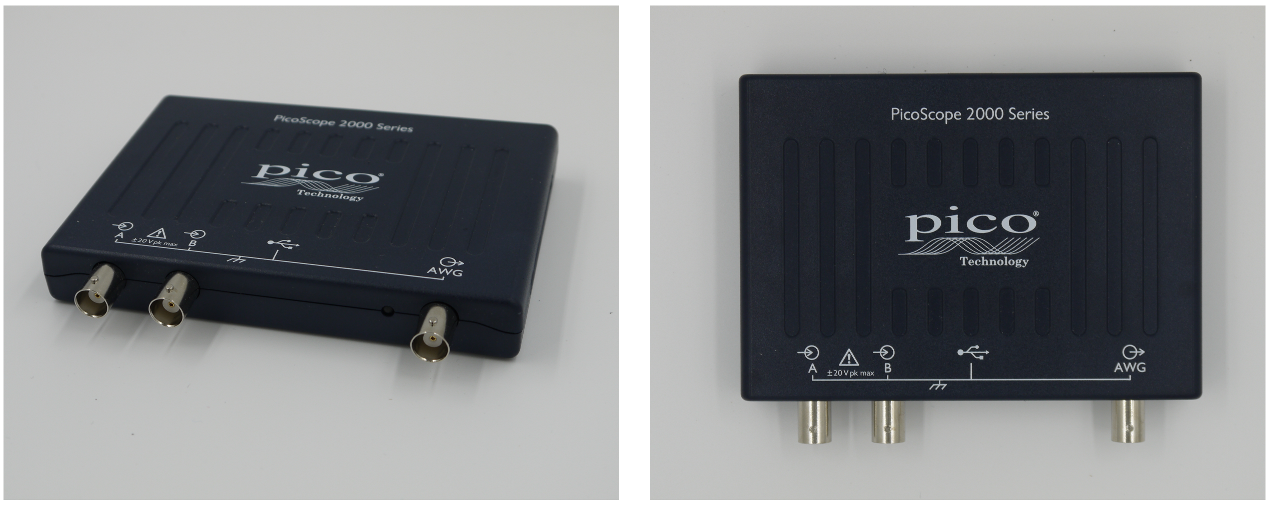


Figure 1: *A physical illustration of the PicoScope 2206B oscilloscope.*

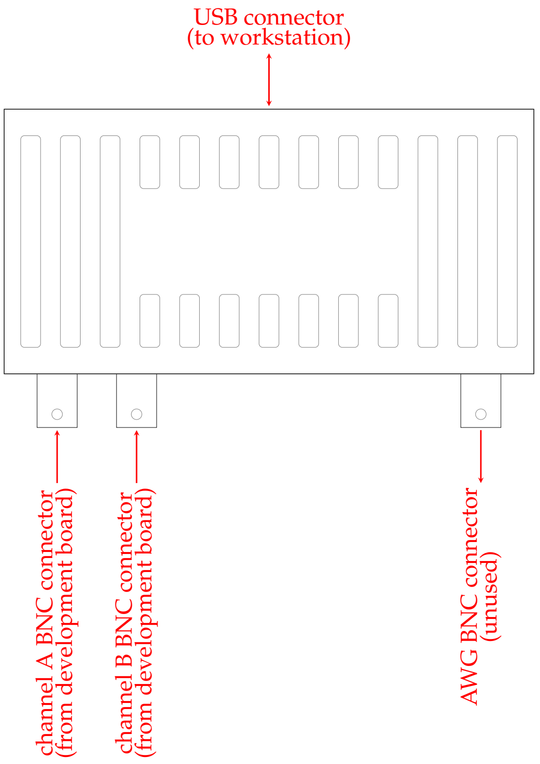


Figure 2: *A logical illustration of the PicoScope 2206B oscilloscope.*

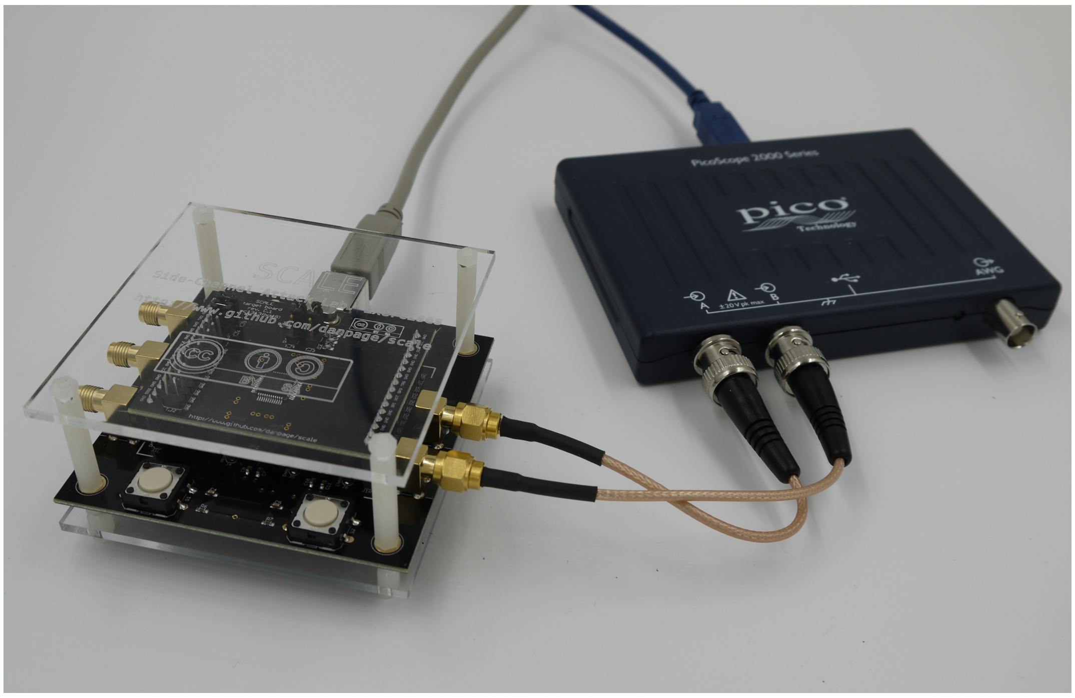


Figure 3: *A physical illustration of an automated "bulk" acquisition workflow, involving the SCALE development board and PicoScope 2206B oscilloscope.*

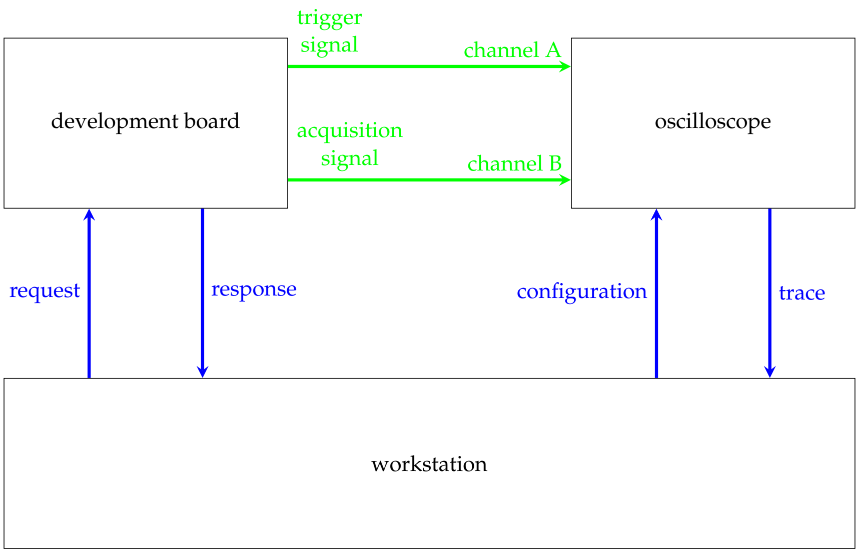


Figure 4: *A logical illustration of an automated "bulk" acquisition workflow, involving the SCALE development board and PicoScope 2206B oscilloscope; USB-to-USB connections are shown in blue, SMA-to-BNC connections are shown in green.*

forms of functionality, and c) omits any display or processing capability. Per Figure 1, the last feature means the 2206B might not match your expectation of what an oscilloscope should look like: the idea is to treat it as a peripheral device for some control workstation, with traces transferred from the former to the latter for analysis once acquired. This approach, which positions the 2206B as a PC-based DSO, means it can be low(er)-cost, and, crucially, portable and robust.

