PicoScope® 2000 Series (A API)

PC Oscilloscopes and MSOs

Programmer's Guide

ps2000apg.en-12

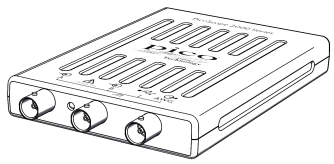
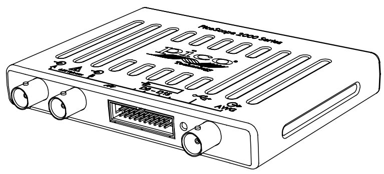
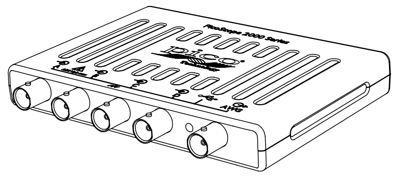
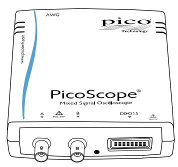
Contents

1 Introduction

1.1 Overview

The **PicoScope 2000 Series PC Oscilloscopes** from Pico Technology are high-speed real-time measuring instruments. They obtain their power from the USB port so do not need an additional power supply. With a built-in arbitrary waveform generator, these scopes contain everything you need in a convenient, portable unit.

This manual explains how to develop your own programs for collecting and analyzing data from the PicoScope 2000 Series oscilloscopes. It applies to all devices supported by the ps2000a application programming interface (API), as listed below:

<table>
<tr><th>2-channel</th><th colspan="2">2-channel MSO</th><th>4-channel</th></tr>
<tr><td></td><td colspan="2"></td><td></td></tr>
<tr><td rowspan="2">PicoScope 2206
PicoScope 2206A
PicoScope 2206B
PicoScope 2207
PicoScope 2207A
PicoScope 2207B
PicoScope 2208
PicoScope 2208A
PicoScope 2208B</td><td colspan="2">PicoScope 2205A MSO
PicoScope 2206B MSO
PicoScope 2207B MSO
PicoScope 2208B MSO</td><td rowspan="2">PicoScope 2405A
PicoScope 2406B
PicoScope 2407B
PicoScope 2408B</td></tr>
<tr><td></td><td>PicoScope
2205 MSO</td></tr>
</table>

The Pico Software Development Kit (PicoSDK) is available free of charge from www.picotech.com/downloads. This download includes support for all PicoScope oscilloscopes including the ps2000a API described in this manual, as well as the original ps2000 API for older oscilloscopes in the PicoScope 2000 Series.

Example code is available from repositories under the "picotech" organization on GitHub.

SDK version: 10.6.12

1.2 PC requirements

To ensure that your **PicoScope 2000 Series PC Oscilloscope** operates correctly with the SDK, you must have a computer with at least the minimum system requirements to run one of the supported operating systems, as shown in the following table. The performance of the oscilloscope will be better with a more powerful PC, and will benefit from a multi-core processor.

Item	Specification
Operating system	Windows 7, 8 or 10 32-bit or 64-bit
Processor **Memory** **Free disk space**	As required by Windows
Ports*	USB 2.0 or USB 3.0 port USB 1.1 port (absolute minimum)

* PicoScope oscilloscopes will operate slowly on a USB 1.1 port. Not recommended.
USB 3.0 connections will run at about the same speed as USB 2.0.

1.3 Legal information

The material contained in this release is licensed, not sold. Pico Technology Limited grants a licence to the person who installs this software, subject to the conditions listed below.

Access. The licensee agrees to allow access to this software only to persons who have been informed of these conditions and agree to abide by them.

Usage. The software in this release is for use only with Pico products or with data collected using Pico products.

Copyright. Pico Technology Ltd. claims the copyright of, and retains the rights to, all material contained in this SDK. You may copy and distribute the SDK without restriction, as long as you do not remove any Pico Technology copyright statements.

Liability. Pico Technology and its agents shall not be liable for any loss, damage or injury, howsoever caused, related to the use of Pico Technology equipment or software, unless excluded by statute.

Fitness for purpose. As no two applications are the same, Pico Technology cannot guarantee that its equipment or software is suitable for a given application. It is your responsibility, therefore, to ensure that the product is suitable for your application.

Mission-critical applications. This software is intended for use on a computer that may be running other software products. For this reason, one of the conditions of the licence is that it excludes use in mission-critical applications, for example life support systems.

Viruses. This software was continuously monitored for viruses during production, but you are responsible for virus-checking the software once it is installed.

Support. If you are dissatisfied with the performance of this software, please contact our technical support staff, who will try to fix the problem within a reasonable time. If you are still dissatisfied, please return the product and software to your supplier within 14 days of purchase for a full refund.

Upgrades. We provide upgrades, free of charge, from our web site at www.picotech.com. We reserve the right to charge for updates or replacements sent out on physical media.

Trademarks. Windows is a trademark or registered trademark of Microsoft Corporation. Pico Technology Limited and PicoScope are internationally registered trademarks.

2 Concepts

2.1 Driver

Your application will communicate with a PicoScope 2000 (A API) driver called `ps2000a.dll`, which is supplied in 32-bit and 64-bit versions. The driver exports the ps2000a function definitions in standard C format, but this does not limit you to programming in C. You can use the API with any programming language that supports standard C calls.

The API driver depends on another DLL, `picoipp.dll` (which is supplied in 32-bit and 64-bit versions) and a low-level driver called `WinUsb.sys`. These are installed by the SDK and configured when you plug the oscilloscope into each USB port for the first time. Your application does not call these drivers directly.

2.2 General procedure

A typical program for capturing data consists of the following steps:

1. Open the scope unit.
2. Set up the input channels with the required voltage ranges and coupling type.
3. Set up triggering.
4. Start capturing data. (See Sampling modes, where programming is discussed in more detail.)
5. Wait until the scope unit is ready.
6. Copy data to a buffer.
7. Stop capturing data.
8. Close the scope unit.

Many example programs are available on GitHub. These demonstrate how to use the functions of the driver software in each of the modes available.

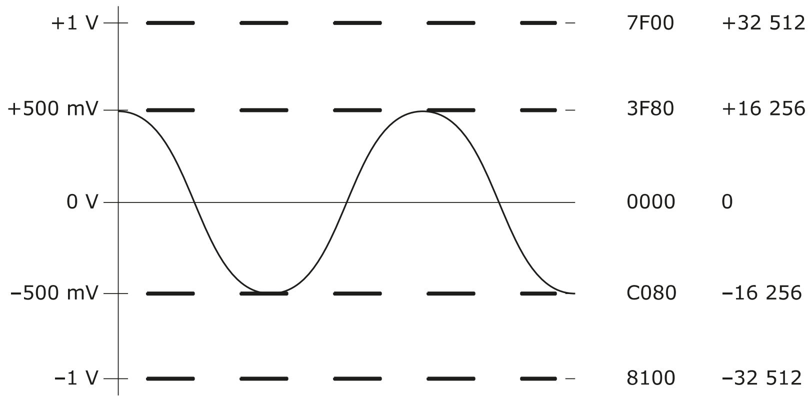
2.3 Voltage ranges

Analog input channels
You can set a device input channel to any voltage range from ±20 mV to ±20 V (subject to the device specification) with `ps2000aSetChannel()`. Each sample is scaled to 16 bits, and the minimum and maximum values returned to your application are given by `ps2000aMinimumValue()` and `ps2000aMaximumValue()` as follows:

Function	Voltage	Value returned	
		decimal	hex
`ps2000aMaximumValue()`	maximum	32 512	7F00
	zero	0	0000
`ps2000aMinimumValue()`	minimum	−32 512	8100

Example

1. Call `ps2000aSetChannel()` with `range` set to `PS2000A_1V`.
2. Apply a sine wave input of 500 mV amplitude to the oscilloscope.
3. Capture some data using the desired sampling mode.
4. The data will be encoded as shown opposite.



External trigger input (PicoScope 2206, 2207 and 2208 only)
The external trigger input (marked **EXT**) is scaled to a 16-bit value as follows:

Voltage	Constant	Digital value
-5 V	`PS2000A_EXT_MIN_VALUE`	−32 767
0 V		0
+5 V	`PS2000A_EXT_MAX_VALUE`	+32 767

2.4 MSO digital data

This section applies to mixed-signal oscilloscopes (MSOs) only

A PicoScope MSO has two 8-bit digital ports—PORT0 and PORT1—containing a total of 16 digital channels.

The data from each port is returned in a separate buffer that is set up by the ps2000aSetDataBuffer() and ps2000aSetDataBuffers() functions. For compatibility with the analog channels, each buffer is an array of 16-bit words. The 8-bit port data occupies the lower 8 bits of the word, and the upper 8 bits of the word are undefined.

	PORT1 buffer	PORT0 buffer
Sample_0	$[\text{XXXXXXXX,D15...D8}]_0$	$[\text{XXXXXXXX,D7...D0}]_0$
...	...	...
Sample_{n-1}	$[\text{XXXXXXXX,D15...D8}]_{n-1}$	$[\text{XXXXXXXX,D7...D0}]_{n-1}$

Retrieving stored digital data
The following C code snippet shows how to combine data from the two 8-bit ports into a single 16-bit word and then extract individual bits from the 16-bit word.

```c
// Mask Port 1 values to get lower 8 bits
portValue = 0x00ff & appDigiBuffers[2][i];

// Shift by 8 bits to place in upper 8 bits of 16-bit word
portValue <<= 8;

// Mask Port 0 values to get lower 8 bits and apply bitwise
// inclusive OR to combine with Port 1 values
portValue |= 0x00ff & appDigiBuffers[0][i];

for (bit = 0; bit < 16; bit++)
{
  // Shift value (32768 - binary 1000 0000 0000 0000),
  // AND with value to get 1 or 0 for channel.
  // Order will be D15 to D8, then D7 to D0.

  bitValue = (0x8000 >> bit) & portValue? 1 : 0;
}
```


2.5 Triggering

PicoScope oscilloscopes can either start collecting data immediately or be programmed to wait for a trigger event.

For simple trigger setups, call this single function:

- `ps2000aSetSimpleTrigger()`

For more complex trigger setups, call the three individual trigger functions:

- `ps2000aSetTriggerChannelConditions()`
- `ps2000aSetTriggerChannelDirections()`
- `ps2000aSetTriggerChannelProperties()`

A trigger event can occur when one of the signal or trigger input channels crosses a threshold voltage on either a rising or a falling edge. It is also possible to combine two inputs using the logic trigger function.

To set up pulse width, delay and dropout triggers, you can also call the pulse width qualifier function:

- `ps2000aSetPulseWidthQualifier()`

2.6 Sampling modes

PicoScope 2000 Series oscilloscopes can run in various **sampling modes**.

- **Block mode.** In this mode, the scope stores data in internal buffer memory and then transfers it to the PC. When the data has been collected it is possible to examine the data, with an optional downsampling factor. The data is lost when a new run is started in the same segment, the settings are changed, or the scope is powered down.

- **ETS mode.** In this mode, it is possible to increase the effective sampling rate of the scope when capturing repetitive signals. It is a modified form of block mode.

- **Rapid block mode.** This is a variant of block mode that allows you to capture more than one waveform at a time with a minimum of delay between captures. You can use downsampling in this mode if you wish.

- **Streaming mode.** In this mode, data is passed directly to the PC without being stored in the scope's internal buffer memory. This enables long periods of data collection for chart recorder and data-logging applications. Streaming mode supports downsampling and triggering, while providing fast streaming at typical rates of 1 to 10 MS/s, as specified in the data sheet for your device.

In all sampling modes, the driver returns data asynchronously using a callback. This is a call to one of the functions in your own application. When you request data from the scope, you pass to the driver a pointer to your callback function. When the driver has written the data to your buffer, it makes a callback (calls your function) to signal that the data is ready. The callback function then signals to the application that the data is available.

Because the callback is called asynchronously from the rest of your application, in a separate thread, you must ensure that it does not corrupt any global variables while it runs.

For compatibility with programming environments not supporting C-style callback functions, polling of the driver is available in block mode.

2.6.1 Block mode

In **block mode**, the computer prompts a PicoScope 2000 Series oscilloscope to collect a block of data into its internal memory. When the oscilloscope has collected the whole block, it signals that it is ready and then transfers the whole block to the computer's memory through the USB port.

- **Block size.** The maximum number of values depends upon the size of the oscilloscope's memory. The memory buffer is shared between the enabled channels, so if two channels are enabled, each receives half the memory, and if three or four channels are enabled, each receives a quarter of the memory. This partitioning is handled transparently by the driver. The block size also depends on the number of memory segments in use – see `ps2000aMemorySegments()`.

  Note: The PicoScope MSO models behave differently. If only the two analog channels or only the two digital ports are enabled, each receives half the memory. If any combination of one or two analog channels and one or two digital ports is enabled, each receives a quarter of the memory.

- **Sampling rate.** A PicoScope 2000 Series oscilloscope can sample at different rates according to the selected timebase and the combination of enabled channels. See the Timebases section for the specifications that apply to your scope model.

- **Setup time.** The driver normally performs a number of setup operations, which can take up to 50 milliseconds, before collecting each block of data. If you need to collect data with the minimum time interval between blocks, use rapid block mode and avoid calling setup functions between calls to `ps2000aRunBlock()`, `ps2000aStop()` and `ps2000aGetValues()`.

- **Downsampling.** When the data has been collected, you can set an optional downsampling factor and examine the data. Downsampling is a process that reduces the amount of data by combining adjacent samples. It is useful for zooming in and out of the data without having to repeatedly transfer the entire contents of the scope's buffer to the PC.

- **Memory segmentation.** The scope's internal memory can be divided into segments so that you can capture several waveforms in succession. Configure this using `ps2000aMemorySegments()`.

- **Data retention.** The data is lost when a new run is started in the same segment, the settings are changed, or the scope is powered down.

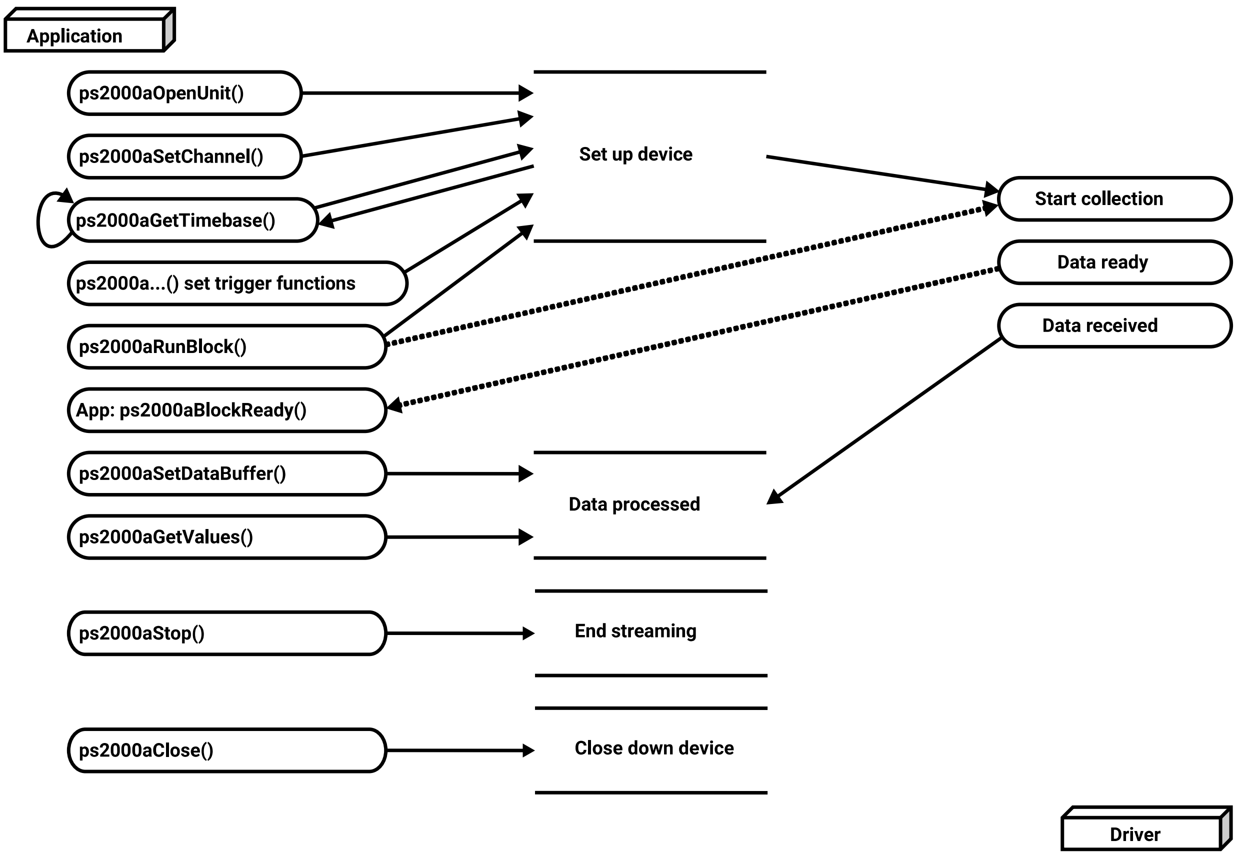
See Using block mode for programming details.

2.6.1.1 Using block mode

This is the general procedure for reading and displaying data in block mode using a single memory segment:

***Note**: Use the * steps when using the digital ports on MSO models.*

1. Open the oscilloscope using `ps2000aOpenUnit()`.
2. Select channel ranges and AC/DC coupling using `ps2000aSetChannel()`.
2*. Set the digital port using `ps2000aSetDigitalPort()`.
3. Using `ps2000aGetTimebase()`, select timebases until the required nanoseconds per sample is located.
4. Use the trigger setup functions `ps2000aSetTriggerChannelConditions()`, `ps2000aSetTriggerChannelDirections()` and `ps2000aSetTriggerChannelProperties()` to set up the trigger if required.
4*. Use the trigger setup functions `ps2000aSetTriggerDigitalPortProperties()` and `ps2000aSetTriggerChannelConditions()` to set up the digital trigger if required.
5. Start the oscilloscope running using `ps2000aRunBlock()`.
6. Wait until the oscilloscope is ready using the `ps2000aBlockReady()` callback (or poll using `ps2000aIsReady()`).
7. Use `ps2000aSetDataBuffer()` to tell the driver where your memory buffer is. (For greater efficiency when doing multiple captures, you can call this function outside the loop, after step 4.)
8. Transfer the block of data from the oscilloscope using `ps2000aGetValues()`.
9. Display the data.
10. Repeat steps 5 to 9.
11. Stop the oscilloscope using `ps2000aStop()`.
12. Request new views of stored data using different downsampling parameters. See Retrieving stored data.
13. Call `ps2000aCloseUnit()`.



2.6.1.2 Asynchronous calls in block mode

To avoid blocking the calling thread when calling `ps2000aGetValues()`, it is possible to call `ps2000aGetValuesAsync()` instead. This immediately returns control to the calling thread, which then has the option of waiting for the data or calling `ps2000aStop()` to abort the operation.

2.6.2 Rapid block mode

In normal block mode, the PicoScope 2000 Series scopes collect one waveform at a time. You start the the device running, wait until all samples are collected by the device, and then download the data to the PC or start another run. There is a time overhead of tens of milliseconds associated with starting a run, causing a gap between waveforms. When you collect data from the device, there is another minimum time overhead which is most noticeable when using a small number of samples.

Rapid block mode allows you to sample several waveforms in succession with minimal time between waveforms. It reduces the gap from milliseconds to less than 2 microseconds (on the fastest timebase). Each waveform is stored in a separate buffer segment.

2.6.2.1 Using rapid block mode

You can use rapid block mode with or without aggregation. With aggregation, you need to set up two buffers per channel to receive the minimum and maximum values.

Note*: Use the * steps when using the digital ports on the mixed-signal (MSO) models.*

Without aggregation

1. Open the oscilloscope using `ps2000aOpenUnit()`.
2. Select channel ranges and AC/DC coupling using `ps2000aSetChannel()`.
3. [MSOs only] Set the digital port using `ps2000aSetDigitalPort()`.
4. Set the number of memory segments equal to or greater than the number of captures required using `ps2000aMemorySegments()`. Use `ps2000aSetNoOfCaptures()` before each run to specify the number of waveforms to capture.
5. Using `ps2000aGetTimebase()`, select timebases from zero upwards until the required number of nanoseconds per sample is located.
6. Use the trigger setup functions `ps2000aSetTriggerChannelConditions()`, `ps2000aSetTriggerChannelDirections()` and `ps2000aSetTriggerChannelProperties()` to set up the trigger if required.
7. [MSOs only] Use the trigger setup functions `ps2000aSetTriggerDigitalPortProperties()` and `ps2000aSetTriggerChannelConditions()` to set up the digital trigger if required.
8. Start the oscilloscope running using `ps2000aRunBlock()`.
9. Wait until the oscilloscope is ready using the `ps2000aIsReady()` or wait on the callback function.
10. Use `ps2000aSetDataBuffer()` to tell the driver where your memory buffers are. Call the function once for each channel/segment combination for which you require data. For greater efficiency, these calls can be made outside the loop, between steps 7 and 8.
11. Transfer the blocks of data from the oscilloscope using `ps2000aGetValuesBulk()`.
12. Retrieve the time offset for each data segment using `ps2000aGetValuesTriggerTimeOffsetBulk64()`.
13. Display the data.
14. Repeat steps 8 to 13 if you wish to capture more data.
15. Stop the oscilloscope using `ps2000aStop()`.
16. Call `ps2000aCloseUnit()`.

With aggregation

To use rapid block mode with aggregation, follow steps 1 to 9 above and then:

10a. Call `ps2000aSetDataBuffer()` or (`ps2000aSetDataBuffers()`) to set up one pair of buffers for every waveform segment required.
11a. Call `ps2000aGetValuesBulk()` for each pair of buffers.
12a. Retrieve the time offset for each data segment using `ps2000aGetValuesTriggerTimeOffsetBulk64()`.

Continue from step 13.

2.6.2.2 Rapid block mode example 1: no aggregation

```
#define MAX_SAMPLES 1000
```

Set up the device up as usual.

- Open the device
- Channels
- Trigger
- Number of memory segments (this should be equal or more than the no of captures required)

```
// Set the number of waveforms to 32
ps2000aSetNoOfCaptures (handle, 32);

pParameter = false;
ps2000aRunBlock
(
  handle,
  0,             // noOfPreTriggerSamples
  MAX_SAMPLES,   // noOfPostTriggerSamples
  1,             // timebase to be used
  1,
  &timeIndisposedMs,
  0,             // segment index
  lpReady,
  &pParameter
);
```

Comment: these variables have been set as an example and can be any valid value. `pParameter` will be set true by your callback function `lpReady`.

```
while (!pParameter) Sleep (0);

for (int i = 0; i < 10; i++)
{
  for (int c = PS2000A_CHANNEL_A; c <= PS2000A_CHANNEL_B; c++)
  {
    ps2000aSetDataBuffer
    (
      handle,
      c,
      &buffer[c][i],
      MAX_SAMPLES,
      i,
      PS2000A_RATIO_MODE_NONE
    );
  }
}
```

Comments: buffer has been created as a two-dimensional array of pointers to int16_t, which will contain 1000 samples as defined by `MAX_SAMPLES`. There are only 10 buffers set, but it is possible to set up to the number of captures you have requested.


```
ps2000aGetValuesBulk
(
  handle,
  &noOfSamples,            // set to MAX_SAMPLES on entering the function
  10,                      // fromSegmentIndex
  19,                      // toSegmentIndex
  1,                       // downsampling ratio
  PS2000A_RATIO_MODE_NONE, // downsampling ratio mode
  overflow                 // an array of size 10 int16_t
)
```

Comments: See the earlier snippets for code to set up the segment buffers.