The 2206B datasheet [2, Pages 14+15] has a detailed specification. With reference to Figure 2, a set of pertinent features and functionality are summarised by the following:

- The 2206B has two channels (i.e., can sample two signals), each of which has a 50MHz bandwidth and 8-bit ADC resolution; when both channels are used, the maximum sampling rate is 250MS s^{-1}, with samples stored in a 32MS buffer before transfer to the associated workstation.
- Although we do not make use of it, the 2206B also has an Arbitrary Waveform Generator (AWG) function. This means the right-most BNC connector is used to *generate* (versus *sample*) signals, including various standard forms such as square or “saw tooth”.
- Rather than continuous acquisition, the 2206B can be configured to acquire a trace based on a trigger. The idea is basically that it can wait until some condition is satisfied, and only then start to acquire data; various conditions can be specified, the most useful of which is arguably detection of a positive (or rising) edge on a given channel.

Note that Pico Technology maintain detailed reference material[6] for their oscilloscopes, and for oscilloscopes in general, which may help to understand some of the above in more detail if/when need be.

2.2. Software

There are two main ways to interact with the 2206B, namely via

- the PicoScope application[7] which offers a traditional, oscilloscope-like user interface, or
- the PicoScope programming API [3]; bindings exist for various programming languages, including Python and C.

Although the PicoScope application is ideal for interactive use of the 2206B, programmatic use via the API is preferable if “bulk” acquisition of traces is required; a control workstation can then be used to coordinate and so automate the process. Figure 4 illustrates the associated workflow, where the annotated steps involved are as follows:

1. the workstation configures the 2206B so a trace is acquired once a positive edge on the trigger signal is detected,
2. the workstation sends a request (or command) to the target device, including any input data x,
3. the target device
   (a) sets the trigger signal to 1 (i.e., produces a positive edge),
   (b) executes the target functionality to compute $r = f(x)$,
   (c) sets the trigger signal to 0 (i.e., produces a negative edge),
4. the target device sends a response to the workstation, including any output data r,
5. the workstation retrieves the acquired trace from the 2206B.

3. Some hands-on tasks and challenges

The following Sections provide a tutorial-style introduction to use of the 2206B. Using a terminal[8] to execute the commands, you should be able to progress step-by-step to at *least* the point where you have acquired a trace from the 2206B; this trace will reflect the power consumption and hence behaviour of the development board while executing an example program.

Keep in mind that although you can simply copy-and-paste the commands, it remains important to understand their purpose. The short-term goal is to equip you with experience of use in the long(er)-term: if a given step, concept, or detail is unclear, *now* is the time to ask questions.

3.1. Prepare the hardware

1. Take a coaxial cable, and connect the trigger signal connector on the development board to the channel A connector on the 2206B.

[6]See, e.g., http://www.picotech.com/library

[7]Version 6 [4] of the PicoScope application is the most recent release, but, in the medium- to long-term, version 7 (which is currently in an early-access phase) will replace it: we focus on the former, but, for our use-case, the details will easily translate to either (i.e., few details are specific to either version).

[8]That is, within a BASH shell (or prompt, e.g., a terminal window) or similar.

2. Take a coaxial cable, and connect the acquisition signal connector on the development board to the channel B connector on the 2206B.
3. Take the (grey) USB cable, and connect the development board to the workstation you are using; you should see the power status LED lit.
4. The kernel *should* recognise a new device[9] as having been connected, and then create a device node in order to interact with it. You can check this via

   ```
   ls -l /dev/scale-board
   ```

   with any error (e.g., `cannot access '/dev/scale-board': No such file or directory`) suggesting that the device has not been recognised: ask for help!
5. Take the (blue) USB cable, and connect the 2206B to the workstation you are using; you should see the status LED lit.
6. The kernel *should* recognise a new device as having been connected, and then create a device node in order to interact with it. You can check this via

   ```
   ls -l /dev/scale-scope
   ```

   with any error (e.g., `cannot access '/dev/scale-scope': No such file or directory`) suggesting that the device has not been recognised: ask for help!

3.2. Prepare the software

Either replicate steps to reproduce, or directly reuse[10] the BSP built in the analogous Section from lab. worksheet #1.1.

3.3. Prepare then execute an example program

Figure 5 captures an example program for the development board. The source code is heavily annotated to describe each major feature or step; it realises a form of benchmark, by iterating over the execution of three instruction classes (or kernels) with the aim of highlighting pertinent differences between them.