The number of samples could be up to `noOfPreTriggerSamples + noOfPostTriggerSamples`, the values set in `ps2000aRunBlock`. The samples are always returned from the first sample taken, unlike the `ps2000aGetValues` function which allows the sample index to be set. The above segments start at 10 and finish at 19 inclusive. It is possible for the `fromSegmentIndex` to wrap around to the `toSegmentIndex`, by setting the `fromSegmentIndex` to 28 and the `toSegmentIndex` to 7.

```
ps2000aGetValuesTriggerTimeOffsetBulk64
(
  handle,
  times,
  timeUnits,
  10,
  19
)
```

Comments: the above segments start at 10 and finish at 19 inclusive. It is possible for the `fromSegmentIndex` to wrap around to the `toSegmentIndex`, if the `fromSegmentIndex` is set to 28 and the `toSegmentIndex` to 7.

2.6.2.3 Rapid block mode example 2: using aggregation

```
#define MAX_SAMPLES 1000
```

Set up the device up as usual.

- Open the device
- Channels
- Trigger
- Number of memory segments (this should be equal or more than the number of captures required)

```
// Set the number of waveforms to 32
ps2000aSetNoOfCaptures(handle, 32);

pParameter = false;
ps2000aRunBlock
(
  handle,
  0,                    // noOfPreTriggerSamples,
  MAX_SAMPLES,          // noOfPostTriggerSamples,
  1,                    // timebase to be used,
  1,
  &timeIndisposedMs,
  1,                    // SegmentIndex
  lpReady,
  &pParameter
);
```

Comments: the set-up for running the device is exactly the same whether or not aggregation will be used when you retrieve the samples.

```
for (int segment = 10; segment < 20; segment++)
{
  for (int c = PS2000A_CHANNEL_A; c <= PS2000A_CHANNEL_D; c++)
  {
    ps2000aSetDataBuffers
    (
      handle,
      c,
      &bufferMax[c],
      &bufferMin[c]
      MAX_SAMPLES
      segment,
      PS2000A_RATIO_MODE_AGGREGATE
    );
  }
}
```

Comments: since only one waveform will be retrieved at a time, you only need to set up one pair of buffers; one for the maximum samples and one for the minimum samples. Again, the buffer sizes are 1000 (MAX_SAMPLES) samples.

```
ps2000aGetValues
(
```



```
    handle,
    0,
    &noOfSamples,          // set to MAX_SAMPLES on entering
    10,
    &downSampleRatioMode, //set to RATIO_MODE_AGGREGATE
    index,
    overflow
  );

  ps2000aGetTriggerTimeOffset64
  (
    handle,
    &time,
    &timeUnits,
    index
  );
```

Comments: each waveform is retrieved one at a time from the driver with an aggregation of 10.

2.6.3 ETS (Equivalent Time Sampling)

ETS is a way of increasing the effective sampling rate of the scope when capturing repetitive signals. It is a modified form of block mode, and is controlled by the ps2000a set of trigger functions and the `ps2000aSetEts()` function.

- **Overview.** ETS works by capturing several cycles of a repetitive waveform, then combining them to produce a composite waveform that has a higher effective sampling rate than the individual captures. The scope hardware accurately measures the delay, which is a small fraction of a single sampling interval, between each trigger event and the subsequent sample. The driver then shifts each capture slightly in time and overlays them so that the trigger points are exactly lined up. The result is a larger set of samples spaced by a small fraction of the original sampling interval. The maximum effective sampling rates that can be achieved with this method are listed in the User's Guide for the scope device. Other scopes do not contain special ETS hardware, so the composite waveform is created by software.

- **Trigger stability.** Because of the high sensitivity of ETS mode to small time differences, the trigger must be set up to provide a stable waveform that varies as little as possible from one capture to the next.

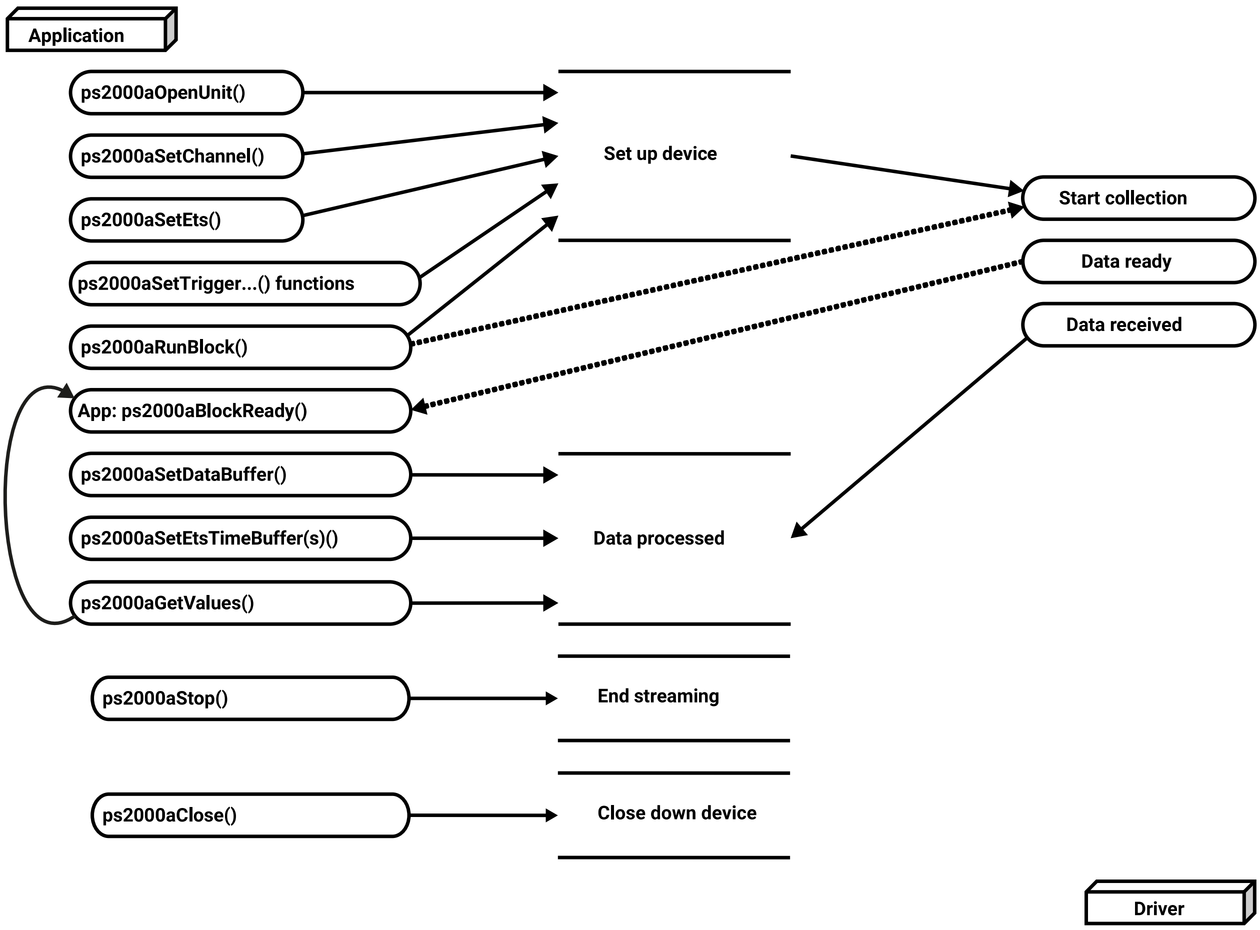
- **Callback.** ETS mode calls the `ps2000aBlockReady()` callback function when a new waveform is ready for collection. The `ps2000aGetValues()` function needs to be called for the waveform to be retrieved.

Applicability	Available in block mode only. Not suitable for one-shot (non-repetitive) signals. Aggregation is not supported. Edge-triggering only. Trigger source may be limited to specific input channels - see device datasheet. Auto trigger delay (`autoTriggerMilliseconds`) is ignored. Cannot be used when MSO digital ports are enabled.

2.6.3.1 Using ETS mode

This is the general procedure for reading and displaying data in ETS mode using a single memory segment:

1. Open the oscilloscope using `ps2000aOpenUnit()`.
2. Select channel ranges and AC/DC coupling using `ps2000aSetChannel()`.
3. Use `ps2000aSetEts()` to enable ETS and set the parameters.
4. Use the trigger setup functions `ps2000aSetTriggerChannelConditions()`, `ps2000aSetTriggerChannelDirections()` and `ps2000aSetTriggerChannelProperties()` to set up the trigger if required.
5. Start the oscilloscope running using `ps2000aRunBlock()`.
6. Wait until the oscilloscope is ready using the `ps2000aBlockReady()` callback (or poll using `ps2000aIsReady()`).
7. Use `ps2000aSetDataBuffer()` to tell the driver where to store sampled data.
8. Use `ps2000aSetEtsTimeBuffer()` or `ps2000aSetEtsTimeBuffers()` to tell the driver where to store sample times.
9. Transfer the block of data from the oscilloscope using `ps2000aGetValues()`.
10. Display the data.
11. While you want to collect updated captures, repeat steps 7 to 10.
12. Repeat steps 5 to 11.
13. Stop the oscilloscope using `ps2000aStop()`.
14. Call `ps2000aCloseUnit()`.



2.6.4 Streaming mode

Streaming mode, unlike block mode, can capture data without gaps between blocks. Streaming mode supports downsampling and triggering, while providing fast streaming. This makes it suitable for **high-speed data acquisition**, allowing you to capture long data sets limited only by the computer's memory.

Aggregation
The driver returns aggregated readings while the device is streaming. If aggregation is set to 1, only one buffer is used per channel. When aggregation is set above 1, two buffers (maximum and minimum) per channel are used.

See Using streaming mode for programming details.

2.6.4.1 Using streaming mode

This is the general procedure for reading and displaying data in streaming mode:

***Note**: Please use the * steps when using the digital ports on the mixed-signal (MSO) models.*

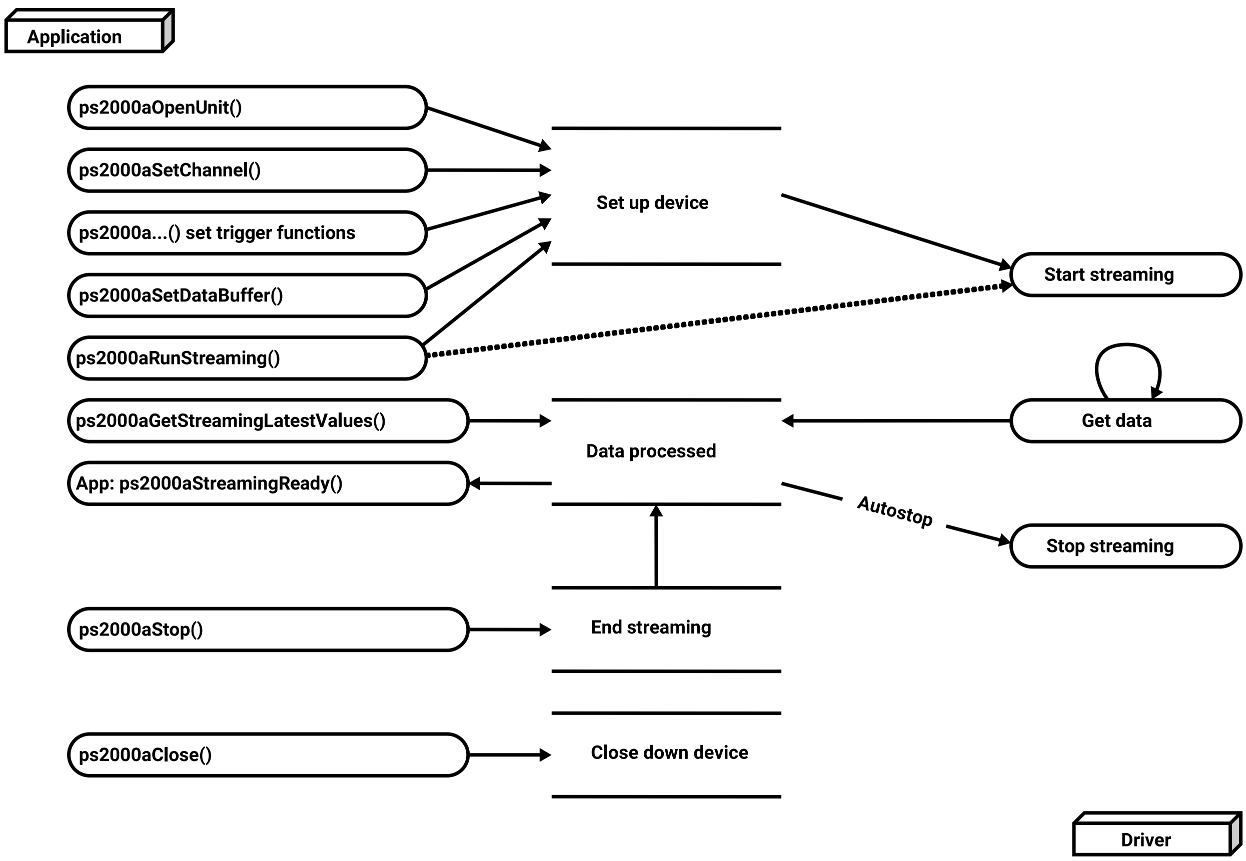
1. Open the oscilloscope using `ps2000aOpenUnit()`.
2. Select channels, ranges and AC/DC coupling using `ps2000aSetChannel()`.

*2. Set the digital port using `ps2000aSetDigitalPort()`.

3. Use the trigger setup functions `ps2000aSetTriggerChannelConditions()`, `ps2000aSetTriggerChannelDirections()` and `ps2000aSetTriggerChannelProperties()` to set up the trigger if required.

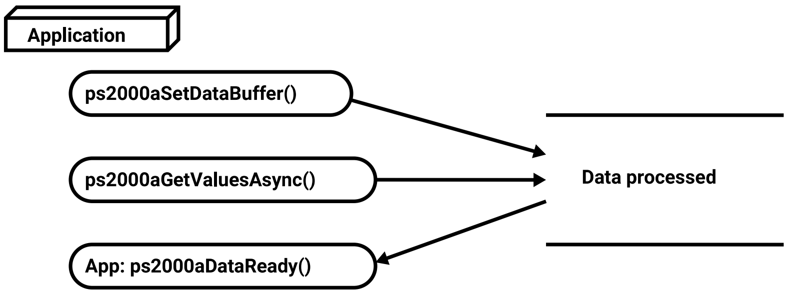
*3. Use the trigger setup functions `ps2000aSetTriggerDigitalPortProperties()` and `ps2000aSetTriggerChannelConditions()` to set up the digital trigger if required.

4. Call `ps2000aSetDataBuffer()` (or `ps2000aSetDataBuffers()` if you will be using aggregation) to tell the driver where your data buffer is.
5. Start the oscilloscope running using `ps2000aRunStreaming()`.
6. Call `ps2000aGetStreamingLatestValues()` to get data.
7. Process data returned to your application's function. This example is using `autoStop`, so after the driver has received all the data points requested by the application, it stops the device streaming.
8. Call `ps2000aStop()`, even if `autoStop` is enabled.
9. Request new views of stored data using different downsampling parameters: see Retrieving stored data.
10. Call `ps2000aCloseUnit()`.



2.6.5 Retrieving stored data

You can collect data from the ps2000a driver with a different downsampling factor when ps2000aRunBlock() or ps2000aRunStreaming() has already been called and has successfully captured all the data. Use ps2000aGetValuesAsync().



2.7 Timebases

The ps2000a API allows you to select any of 2^{32} different timebases based on the maximum sampling rate† of your oscilloscope. The timebases allow slow enough sampling in block mode to overlap the streaming sample intervals, so that you can make a smooth transition between block mode and streaming mode. Calculate the timebase using `ps2000aGetTimebase()`.

500 MS/s maximum sampling rate models:

timebase (n)	sample interval formula	sample interval values
0	2^n / 500,000,000	2 ns*
1		4 ns
2		8 ns
3 to $2^{32}-1$	(n – 2) / 62,500,000	3 => 16 ns ... $2^{32}-1$ => ~ 69 s

1 GS/s maximum sampling rate models:

timebase (n)	sample interval formula	sample interval values
0	2^n / 1,000,000,000	1 ns*
1		2 ns
2		4 ns
3 to $2^{32}-1$	(n – 2) / 125,000,000	3 => 8 ns ... $2^{32}-1$ => ~ 34 s

PicoScope 2205 MSO:

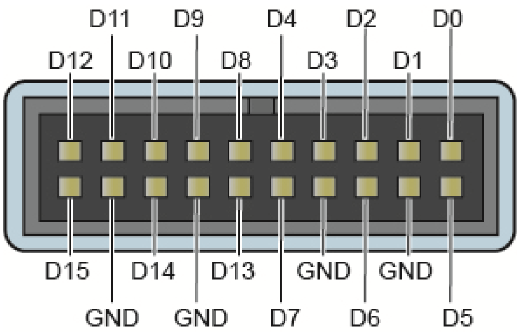
timebase (n)	sample interval formula	sample interval values
0	2^n / 200,000,000	0 => 5 ns**
1	n / 100,000,000	10 ns
2		20 ns
3 to $2^{32}-1$		3 => 30 ns ... $2^{32}-1$ => ~ 43 s

† The fastest available sampling rate may depend on which channels are enabled, and on the sampling mode. Refer to the oscilloscope data sheet for sampling rate specifications. In streaming mode the sampling rate may additionally be limited by the speed of the USB port.
* Available only in single-channel mode.
** Not available when channel B active, nor when channel A and both digital ports active.

ETS mode
In ETS mode the sample time is not set according to the above tables but is instead calculated and returned by `ps2000aSetEts()`.

2.8 MSO digital connector

The MSO models have a digital input connector. The layout of the 20-pin header plug is detailed below. The diagram is drawn as you look at the front panel of the device.



2.9 Combining oscilloscopes

It is possible to collect data using up to 64 PicoScope 2000 Series oscilloscopes at the same time, subject to the capabilities of the PC. Each oscilloscope must be connected to a separate USB port. The `ps2000aOpenUnit()` function returns a handle to an oscilloscope. All the other functions require this handle for oscilloscope identification. For example, to collect data from two oscilloscopes at the same time:

```
CALLBACK ps2000aBlockReady(...)
// define callback function specific to application

handle1 = ps2000aOpenUnit()
handle2 = ps2000aOpenUnit()

ps2000aSetChannel(handle1)

// set up unit 1
ps2000aSetDigitalPort(handle1) // only when using MSO
ps2000aRunBlock(handle1)

ps2000aSetChannel(handle2)

// set up unit 2
ps2000aSetDigitalPort(handle2) // only when using MSO
ps2000aRunBlock(handle2)

// data will be stored in buffers
// and application will be notified using callback

ready = FALSE
while not ready
   ready = handle1_ready
   ready &= handle2_ready
```


3 API functions

The ps2000a API exports a number of functions for you to use in your own applications. All functions are C functions using the standard call naming convention (`__stdcall`). They are all exported with both decorated and undecorated names.

3.1 ps2000aBlockReady() – find out if block-mode data ready

```
typedef void (CALLBACK *ps2000aBlockReady)
(
  int16_t          handle,
  PICO_STATUS      status,
  void           * pParameter
)
```

This callback function is part of your application. You register it with the ps2000a driver using `ps2000aRunBlock()`, and the driver calls it back when block-mode data is ready. The callback function may check that data is available or detect that an error has occurred, but should not attempt to retrieve captured data by calling other ps2000a functions. After the callback function has returned, another part of your application can download the data using `ps2000aGetValues()`.

Applicability	Block mode only

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`status`, indicates whether an error occurred during collection of the data.

`* pParameter`, a void pointer passed from `ps2000aRunBlock()`. Your callback function can write to this location to send any data, such as a status flag, back to your application.

Returns	nothing

3.2 ps2000aCloseUnit() – close a scope device

```
PICO_STATUS ps2000aCloseUnit
(
  int16_t   handle
)
```

This function shuts down an oscilloscope.

Applicability	All modes

Arguments
handle, device identifier returned by ps2000aOpenUnit().

Returns	PICO_OK PICO_HANDLE_INVALID PICO_USER_CALLBACK PICO_DRIVER_FUNCTION

3.3 ps2000aDataReady() – find out if post-collection data ready

```
typedef void (__stdcall *ps2000aDataReady)
(
  int16_t          handle,
  PICO_STATUS      status,
  uint32_t         noOfSamples,
  int16_t          overflow,
  void           * pParameter
)
```

This is a callback function that you write to collect data from the driver. You supply a pointer to the function when you call `ps2000aGetValuesAsync`, and the driver calls your function back when the data is ready.

Applicability	All modes

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`status`, a `PICO_STATUS` code returned by the driver.

`noOfSamples`, the number of samples collected.

`overflow`, a set of flags that indicates whether an overvoltage has occurred and on which channels. It is a bit field with bit 0 representing Channel A.

`* pParameter`, a void pointer passed from `ps2000aGetValuesAsync()`. The callback function can write to this location to send any data, such as a status flag, back to the application. The data type is defined by the application programmer.

Returns	nothing

3.4 ps2000aEnumerateUnits() – find all connected oscilloscopes

```
PICO_STATUS ps2000aEnumerateUnits
(
  int16_t    * count,
  int8_t     * serials,
  int16_t    * serialLth
)
```

This function counts the number of unopened PicoScope 2000 Series (A API) units connected to the computer and returns a list of serial numbers as a string. It does not detect units that already have a handle assigned to them by the driver.

Applicability	All modes

Arguments

`* count,` on exit, the number of ps2000a units found.

`* serials,` on exit, a list of serial numbers separated by commas and terminated by a final null.

Example: `AQ005/139,VDR61/356,ZOR14/107`

Can be `NULL` on entry if serial numbers are not required.

`* serialLth`, on entry, the length of the `char` buffer pointed to by `serials`; on exit, the length of the string written to `serials`

Returns	PICO_OK PICO_BUSY PICO_NULL_PARAMETER PICO_FW_FAIL PICO_CONFIG_FAIL PICO_MEMORY_FAIL PICO_CONFIG_FAIL_AWG PICO_INITIALISE_FPGA

3.5 ps2000aFlashLed() – flash the front-panel LED

```
PICO_STATUS ps2000aFlashLed
(
  int16_t  handle,
  int16_t  start
)
```

This function flashes the LED on the front of the scope without blocking the calling thread. Calls to ps2000aRunStreaming() and ps2000aRunBlock() cancel any flashing started by this function. It is not possible to set the LED to be constantly illuminated, as this state is used to indicate that the scope has not been initialized.