1. Fix the working directory:

   ```
   cd ${ARCHIVE}/board
   ```

2. Build any material related to the example:

   ```
   make PROJECT="benchmark" build
   ```

3. Following the same process as lab. worksheet #1.1, initiate the programming process by executing

   ```
   make PROJECT="benchmark" program
   ```

   then perform the subsequent manual steps. Once the process is complete, the program should start to execute: as the source code in Figure 5 suggests, the only perceivable activity will be (very) short periods where the trigger status LED is on (during which the kernels are being executed).

3.4. Acquire a trace of power consumption

3.4.1. Acquire via application

1. Fix the working directory:

   ```
   cd ${ARCHIVE}/scope
   ```

2. Execute the PicoScope application:

   ```
   make acquire-app
   ```

3. Acquire the trace using the PicoScope application:

   - press the stop button to stop sampling,
   - set channel A to ±5V,
   - set channel B to ±500mV,

[9] `lsusb` lists the device as a FTDI FT232R [1], which is a component tasked with supporting UART-based communication (by the target board) over a USB-based connection (on the host board).

[10] Take care to (re)set any associated environment variables, e.g., `${REPO}` and `${TARGET}`.

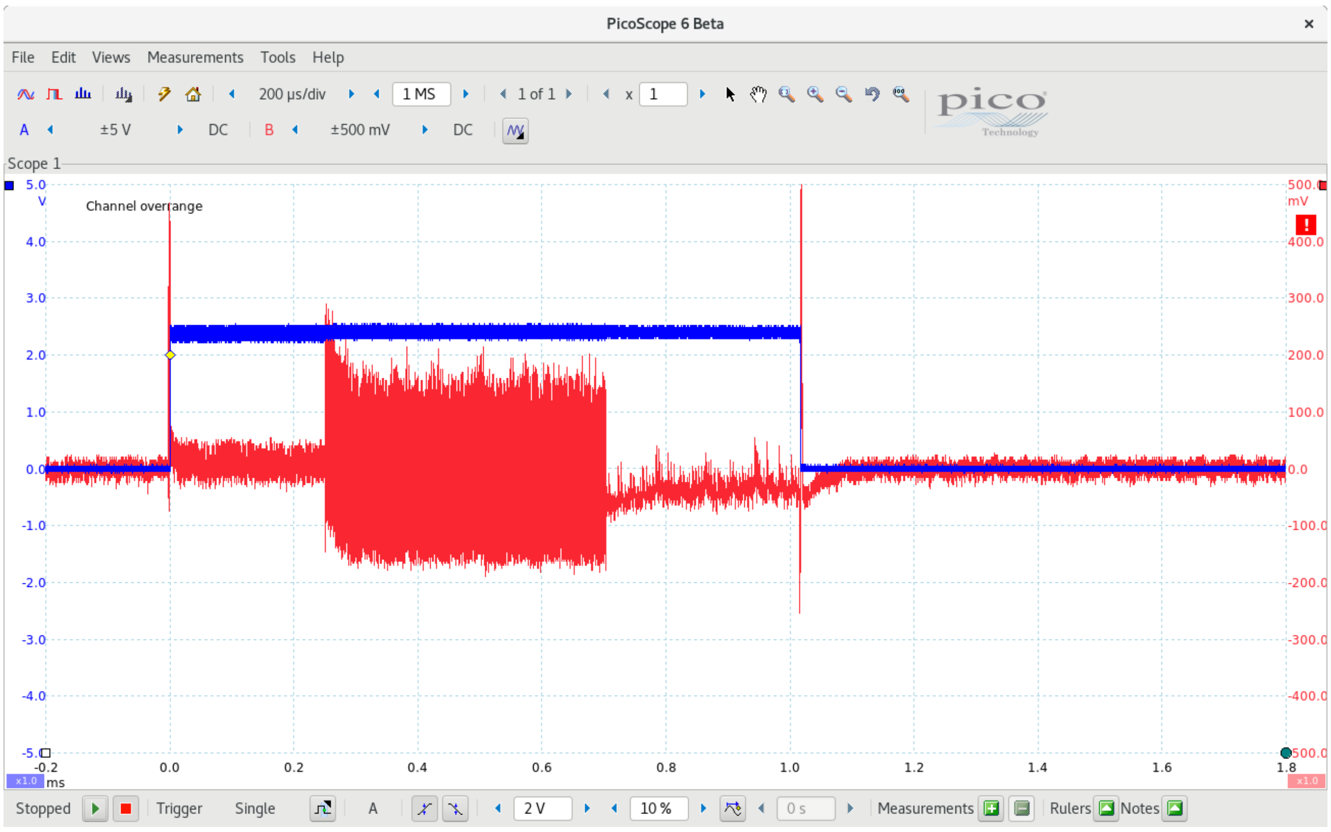
```
#ifndef __BENCHMARK_H
#define __BENCHMARK_H

#include <scale/scale.h>

#endif
```