Applicability	All modes

Arguments

handle, device identifier returned by ps2000aOpenUnit().

start, the action required:

< 0 : flash the LED indefinitely
0 : stop the LED flashing
> 0 : flash the LED start times. If the LED is already flashing on entry to this function, the flash count will be reset to start.

Returns	PICO_OK PICO_HANDLE_INVALID PICO_BUSY PICO_DRIVER_FUNCTION PICO_NOT_RESPONDING

3.6 ps2000aGetAnalogueOffset() – get allowable offset range

```
PICO_STATUS ps2000aGetAnalogueOffset
(
  int16_t                handle,
  PS2000A_RANGE          range,
  PS2000A_COUPLING       coupling
  float                * maximumVoltage,
  float                * minimumVoltage
)
```

This function is used to get the maximum and minimum allowable analog offset for a specific voltage range.

Applicability	All ps2000a units except the PicoScope 2205 MSO

Arguments

handle, device identifier returned by ps2000aOpenUnit().

range, the voltage range to be used when gathering the min and max information.

coupling, the type of AC/DC coupling used.

* maximumVoltage, output: maximum voltage allowed for the range. Pointer will be ignored if NULL. If device does not support analog offset, zero will be returned.

* minimumVoltage, output: minimum voltage allowed for the range. Pointer will be ignored if NULL. If device does not support analog offset, zero will be returned.

If both maximumVoltage and minimumVoltage are NULL, the driver will return PICO_NULL_PARAMETER.

Returns	PICO_OK PICO_INVALID_HANDLE PICO_DRIVER_FUNCTION PICO_INVALID_VOLTAGE_RANGE PICO_NULL_PARAMETER

3.7 ps2000aGetChannelInformation() – get list of available ranges

```
PICO_STATUS ps2000aGetChannelInformation
(
  int16_t                    handle,
  PS2000A_CHANNEL_INFO       info,
  int32_t                    probe
  int32_t                  * ranges
  int32_t                  * length
  int32_t                    channels
)
```

This function queries which ranges are available on a scope device.

Applicability	All modes

Arguments

handle, device identifier returned by ps2000aOpenUnit().

info, the type of information required. The following value is currently supported:
PS2000A_CI_RANGES

probe, not used, must be set to 0.

* ranges, an array that will be populated with available PS2000A_RANGE values for the given info. If NULL, length is set to the number of ranges available.

* length, input: length of ranges array; output: number of elements written to ranges array.

channels, the channel for which the information is required.

Returns	PICO_OK PICO_HANDLE_INVALID PICO_BUSY PICO_DRIVER_FUNCTION PICO_NOT_RESPONDING PICO_NULL_PARAMETER PICO_INVALID_CHANNEL PICO_INVALID_INFO

3.8 ps2000aGetMaxDownSampleRatio() – get aggregation ratio for data

```
PICO_STATUS ps2000aGetMaxDownSampleRatio
(
  int16_t                  handle,
  uint32_t                 noOfUnaggregatedSamples,
  uint32_t               * maxDownSampleRatio,
  PS2000A_RATIO_MODE       downSampleRatioMode,
  uint32_t                 segmentIndex
)
```

This function returns the maximum downsampling ratio that can be used for a given number of samples in a given downsampling mode.

Applicability	All modes

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`noOfUnaggregatedSamples`, the number of unprocessed samples to be downsampled.

`* maxDownSampleRatio`, the maximum possible downsampling ratio output.

`downSampleRatioMode`, the downsampling mode. See `ps2000aGetValues()`.

`segmentIndex`, the memory segment where the data is stored.

Returns	`PICO_OK`
	`PICO_INVALID_HANDLE`
	`PICO_NO_SAMPLES_AVAILABLE`
	`PICO_NULL_PARAMETER`
	`PICO_INVALID_PARAMETER`
	`PICO_SEGMENT_OUT_OF_RANGE`
	`PICO_TOO_MANY_SAMPLES`

3.9 ps2000aGetMaxSegments() – find out how many segments allowed

```
PICO_STATUS ps2000aGetMaxSegments
(
  int16_t        handle,
  uint32_t     * maxsegments
)
```

This function returns the maximum number of segments allowed for the opened variant. Refer to `ps2000aMemorySegments()` for specific figures.

Applicability	All modes

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`* maxsegments`, output: maximum number of segments allowed.

Returns	PICO_OK PICO_INVALID_HANDLE PICO_DRIVER_FUNCTION PICO_NULL_PARAMETER

3.10 ps2000aGetNoOfCaptures() – get number of captures available

```
PICO_STATUS ps2000aGetNoOfCaptures
(
  int16_t        handle,
  uint32_t     * nCaptures
)
```

This function finds out how many captures are available in rapid block mode after `ps2000aRunBlock()` has been called. It can be called during data capture, or after the normal end of collection, or after data collection was terminated by `ps2000aStop()`. The returned value (`* nCaptures`) can then be used to iterate through the number of segments using `ps2000aGetValues()`, or in a single call to `ps2000aGetValuesBulk()` where it is used to calculate the `toSegmentIndex` parameter.

Applicability	Rapid block mode

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`* nCaptures`, output: the number of available captures that has been collected from calling `ps2000aRunBlock()`.

Returns	`PICO_OK`
	`PICO_DRIVER_FUNCTION`
	`PICO_INVALID_HANDLE`
	`PICO_NOT_RESPONDING`
	`PICO_NO_SAMPLES_AVAILABLE`
	`PICO_NULL_PARAMETER`
	`PICO_INVALID_PARAMETER`
	`PICO_SEGMENT_OUT_OF_RANGE`
	`PICO_TOO_MANY_SAMPLES`

3.11 ps2000aGetNoOfProcessedCaptures() – get number of captures processed

```
PICO_STATUS ps2000aGetNoOfProcessedCaptures
(
  int16_t         handle,
  uint32_t      * nCaptures
)
```

This function finds out how many captures in rapid block mode have been processed after ps2000aRunBlock() has been called and the collection is either still in progress, completed, or interrupted by a call to ps2000aStop().

It is mainly intended for use while capture is still in progress and you are collecting data using ps2000aGetValuesOverlappedBulk(). The returned value (* nCaptures) indicates how many captures have been completed and therefore how many buffer segments have been filled.

Applicability	Rapid block mode

Arguments

handle, device identifier returned by ps2000aOpenUnit().

* nCaptures, output: the number of available captures resulting from the call to ps2000aRunBlock().

Returns	PICO_OK PICO_DRIVER_FUNCTION PICO_INVALID_HANDLE PICO_NO_SAMPLES_AVAILABLE PICO_NULL_PARAMETER PICO_INVALID_PARAMETER PICO_SEGMENT_OUT_OF_RANGE PICO_TOO_MANY_SAMPLES

3.12 ps2000aGetStreamingLatestValues() – get streaming data while scope is running

```
PICO_STATUS ps2000aGetStreamingLatestValues
(
  int16_t                     handle,
  ps2000aStreamingReady       lpPs2000AReady,
  void                      * pParameter
)
```

This function instructs the driver to return the next block of values to your `ps2000aStreamingReady()` callback function. You must have previously called `ps2000aRunStreaming()` beforehand to set up streaming.

Applicability	Streaming mode only

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`lpPs2000AReady`, a pointer to your `ps2000aStreamingReady()` callback function

`* pParameter`, a void pointer that will be passed to the `ps2000aStreamingReady()` callback function. The callback function may optionally use this pointer to return information to the application.

Returns	`PICO_OK`
	`PICO_INVALID_HANDLE`
	`PICO_NO_SAMPLES_AVAILABLE`
	`PICO_INVALID_CALL`
	`PICO_BUSY`
	`PICO_NOT_RESPONDING`
	`PICO_DRIVER_FUNCTION`

3.13 ps2000aGetTimebase() – find out what timebases are available

```
PICO_STATUS ps2000aGetTimebase
(
  int16_t        handle,
  uint32_t       timebase,
  int32_t        noSamples,
  int32_t      * timeIntervalNanoseconds,
  int16_t        oversample,
  int32_t      * maxSamples
  uint32_t       segmentIndex
)
```

This function calculates the sampling rate and maximum number of samples for a given timebase under the specified conditions. The result depends on the number of channels enabled by the last call to `ps2000aSetChannel()`.

This function is provided for use with programming languages that do not support the `float` data type. The value returned in the `timeIntervalNanoseconds` argument is restricted to integers. If your programming language supports the `float` type, we recommend that you use `ps2000aGetTimebase2()` instead.

To use `ps2000aGetTimebase()` or `ps2000aGetTimebase2()`, first estimate the timebase number that you require using the information in the timebase guide. Next, call one of these functions with the timebase that you have just chosen and verify that the value returned in `timeIntervalNanoseconds` is the one you require. You may need to iterate this process until you obtain the time interval that you need.

Applicability	All modes

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`timebase`, see timebase guide

`noSamples`, the number of samples required

`* timeIntervalNanoseconds`, on exit, the time interval between readings at the selected timebase. Use `NULL` if not required. In ETS mode this argument is not valid; use the sample time returned by `ps2000aSetEts()` instead.

`oversample`, not used

`* maxSamples`, on exit, the maximum number of samples available. The scope allocates a certain amount of memory for internal overheads and this may vary depending on the number of segments, number of channels enabled, and the timebase chosen. Use `NULL` if not required.

`segmentIndex`, the index of the memory segment to use.

Returns	PICO_OK PICO_INVALID_HANDLE PICO_TOO_MANY_SAMPLES PICO_INVALID_CHANNEL PICO_INVALID_TIMEBASE PICO_INVALID_PARAMETER PICO_SEGMENT_OUT_OF_RANGE PICO_DRIVER_FUNCTION

3.14 ps2000aGetTimebase2() – find out what timebases are available

```
PICO_STATUS ps2000aGetTimebase2
(
  int16_t       handle,
  uint32_t      timebase,
  int32_t       noSamples,
  float       * timeIntervalNanoseconds,
  int16_t       oversample,
  int32_t     * maxSamples
  uint32_t      segmentIndex
)
```

This function is an upgraded version of `ps2000aGetTimebase()`, and returns the time interval as a `float` rather than a `long`. This allows it to return sub-nanosecond time intervals. See `ps2000aGetTimebase()` for a full description.

Applicability	All modes

Arguments

`* timeIntervalNanoseconds,` a pointer to the time interval between readings at the selected timebase. If a null pointer is passed, nothing will be written here.

All other arguments: see `ps2000aGetTimebase()`.

Returns	See `ps2000aGetTimebase()`

3.15 ps2000aGetTriggerTimeOffset() – find out when trigger occurred (32-bit)

```
PICO_STATUS ps2000aGetTriggerTimeOffset
(
  int16_t                handle
  uint32_t             * timeUpper
  uint32_t             * timeLower
  PS2000A_TIME_UNITS   * timeUnits
  uint32_t               segmentIndex
)
```

This function retrieves the time offset, as lower and upper 32-bit values, for a waveform obtained in block mode or rapid block mode. The time offset of a waveform is the delay from the trigger sampling instant to the time at which the driver estimates the waveform to have crossed the trigger threshold. You can add this offset to the time of each sample in the waveform to reduce trigger jitter. Without using the time offset, trigger jitter can be up to 1 sample period; adding the time offset reduces jitter to a small fraction of a sample period.

Call it after block-mode data has been captured or when data has been retrieved from a previous block-mode capture. A 64-bit version of this function, `ps2000aGetTriggerTimeOffset64()`, is also available.

Applicability	Block mode, rapid block mode

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`* timeUpper`, on exit, the upper 32 bits of the time at which the trigger point occurred

`* timeLower`, on exit, the lower 32 bits of the time at which the trigger point occurred

`* timeUnits`, returns the time units in which `timeUpper` and `timeLower` are measured. The allowable values are:

`PS2000A_FS`
`PS2000A_PS`
`PS2000A_NS`
`PS2000A_US`
`PS2000A_MS`
`PS2000A_S`

`segmentIndex`, the number of the memory segment for which the information is required.