(a) *${ARCHIVE}/board/benchmark.h*

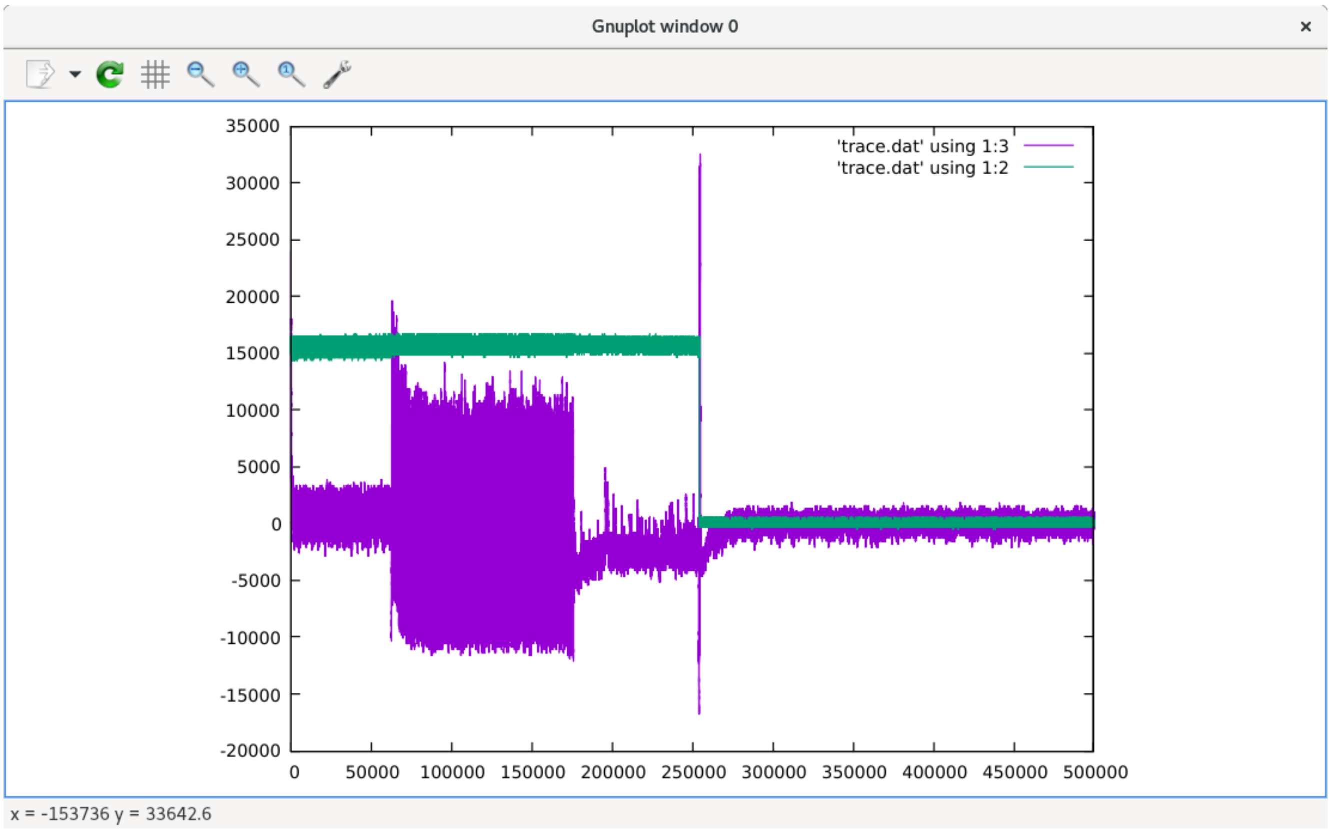
```
#include "benchmark.h"

/** Initialise the SCALE development board, then loop indefinitely.  Each loop
  * iteration will execute 3 benchmark kernels, namely
  *
  * 1. 1000 nop (or   no-operation) instructions,
  * 2. 1000 ldr (or           load) instructions, and
  * 3. 1000 mul (or multiplication) instructions.
  *
  * before the kernels there is a 1000 ms separating delay, and around them the
  * trigger signal (and thus status LED) is toggled on and off.
  */

int main( int argc, char* argv[] ) {
  // select a configuration st. the external 16 MHz oscillator is used
  scale_conf_t scale_conf = {
    .clock_type        = SCALE_CLOCK_TYPE_EXT,
    .clock_freq_source = SCALE_CLOCK_FREQ_16MHZ,
    .clock_freq_target = SCALE_CLOCK_FREQ_16MHZ,

    .tsc               = false
  };

  // initialise the development board
  if( !scale_init( &scale_conf ) ) {
    return -1;
  }

  uint32_t A[ 1024 ], r;

  while( true ) {
    // delay for 1000 ms = 1 s
    scale_delay_ms( 1000 );

    // write the trigger pin, and hence LED    : TRG <- 1 (positive edge)
    scale_gpio_wr( SCALE_GPIO_PIN_TRG, true  );

    // execute kernel #1: 1000 * nop (or "no operation")
    for( int i = 0; i < 1000; i++ ) {
      __asm__ __volatile__( "nop"                                             );
    }
    // execute kernel #2: 1000 * ldr (or "indexed load from memory into register")
    for( int i = 0; i < 1000; i++ ) {
      __asm__ __volatile__( "ldr %0, [%1,%2]" : "=&l" (r) : "l" (&A), "l" (i) );
    }
    // execute kernel #3: 1000 * mul (or "integer multiplication")
    for( int i = 0; i < 1000; i++ ) {
      __asm__ __volatile__( "mul %0, %1, %2"  : "=&l" (r) : "l"  (i), "l" (i) );
    }

    // write the trigger pin, and hence LED    : TRG <- 0 (negative edge)
    scale_gpio_wr( SCALE_GPIO_PIN_TRG, false );
  }

  return 0;
}
```

(b) *${ARCHIVE}/board/benchmark.c*

Figure 5: *An example program for the SCALE development board.*



(a) *PicoScope application.*



(b) *(Python-based) PicoScope API.*

Figure 6: *Example traces acquired by the PicoScope 2206B during execution of the example program in Figure 5 on the attached development board.*

- set the collection time to 200µs/div,
- set the trigger mode to single,
- set the trigger channel to A,
- set the trigger direction to positive edge,
- set the trigger threshold to 2V,
- set the pre-trigger period to 10%, then, finally,
- press the start button to start sampling,

taking care throughout with respect to the UI, which is less than ideal[11] in certain areas.

4. Inspect the trace: it should be *similar* to Figure 6a.

- The blue waveform is channel A, the trigger signal: it transitions from ~ 0V to ~ 2.5V at ~ 0ms (a positive edge), and then from ~ 2.5V to ~ 0V at ~ 1ms (a negative edge). Per the configuration above, the first transition prompts the 2206B to then start acquiring the trace.
- The red waveform is channel B, the acquisition signal. The signal exhibits a different form during each of three distinct periods; these directly reflect the execution of the three instruction classes, demonstrating, for example, that a memory access can be distinguished from an integer multiplication.

3.4.2. Acquire via API

As alluded to earlier, we can automate the manual process of acquiring traces: to do so, we implement a program that interacts with the 2206B via the associated API. Within said implementation we employ block mode acquisition [3, Section 2.7.1], so roughly mirror the workflow outlined in Section 2.2.

Although a C-based implementation is also included, a Python-based[12] implementation is shown in Figure 7; we focus on this as an exemplar, because it is more concise (and arguably "cleaner") and illustrates the concepts more clearly. The source code can be viewed as capturing two phases, namely 1) the acquisition of data from the 2260B into memory, and 2) the storage of data from memory into a file; in the former case, it is heavily annotated to reference associated documentation in the API [3] and steps in the exemplar block mode acquisition process [3, Section 2.7.1.1]. Beyond this, the concept of a timebase is important: this has specific documentation [3, Section 2.8] to explain it.