Returns	`PICO_OK`
	`PICO_INVALID_HANDLE`
	`PICO_DEVICE_SAMPLING`
	`PICO_SEGMENT_OUT_OF_RANGE`
	`PICO_NOT_USED_IN_THIS_CAPTURE_MODE`
	`PICO_NOT_RESPONDING`
	`PICO_NULL_PARAMETER`
	`PICO_NO_SAMPLES_AVAILABLE`
	`PICO_DRIVER_FUNCTION`

3.16 ps2000aGetTriggerTimeOffset64() – find out when trigger occurred (64-bit)

```
PICO_STATUS ps2000aGetTriggerTimeOffset64
(
  int16_t                 handle,
  int64_t               * time,
  PS2000A_TIME_UNITS    * timeUnits,
  uint32_t                segmentIndex
)
```

This function retrieves the time offset for a waveform obtained in block mode or rapid block mode. The time offset of a waveform is the delay from the trigger sampling instant to the time at which the driver estimates the waveform to have crossed the trigger threshold. You can add this offset to the time of each sample in the waveform to reduce trigger jitter. Without using the time offset, trigger jitter can be up to 1 sample period; adding the time offset reduces jitter to a small fraction of a sample period.

Call it after block-mode data has been captured or when data has been retrieved from a previous block-mode capture. A 32-bit version of this function, `ps2000aGetTriggerTimeOffset()`, is also available.

Applicability	Block mode, rapid block mode

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`* time`, on exit, the time at which the trigger point occurred.

`* timeUnits`, on exit, the time units in which time is measured. The possible values are:
`PS2000A_FS`
`PS2000A_PS`
`PS2000A_NS`
`PS2000A_US`
`PS2000A_MS`
`PS2000A_S`

`segmentIndex`, the number of the memory segment for which the information is required.

Returns	`PICO_OK`
	`PICO_INVALID_HANDLE`
	`PICO_DEVICE_SAMPLING`
	`PICO_SEGMENT_OUT_OF_RANGE`
	`PICO_NOT_USED_IN_THIS_CAPTURE_MODE`
	`PICO_NOT_RESPONDING`
	`PICO_NULL_PARAMETER`
	`PICO_NO_SAMPLES_AVAILABLE`
	`PICO_DRIVER_FUNCTION`

3.17 ps2000aGetUnitInfo() – get information about scope device

```
PICO_STATUS ps2000aGetUnitInfo
(
  int16_t       handle,
  int8_t      * string,
  int16_t       stringLength,
  int16_t     * requiredSize
  PICO_INFO     info
)
```

This function retrieves information about the specified oscilloscope. If the device fails to open, or no device is opened only the driver version is available.

Applicability	All modes

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`. If an invalid handle is passed, only the driver versions can be read.

`* string`, on exit, the unit information string selected specified by the `info` argument. If `string` is NULL, only `requiredSize` is returned.

`stringLength`, the maximum number of chars that may be written to `string`.

`* requiredSize`, on exit, the required length of the `string` array.

`info`, a number specifying what information is required. The possible values are listed in the table below.

`info`		**Example**
0	`PICO_DRIVER_VERSION` Version number of PicoScope 2000A DLL	`1.0.0.1`
1	`PICO_USB_VERSION` Type of USB connection to device: 1.1 or 2.0	`2.0`
2	`PICO_HARDWARE_VERSION` Hardware version of device	`1`
3	`PICO_VARIANT_INFO` Variant number of device	`2206`
4	`PICO_BATCH_AND_SERIAL` Batch and serial number of device	`KJL87/006`
5	`PICO_CAL_DATE` Calibration date of device	`30Sep09`
6	`PICO_KERNEL_VERSION` Version of kernel driver	`1.0`
7	`PICO_DIGITAL_HARDWARE_VERSION` Hardware version of the digital section	`1`
8	`PICO_ANALOGUE_HARDWARE_VERSION` Hardware version of the analog section	`1`
9	`PICO_FIRMWARE_VERSION_1`	`1.0.0.0`
10	`PICO_FIRMWARE_VERSION_2`	`1.0.0.0`

Returns	PICO_OK PICO_INVALID_HANDLE PICO_NULL_PARAMETER PICO_INVALID_INFO PICO_INFO_UNAVAILABLE PICO_DRIVER_FUNCTION

3.18 ps2000aGetValues() – get block-mode data with callback

```
PICO_STATUS ps2000aGetValues
(
  int16_t                 handle,
  uint32_t                startIndex,
  uint32_t              * noOfSamples,
  uint32_t                downSampleRatio,
  PS2000A_RATIO_MODE      downSampleRatioMode,
  uint32_t                segmentIndex,
  int16_t               * overflow
)
```

This function retrieves block-mode data, either with or without downsampling, starting at the specified sample number. It is used to get the stored data from the scope after data collection has stopped, and store it in a user buffer previously passed to ps2000aSetDataBuffer() or ps2000aSetDataBuffers(). It blocks the calling function while retrieving data.

If multiple channels are enabled, a single call to this function is sufficient to retrieve data for all channels.

Note that if you are using block mode and call this function before the oscilloscope is ready, no capture will be available and the driver will return PICO_NO_SAMPLES_AVAILABLE.

Applicability	Block mode, rapid block mode

Arguments

handle, device identifier returned by ps2000aOpenUnit().

startIndex, a zero-based index that indicates the start point for data collection. It is measured in sample intervals from the start of the buffer.

* noOfSamples, on entry, the number of samples required. On exit, the actual number retrieved. The number of samples retrieved will not be more than the number requested, and the data retrieved starts at startIndex.

downSampleRatio, the downsampling factor that will be applied to the raw data.

downSampleRatioMode, which downsampling mode to use. The available values are:
  PS2000A_RATIO_MODE_NONE (downSampleRatio is ignored)
  PS2000A_RATIO_MODE_AGGREGATE
  PS2000A_RATIO_MODE_AVERAGE
  PS2000A_RATIO_MODE_DECIMATE

AGGREGATE, AVERAGE, DECIMATE are single-bit constants that can be ORed to apply multiple downsampling modes to the same data.

segmentIndex, the zero-based number of the memory segment where the data is stored.

* overflow, on exit, a set of flags that indicate whether an overvoltage has occurred on any of the channels. It is a bit field with bit 0 denoting Channel A.

Returns	PICO_OK PICO_INVALID_HANDLE PICO_NO_SAMPLES_AVAILABLE PICO_DEVICE_SAMPLING PICO_NULL_PARAMETER PICO_SEGMENT_OUT_OF_RANGE PICO_STARTINDEX_INVALID PICO_ETS_NOT_RUNNING PICO_BUFFERS_NOT_SET PICO_INVALID_PARAMETER PICO_TOO_MANY_SAMPLES PICO_DATA_NOT_AVAILABLE PICO_STARTINDEX_INVALID PICO_INVALID_SAMPLERATIO PICO_INVALID_CALL PICO_NOT_RESPONDING PICO_MEMORY PICO_RATIO_MODE_NOT_SUPPORTED PICO_DRIVER_FUNCTION

3.18.1 Downsampling modes

Various methods of data reduction, or **downsampling**, are possible with the PicoScope 2000 Series oscilloscopes. The downsampling is done at high speed, making your application faster and more responsive than if you had to do all your own data processing.

You specify the downsampling mode when you call one of the data collection functions such as `ps2000aGetValues()`. The following modes are available:

PS2000A_RATIO_MODE_NONE	No downsampling. Returns the raw data values.
PS2000A_RATIO_MODE_AGGREGATE	Reduces every block of *n* values to just two values: a minimum and a maximum. The minimum and maximum values are returned in two separate buffers.
PS2000A_RATIO_MODE_AVERAGE	Reduces every block of *n* values to a single value representing the average (arithmetic mean) of all the values. Equivalent to the 'oversampling' function on older scopes.
PS2000A_RATIO_MODE_DECIMATE	Reduces every block of *n* values to just the first value in the block, discarding all the other values.

Retrieving multiple types of downsampled data
You can optionally retrieve data using more than one downsampling mode with a single call to `ps2000aGetValues()`. Set up a buffer for each downsampling mode by calling `ps2000aSetDataBuffer()`. Then, when calling `ps2000aGetValues()`, set `downSampleRatioMode` to the bitwise OR of the required downsampling modes.

Retrieving both raw and downsampled data
You cannot retrieve raw data and downsampled data in a single operation. If you require both raw and downsampled data, first retrieve the downsampled data as described above and then continue as follows:

1. Call `ps2000aStop()`.

2. Set up a data buffer for each channel using `ps2000aSetDataBuffer()` with the ratio mode set to `PS2000A_RATIO_MODE_NONE`.

3. Call `ps2000aGetValues()` to retrieve the data.

3.19 ps2000aGetValuesAsync() – get streaming data with callback

```
PICO_STATUS ps2000aGetValuesAsync
(
  int16_t                  handle,
  uint32_t                 startIndex,
  uint32_t                 noOfSamples,
  uint32_t                 downSampleRatio,
  PS2000A_RATIO_MODE       downSampleRatioMode,
  uint32_t                 segmentIndex,
  void                   * lpDataReady,
  void                   * pParameter
)
```

This function returns data either with or without downsampling, starting at the specified sample number. It is used to get the stored data from the scope after data collection has stopped. It returns the data using a callback so as not to block the calling function. It can also be used in streaming mode to retrieve data from the driver, but in this case it blocks the calling function.

If multiple channels are enabled, a single call to this function is sufficient to retrieve data for all channels.

Applicability	Streaming mode and block mode

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`startIndex`, see `ps2000aGetValues()`
`noOfSamples`, see `ps2000aGetValues()`
`downSampleRatio`, see `ps2000aGetValues()`
`downSampleRatioMode`, see `ps2000aGetValues()`
`segmentIndex`, see `ps2000aGetValues()`

`* lpDataReady`, a pointer to the user-supplied function that will be called when the data is ready. This will be a `ps2000aDataReady()` function for block-mode data or a `ps2000aStreamingReady()` function for streaming-mode data.

`* pParameter`, a void pointer that will be passed to the callback function. The data type is determined by the application.

Returns	PICO_OK PICO_INVALID_HANDLE PICO_NO_SAMPLES_AVAILABLE PICO_DEVICE_SAMPLING PICO_NULL_PARAMETER PICO_STARTINDEX_INVALID PICO_SEGMENT_OUT_OF_RANGE PICO_INVALID_PARAMETER PICO_DATA_NOT_AVAILABLE PICO_INVALID_SAMPLERATIO PICO_INVALID_CALL PICO_DRIVER_FUNCTION

3.20 ps2000aGetValuesBulk() – get data in rapid block mode

```
PICO_STATUS ps2000aGetValuesBulk
(
  int16_t                 handle,
  uint32_t              * noOfSamples,
  uint32_t                fromSegmentIndex,
  uint32_t                toSegmentIndex,
  uint32_t                downSampleRatio,
  PS2000A_RATIO_MODE      downSampleRatioMode,
  int16_t               * overflow
)
```

This function retrieves waveforms captured using rapid block mode. The waveforms must have been collected sequentially and in the same run.

If multiple channels are enabled, a single call to this function is sufficient to retrieve data for all channels.

Applicability	Rapid block mode

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`* noOfSamples`, on entry, the number of samples required; on exit, the actual number retrieved. The number of samples retrieved will not be more than the number requested. The data retrieved always starts with the first sample captured.

`fromSegmentIndex`, the first segment from which the waveform should be retrieved.

`toSegmentIndex`, the last segment from which the waveform should be retrieved.

`downSampleRatio`, see `ps2000aGetValues()`.
`downSampleRatioMode`, see `ps2000aGetValues()`.

`* overflow`, an array of integers equal to or larger than the number of waveforms to be retrieved. Each segment index has a corresponding entry in the `overflow` array, with `overflow[0]` containing the flags for the segment numbered `fromSegmentIndex` and the last element in the array containing the flags for the segment numbered `toSegmentIndex`. Each element in the array is a bit field as described under `ps2000aGetValues()`.

Returns	`PICO_OK`
	`PICO_INVALID_HANDLE`
	`PICO_INVALID_PARAMETER`
	`PICO_INVALID_SAMPLERATIO`
	`PICO_ETS_NOT_RUNNING`
	`PICO_BUFFERS_NOT_SET`
	`PICO_TOO_MANY_SAMPLES`
	`PICO_SEGMENT_OUT_OF_RANGE`
	`PICO_NO_SAMPLES_AVAILABLE`
	`PICO_NOT_RESPONDING`
	`PICO_DRIVER_FUNCTION`

3.21 ps2000aGetValuesOverlapped() – set up data collection ahead of capture

```
PICO_STATUS ps2000aGetValuesOverlapped
(
  int16_t               handle,
  uint32_t              startIndex,
  uint32_t            * noOfSamples,
  uint32_t              downSampleRatio,
  PS2000A_RATIO_MODE    downSampleRatioMode,
  uint32_t              segmentIndex,
  int16_t             * overflow
)
```

This function allows you to make a deferred data-collection request in block mode. The request will be executed, and the arguments validated, when you call `ps2000aRunBlock()`. The advantage of this function is that the driver makes contact with the scope only once, when you call `ps2000aRunBlock()`, compared with the two contacts that occur when you use the conventional `ps2000aRunBlock()`, `ps2000aGetValues()` calling sequence. This slightly reduces the dead time between successive captures in block mode.

After calling `ps2000aRunBlock()`, you can optionally use `ps2000aGetValues()` to request further copies of the data. This might be required if you wish to display the data with different data reduction settings.

If multiple channels are enabled, a single call to this function is sufficient to retrieve data for all channels.

Applicability	Block mode

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`startIndex`, see `ps2000aGetValues()`.

`* noOfSamples`, on entry, the number of raw samples to be collected before any downsampling is applied. On exit, the actual number stored in the buffer. The number of samples retrieved will not be more than the number requested, and the data retrieved starts at `startIndex`.

`downSampleRatio`, see `ps2000aGetValues()`
`downSampleRatioMode`, see `ps2000aGetValues()`
`segmentIndex`, see `ps2000aGetValues()`
`* overflow`, see `ps2000aGetValuesBulk()`

Returns	`PICO_OK`
	`PICO_INVALID_HANDLE`
	`PICO_INVALID_PARAMETER`
	`PICO_DRIVER_FUNCTION`

3.21.1 Using the GetValuesOverlapped functions

1. Open the oscilloscope using `ps2000aOpenUnit()`.
2. Select channel ranges and AC/DC coupling using `ps2000aSetChannel()`.
3. Using `ps2000aGetTimebase()`, select timebases until the required nanoseconds per sample is located.
4. Use the trigger setup functions `ps2000aSetTriggerChannelDirections()` and `ps2000aSetTriggerChannelProperties()` to set up the trigger if required.
5. Wait until the oscilloscope is ready using the `ps2000aBlockReady()` callback (or poll using `ps2000aIsReady()`).
6. Use `ps2000aSetDataBuffer()` to tell the driver where your memory buffer is.
7. Set up the transfer of the block of data from the oscilloscope using `ps2000aGetValuesOverlapped()`.
8. Start the oscilloscope running using `ps2000aRunBlock()`.
9. Display the data.
10. Stop the oscilloscope.
11. Repeat steps 8 and 9 if needed.

A similar procedure can be used with rapid block mode and `ps2000aGetValuesOverlappedBulk()`.

3.22 ps2000aGetValuesOverlappedBulk() – set up data collection in rapid block mode

```
PICO_STATUS ps2000aGetValuesOverlappedBulk
(
  int16_t                    handle,
  uint32_t                   startIndex,
  uint32_t                 * noOfSamples,
  uint32_t                   downSampleRatio,
  PS2000A_RATIO_MODE         downSampleRatioMode,
  uint32_t                   fromSegmentIndex,
  uint32_t                   toSegmentIndex,
  int16_t                  * overflow
)
```

This function allows you to make a deferred data-collection request, which will later be executed, and the arguments validated, when you call `ps2000aRunBlock()` in rapid block mode. The advantage of this method is that the driver makes contact with the scope only once, when you call `ps2000aRunBlock()`, compared with the two contacts that occur when you use the conventional `ps2000aRunBlock()`, `ps2000aGetValuesBulk()` calling sequence. This slightly reduces the dead time between successive captures in rapid block mode.

After calling `ps2000aRunBlock()`, you can optionally use `ps2000aGetValues()` to request further copies of the data. This might be required if you wish to display the data with different data reduction settings.

If multiple channels are enabled, a single call to this function is sufficient to retrieve data for all channels.

Applicability	Rapid block mode

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`startIndex`, see `ps2000aGetValues()`
`* noOfSamples`, see `ps2000aGetValuesOverlapped()`
`downSampleRatio`, see `ps2000aGetValues()`
`downSampleRatioMode`, see `ps2000aGetValues()`
`fromSegmentIndex`, see `ps2000aGetValuesBulk()`
`toSegmentIndex`, see `ps2000aGetValuesBulk()`
`* overflow`, see `ps2000aGetValuesBulk()`

Returns	`PICO_OK`
	`PICO_INVALID_HANDLE`
	`PICO_INVALID_PARAMETER`
	`PICO_DRIVER_FUNCTION`

3.23 ps2000aGetValuesTriggerTimeOffsetBulk() – get rapid-block waveform times (32-bit)

```
PICO_STATUS ps2000aGetValuesTriggerTimeOffsetBulk
(
  int16_t                 handle,
  uint32_t              * timesUpper,
  uint32_t              * timesLower,
  PS2000A_TIME_UNITS    * timeUnits,
  uint32_t                fromSegmentIndex,
  uint32_t                toSegmentIndex
)
```

This function retrieves the time offsets, as lower and upper 32-bit values, for waveforms obtained in rapid block mode. The time offset of a waveform is the delay from the trigger sampling instant to the time at which the driver estimates the waveform to have crossed the trigger threshold. You can add this offset to the time of each sample in the waveform to reduce trigger jitter. Without using the time offset, trigger jitter can be up to 1 sample period; adding the time offset reduces jitter to a small fraction of a sample period.

This function is provided for use in programming environments that do not support 64-bit integers. If your programming environment supports this data type, it is easier to use `ps2000aGetValuesTriggerTimeOffsetBulk64()`.

Applicability	Rapid block mode

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`* timesUpper`, an array of integers. On exit, the most significant 32 bits of the time offset for each requested segment index. `times[0]` will hold the `fromSegmentIndex` time offset and the last `times` index will hold the `toSegmentIndex` time offset. The array must be long enough to hold the number of requested times.

`* timesLower`, an array of integers. On exit, the least significant 32 bits of the time offset for each requested segment index. `times[0]` will hold the `fromSegmentIndex` time offset and the last `times` index will hold the `toSegmentIndex` time offset. The array size must be long enough to hold the number of requested times.

`* timeUnits`, an array of integers. The array must be long enough to hold the number of requested times. On exit, `timeUnits[0]` will contain the time unit for `fromSegmentIndex` and the last element will contain the time unit for `toSegmentIndex`. Refer to `ps2000aGetTriggerTimeOffset()` for allowable values.

`fromSegmentIndex`, the first segment for which the time offset is required.

`toSegmentIndex`, the last segment for which the time offset is required. If `toSegmentIndex` is less than `fromSegmentIndex` then the driver will wrap around from the last segment to the first.

Returns	PICO_OK PICO_INVALID_HANDLE PICO_NOT_USED_IN_THIS_CAPTURE_MODE PICO_NOT_RESPONDING PICO_NULL_PARAMETER PICO_DEVICE_SAMPLING PICO_SEGMENT_OUT_OF_RANGE PICO_NO_SAMPLES_AVAILABLE PICO_DRIVER_FUNCTION

3.24 ps2000aGetValuesTriggerTimeOffsetBulk64() – get rapid-block waveform times (64-bit)

```
PICO_STATUS ps2000aGetValuesTriggerTimeOffsetBulk64
(
  int16_t                handle,
  int64_t              * times,
  PS2000A_TIME_UNITS   * timeUnits,
  uint32_t               fromSegmentIndex,
  uint32_t               toSegmentIndex
)
```

This function retrieves the 64-bit time offsets for waveforms captured in rapid block mode.

A 32-bit version of this function, `ps2000aGetValuesTriggerTimeOffsetBulk()`, is available for use with programming languages that do not support 64-bit integers.

Applicability	Rapid block mode

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`* times`, an array of integers. On exit, this will hold the time offset for each requested segment index. `times[0]` will hold the time offset for `fromSegmentIndex`, and the last `times` index will hold the time offset for `toSegmentIndex`. The array must be long enough to hold the number of times requested.

`* timeUnits`, an array of integers long enough to hold the number of requested times. `timeUnits[0]` will contain the time unit for `fromSegmentIndex`, and the last element will contain the `toSegmentIndex`. Refer to `ps2000aGetTriggerTimeOffset64()` for specific figures.

`fromSegmentIndex`, the first segment for which the time offset is required. The results for this segment will be placed in `times[0]` and `timeUnits[0]`.

`toSegmentIndex`, the last segment for which the time offset is required. The results for this segment will be placed in the last elements of the `times` and `timeUnits` arrays. If `toSegmentIndex` is less than `fromSegmentIndex` then the driver will wrap around from the last segment to the first.

Returns	PICO_OK PICO_INVALID_HANDLE PICO_NOT_USED_IN_THIS_CAPTURE_MODE PICO_NOT_RESPONDING PICO_NULL_PARAMETER PICO_DEVICE_SAMPLING PICO_SEGMENT_OUT_OF_RANGE PICO_NO_SAMPLES_AVAILABLE PICO_DRIVER_FUNCTION

3.25 ps2000aHoldOff() – not supported

```
PICO_STATUS ps2000aHoldOff
(
  int16_t              handle,
  uint64_t             holdoff,
  PS2000A_HOLDOFF_TYPE type
)
```

This function has no effect and is reserved for future use.

Applicability	Not supported. Reserved for future use.

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`holdoff`, reserved for future use.

`type`, reserved for future use.

Returns	`PICO_OK` `PICO_INVALID_HANDLE`

3.26 ps2000aIsReady() – poll driver in block mode

```
PICO_STATUS ps2000aIsReady
(
  int16_t      handle,
  int16_t    * ready
)
```

This function may be used instead of a callback function to receive data from `ps2000aRunBlock()`. To use this method, pass a NULL pointer as the `lpReady` argument to `ps2000aRunBlock()`. You must then poll the driver to see if it has finished collecting the requested samples.

Applicability	Block mode

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`* ready`, output: indicates the state of the collection. If zero, the device is still collecting. If non-zero, the device has finished collecting and `ps2000aGetValues()` can be used to retrieve the data.

Returns	`PICO_OK`
	`PICO_INVALID_HANDLE`
	`PICO_DRIVER_FUNCTION`
	`PICO_NULL_PARAMETER`
	`PICO_NO_SAMPLES_AVAILABLE`
	`PICO_CANCELLED`
	`PICO_NOT_RESPONDING`

3.27 ps2000aIsTriggerOrPulseWidthQualifierEnabled() – get trigger status

```
PICO_STATUS ps2000aIsTriggerOrPulseWidthQualifierEnabled
(
  int16_t     handle,
  int16_t   * triggerEnabled,
  int16_t   * pulseWidthQualifierEnabled
)
```

This function discovers whether a trigger, or pulse width triggering, is enabled.