1. Fix the working directory:

```
cd ${ARCHIVE}/scope
```

2. Acquire the trace using *either* the Python-based

```
make acquire-api-py
```

or the C-based

```
make acquire-api-c
```

PicoScope API.

3. Plot the trace:

```
make plot
```

Note that an intentionally simple Gnuplot[13] script is used to achieve this: it is clearly possible to improve the resulting visualisation.

4. Inspect the trace: it should be *similar* to Figure 6b.

[11] Although the Windows-based version is better, the Linux-based UI has some annoying "gotchas" to keep in mind; on the whole, you get used to these and so can easily avoid them with practice. For example: 1) the UI seems to get confused if settings are altered while sampling: per the above it is more reliable to stop sampling, then update the settings, then start sampling, 2) when editing a setting via a text field, the UI will often ignore your edit (i.e., not update the setting to match the edited text) *until* you press return to register it.

[12] The Python-based implementation uses a binding for the PicoScope API, namely http://www.github.com/colinoflynn/pico-python; note that an officially supported alternative, namely https://github.com/picotech/picosdk-python-wrappers now exists, but we deem it harder to use and so prefer the former.

There are some notable differences between the C and Python bindings, with the latter typically more high level, and so user friendly than the former. For example, the C binding demands use of machine-friendly timebases, whereas the Python binding as a human-friendly mechanism for setting the acquisition duration, using a sample period and duration measured in seconds. This means that although the two implementations provided do differ in form, some care has been taken to translate consistently between them so some equivalence is evident. However, one side-effect is that they use a "lowest common denominator" approach: functionality available in one but not both is ignored. For example, both the C binding and the Python binding use raw, machine-friendly ADC values despite the fact that the Python binding *can* automatically convert these to human-friendly voltage levels (via `getDataV`, versus `getDataRaw` as is) based on the current channel configuration.

[13] http://www.gnuplot.info

```
import picoscope.ps2000a as ps2000a, sys, time

PS2000A_RATIO_MODE_NONE      = 0 # Section 3.18.1
PS2000A_RATIO_MODE_AGGREGATE = 1 # Section 3.18.1
PS2000A_RATIO_MODE_DECIMATE  = 2 # Section 3.18.1
PS2000A_RATIO_MODE_AVERAGE   = 4 # Section 3.18.1

class Scope( object ) :
  def __init__( self, open = True ) :
    if ( open ) :
      self.open()

  def  open( self ) :
    self.scope = ps2000a.PS2000a( connect = True )

  def close( self ) :
    self.scope.close()

  def adc2volts( self, range, x ) :
    return ( float( x ) / float( self.scope.getMaxValue() ) ) * range;

  def volts2adc( self, range, x ) :
    return ( float( x ) * float( self.scope.getMaxValue() ) ) / range;

  def acquire_prime( self ) :
    # Section 3.39, Page 69; Step  2: configure channels
    self.scope.setChannel( channel = 'A', enabled = True, coupling = 'DC', VRange =   5.0E-0 )
    self.scope.setChannel( channel = 'B', enabled = True, coupling = 'DC', VRange = 500.0E-3 )

    # Section 3.13, Page 36; Step  3: configure timebase
    ( _, samples, samples_max ) = self.scope.setSamplingInterval( 4.0E-9, 2.0E-3 )

    # Section 3.56, Page 93; Step  4: configure trigger
    self.scope.setSimpleTrigger( 'A', threshold_V = 2.0E-0, direction = 'Rising', timeout_ms = 0 )

    # Section 3.37, Page 65; Step  5: start acquisition
    self.scope.runBlock()

  def acquire_fetch( self ) :
    # Section 3.26, Page 54; Step  6: wait for acquisition to complete
    self.scope.waitReady()

    # Section 3.40, Page 71; Step  7: configure buffers
    # Section 3.18, Page 43; Step  8: transfer  buffers
    ( A, _, _ ) = self.scope.getDataRaw( channel = 'A', downSampleMode = PS2000A_RATIO_MODE_NONE )
    ( B, _, _ ) = self.scope.getDataRaw( channel = 'B', downSampleMode = PS2000A_RATIO_MODE_NONE )

    # Section 3.2,  Page 25; Step 10: stop  acquisition
    self.scope.stop()

    return ( A, B )

if ( __name__ == '__main__' ) :
  scope = Scope()

  # Follow Section 2.7.1.1 of the 2206B programming guide, i.e., use a 1-shot
  # block mode acquisition process: doing so configures the 2206B to
  #
  # - wait for a trigger signal (a positive edge exceeding 2 V) on channel A,
  # - sample from both channel A and B, using appropriate voltage ranges and
  #   for an appropriate amount of time (i.e., ~2 ms),
  # - store the resulting data in buffers with no post-processing (e.g., with
  #   no downsampling).

  scope.acquire_prime() ; ( A, B ) = scope.acquire_fetch()

  # Store the acquired data from channels A and B into a CSV-formated file,
  # truncating the data as necessary to deal with any length mismatch.

  with open( sys.argv[ 1 ], 'w' ) as fd :
    for ( i, ( A_i, B_i ) ) in enumerate( zip( A, B ) ) :
      print( '{0:d}, {1:d}, {2:d}'.format( i, A_i, B_i ), file = fd )

  scope.close()
```

Figure 7: *${ARCHIVE}/scope/acquire.py*

3.5. Develop and experiment with the acquisition process

1. Although the 2206B can be configured to *start* acquisition of data based on a trigger, an analogous mechanism to *stop* said acquisition does not exist. The example in Figure 6 demonstrates this: not all samples (from channel B) are within the trigger period (where channel A exceeds 2.5V). Using either the C- or Python-based block acquisition example as a starting point, implement a post-processing phase that "trims" the trace such that only the trigger period is produced as output.
2. The example in Figure 6 demonstrates the fact that some instruction classes can be distinguished from other classes. Another feature of interest is the data those instructions operate on: can you show, for example, that an instruction operating on high Hamming weight data can be distinguished from low Hamming weight data?

References

[1] *FT232R USB UART IC*. Tech. rep. FT000053. Future Technology Devices International Ltd., 2015. URL: http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT232R.pdf (see p. 6).

[2] *PicoScope 2000 Series: Data Sheet*. Tech. rep. MM071.en-2. Pico Technology Ltd., 2021. URL: https://www.picotech.com/download/datasheets/picoscope-2000-series-data-sheet-en.pdf (see p. 5).

[3] *PicoScope 2000 Series: Programmer's Guide*. Tech. rep. ps2000apg.en-12. Pico Technology Ltd., 2022. URL: http://www.picotech.com/download/manuals/picoscope-2000-series-a-api-programmers-guide.pdf (see pp. 5, 9).

[4] *PicoScope 6 PC Oscilloscope Software: User's Guide*. Tech. rep. psw.en r46. Pico Technology Ltd., 2017. URL: http://www.picotech.com/download/manuals/picoscope-6-users-guide.pdf (see p. 5).

A Frequently Asked Questions (FAQs)

The `/dev/scale-scope` device node doesn't seem to exist; the device is not recognised. This suggests either 1) a problem with the hardware itself, and/or 2) a problem with configuration of associated software (e.g., driver); even if you did not explicitly check the device node, you *may* have inferred the device is not recognised because use of it (e.g., via the Python- or the C-based acquisition examples, or the PicoScope application) failed somehow. As some first steps, check

1. the USB cable is corrected plugged in,
2. the power LED is illuminated,
3. whether power cycling[14] the device (i.e., "turn it off and on again" by removing and reinserting the USB cable).

If the above fail, the cause is something more subtle or significant: ask for help in the lab.

The oscilloscope is making a clicking noise?! The noise stems from it switching constituent relay[15] components between states, in order to turn parts of the oscilloscope front-end on or off. This happens when the oscilloscope configuration is changed, e.g., when a channel is enabled, or the voltage range for a channel is altered. If the clicking is quite rapid, this hints at misuse, e.g., accidental reconfiguration in a loop; otherwise it's expected behaviour, so there is nothing to worry about.

My file system quota, or available (physical) memory is all used up! The underlying challenge here is that the size of data sets related to use of the 2206B can be significant. To address this challenge, it is important to carefully manage both the in-memory and on-disk footprint of such data sets. For example, take care to

- use a sampling frequency and duration that is bounded, to limit the data set size,
- use a space-efficient in-memory and on-disk data structures for said data set (e.g., a binary vs. text format for the latter),
- actively manage memory allocation, even in cases where a garbage collector is available.

[14] http://en.wikipedia.org/wiki/Power_cycling
[15] http://en.wikipedia.org/wiki/Relay