Applicability	Call after setting up the trigger, and just before calling either `ps2000aRunBlock()` or `ps2000aRunStreaming()`.

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`* triggerEnabled`, on exit, indicates whether the trigger will successfully be set when `ps2000aRunBlock()` or `ps2000aRunStreaming()` is called. A non-zero value indicates that the trigger is set, zero that the trigger is not set.

`* pulseWidthQualifierEnabled`, on exit, indicates whether the pulse width qualifier will successfully be set when `ps2000aRunBlock()` or `ps2000aRunStreaming()` is called. A non-zero value indicates that the pulse width qualifier is set, zero that the pulse width qualifier is not set.

Returns	`PICO_OK`
	`PICO_INVALID_HANDLE`
	`PICO_NULL_PARAMETER`
	`PICO_DRIVER_FUNCTION`

3.28 ps2000aMaximumValue() – get maximum ADC count in GetValues calls

```
PICO_STATUS ps2000aMaximumValue
(
  int16_t       handle
  int16_t     * value
)
```

This function returns the maximum ADC count returned by calls to the `GetValues` functions.

Applicability	All modes

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`* value`, output: the maximum ADC value.

Returns	PICO_OK PICO_USER_CALLBACK PICO_INVALID_HANDLE PICO_TOO_MANY_SEGMENTS PICO_MEMORY PICO_DRIVER_FUNCTION

3.29 ps2000aMemorySegments() – divide scope memory into segments

```
PICO_STATUS ps2000aMemorySegments
(
  int16_t       handle
  uint32_t      nSegments,
  int32_t     * nMaxSamples
)
```

This function sets the number of memory segments that the scope will use.

When the scope is opened, the number of segments defaults to 1, meaning that each capture fills the scope's available memory. This function allows you to divide the memory into a number of segments so that the scope can store several waveforms sequentially.

Applicability	Block mode, rapid block mode

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`nSegments`, the number of segments required. Minimum: 1. Maximum: varies according to oscilloscope model – refer to datasheet.

`* nMaxSamples`, on exit, the number of samples available in each segment. This is the total number over all channels, so if two channels or 8-bit MSO ports are in use, the number of samples available to each channel is `nMaxSamples` divided by 2; and for 3 or 4 channels or MSO ports divide by 4.

Returns	PICO_OK PICO_USER_CALLBACK PICO_INVALID_HANDLE PICO_TOO_MANY_SEGMENTS PICO_MEMORY PICO_DRIVER_FUNCTION

3.30 ps2000aMinimumValue() – get minimum ADC count in GetValues calls

```
PICO_STATUS ps2000aMinimumValue
(
  int16_t      handle
  int16_t    * value
)
```

This function returns the minimum ADC count returned by calls to the `GetValues` functions.

Applicability	All modes

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`* value`, output: the minimum ADC value.

Returns	`PICO_OK`
	`PICO_USER_CALLBACK`
	`PICO_INVALID_HANDLE`
	`PICO_TOO_MANY_SEGMENTS`
	`PICO_MEMORY`
	`PICO_DRIVER_FUNCTION`

3.31 ps2000aNoOfStreamingValues() – get number of samples in streaming mode

```
PICO_STATUS ps2000aNoOfStreamingValues
(
  int16_t      handle,
  uint32_t   * noOfValues
)
```

This function returns the number of raw samples stored in the driver after data collection in streaming mode. Call it after calling `ps2000aStop()`.

Applicability	Streaming mode

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`* noOfValues`, on exit, the number of samples.

Returns	`PICO_OK`
	`PICO_INVALID_HANDLE`
	`PICO_NULL_PARAMETER`
	`PICO_NO_SAMPLES_AVAILABLE`
	`PICO_NOT_USED`
	`PICO_BUSY`
	`PICO_DRIVER_FUNCTION`

3.32 ps2000aOpenUnit() – open a scope device

```
PICO_STATUS ps2000aOpenUnit
(
  int16_t     * handle,
  int8_t      * serial
)
```

This function opens a PicoScope 2000 Series (A API) scope attached to the computer. The maximum number of units that can be opened depends on the operating system, the kernel driver and the computer.

Applicability	All modes

Arguments

`* handle`, on exit, the result of the attempt to open a scope:

`-1` : if the scope fails to open
`0` : if no scope is found
`> 0` : a number that uniquely identifies the scope

If a valid handle is returned, it must be used in all subsequent calls to API functions to identify this scope.

`* serial`, on entry, a null-terminated string containing the serial number of the scope to be opened. If `serial` is `NULL` then the function opens the first scope found; otherwise, it tries to open the scope that matches the string.

Returns	`PICO_OK`
	`PICO_OS_NOT_SUPPORTED`
	`PICO_OPEN_OPERATION_IN_PROGRESS`
	`PICO_EEPROM_CORRUPT`
	`PICO_KERNEL_DRIVER_TOO_OLD`
	`PICO_FPGA_FAIL`
	`PICO_MEMORY_CLOCK_FREQUENCY`
	`PICO_FW_FAIL`
	`PICO_MAX_UNITS_OPENED`
	`PICO_NOT_FOUND` (if the specified unit was not found)
	`PICO_NOT_RESPONDING`
	`PICO_MEMORY_FAIL`
	`PICO_ANALOG_BOARD`
	`PICO_CONFIG_FAIL_AWG`
	`PICO_INITIALISE_FPGA`

3.33 ps2000aOpenUnitAsync() – open a scope device without blocking

```
PICO_STATUS ps2000aOpenUnitAsync
(
  int16_t     * status
  int8_t      * serial
)
```

This function opens a scope without blocking the calling thread. You can find out when it has finished by periodically calling `ps2000aOpenUnitProgress()` until that function returns a non-zero value.

Applicability	All modes

Arguments

`* status`, a status code:
 0 if the open operation was disallowed because another open operation is in progress
 1 if the open operation was successfully started

`* serial`, see `ps2000aOpenUnit()`.

Returns	`PICO_OK`
	`PICO_OPEN_OPERATION_IN_PROGRESS`
	`PICO_OPERATION_FAILED`

3.34 ps2000aOpenUnitProgress() – check progress of OpenUnit call

```
PICO_STATUS ps2000aOpenUnitProgress
(
  int16_t    * handle,
  int16_t    * progressPercent,
  int16_t    * complete
)
```

This function checks on the progress of a request made to ps2000aOpenUnitAsync() to open a scope.

Applicability	Use after ps2000aOpenUnitAsync()

Arguments

`* handle`, see ps2000aOpenUnit(). This handle is valid only if the function returns `PICO_OK`.

`* progressPercent`, on exit, the percentage progress towards opening the scope. 100% implies that the open operation is complete.

`* complete`, set to 1 when the open operation has finished.

Returns	`PICO_OK`
	`PICO_NULL_PARAMETER`
	`PICO_OPERATION_FAILED`

3.35 ps2000aPingUnit() – check communication with opened device

```
PICO_STATUS ps2000aPingUnit
(
  int16_t    handle
)
```

This function can be used to check that the already opened device is still connected to the USB port and communication is successful.

Applicability	All modes

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

Returns	PICO_OK PICO_INVALID_HANDLE PICO_DRIVER_FUNCTION PICO_BUSY PICO_NOT_RESPONDING

3.36 ps2000aQueryOutputEdgeDetect() – find out if state trigger edge-detection is enabled

```
PICO_STATUS ps2000aQueryOutputEdgeDetect
(
  int16_t       handle,
  int16_t     * state
)
```

This function obtains the state of the edge-detect flag, which is described in `ps2000aSetOutputEdgeDetect()`.

Applicability	Level and window trigger types

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`state`, on exit, the value of the edge-detect flag:
- `0` : do not wait for a signal transition
- `<> 0` : wait for a signal transition (default)

Returns	`PICO_OK`

3.37 ps2000aRunBlock() – capture in block mode

```
PICO_STATUS ps2000aRunBlock
(
  int16_t                 handle,
  int32_t                 noOfPreTriggerSamples,
  int32_t                 noOfPostTriggerSamples,
  uint32_t                timebase,
  int16_t                 oversample,
  int32_t               * timeIndisposedMs,
  uint32_t                segmentIndex,
  ps2000aBlockReady       lpReady,
  void                  * pParameter
)
```

This function starts collecting data in block mode. For a step-by-step guide to this process, see Using block mode.

The number of samples is determined by `noOfPreTriggerSamples` and `noOfPostTriggerSamples` (see below for details). The total number of samples must not be more than the size of the segment referred to by `segmentIndex`.

Applicability	Block mode, rapid block mode

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`noOfPreTriggerSamples`, the number of samples to store before the trigger event

`noOfPostTriggerSamples`, the number of samples to store after the trigger event

Note: the maximum number of samples returned is always `noOfPreTriggerSamples + noOfPostTriggerSamples`. This is true even if no trigger event has been set.

`timebase`, a number in the range 0 to $2^{32}-1$. See the guide to calculating timebase values. This argument is ignored in ETS mode, when `ps2000aSetEts()` selects the timebase instead.

`oversample`, not used

`* timeIndisposedMs`, on exit, the time, in milliseconds, that the scope will spend collecting samples. This does not include any auto trigger timeout. It is not valid in ETS capture mode. The pointer can be set to null if a value is not required.

`segmentIndex`, zero-based, which memory segment to use

`lpReady`, a pointer to the `ps2000aBlockReady()` callback function that the driver will call when the data has been collected. To use the `ps2000aIsReady()` polling method instead of a callback function, set this pointer to NULL.

`* pParameter`, a void pointer that is passed to the `ps2000aBlockReady()` callback function. The callback can use this pointer to return arbitrary data to the application.

Returns	`PICO_OK` `PICO_BUFFERS_NOT_SET` (in Overlapped mode)

	PICO_INVALID_HANDLE PICO_USER_CALLBACK PICO_SEGMENT_OUT_OF_RANGE PICO_INVALID_CHANNEL PICO_INVALID_TRIGGER_CHANNEL PICO_INVALID_CONDITION_CHANNEL PICO_TOO_MANY_SAMPLES PICO_INVALID_TIMEBASE PICO_NOT_RESPONDING PICO_CONFIG_FAIL PICO_INVALID_PARAMETER PICO_NOT_RESPONDING PICO_TRIGGER_ERROR PICO_DRIVER_FUNCTION PICO_FW_FAIL PICO_NOT_ENOUGH_SEGMENTS (in Bulk mode) PICO_PULSE_WIDTH_QUALIFIER PICO_SEGMENT_OUT_OF_RANGE (in Overlapped mode) PICO_STARTINDEX_INVALID (in Overlapped mode) PICO_INVALID_SAMPLERATIO (in Overlapped mode) PICO_CONFIG_FAIL

3.38 ps2000aRunStreaming() – capture in streaming mode

```
PICO_STATUS ps2000aRunStreaming
(
  int16_t                handle,
  uint32_t             * sampleInterval,
  PS2000A_TIME_UNITS     sampleIntervalTimeUnits
  uint32_t               maxPreTriggerSamples,
  uint32_t               maxPostTriggerSamples,
  int16_t                autoStop,
  uint32_t               downSampleRatio,
  PS2000A_RATIO_MODE     downSampleRatioMode,
  uint32_t               overviewBufferSize
)
```

This function tells the oscilloscope to start collecting data in streaming mode. When data has been collected from the device it is downsampled if necessary and then delivered to the application. Call `ps2000aGetStreamingLatestValues()` to retrieve the data. See Using streaming mode for a step-by-step guide to this process.

The function always starts collecting data immediately, regardless of the trigger settings. Whether a trigger is set or not, the total number of samples stored in the driver is always `maxPreTriggerSamples + maxPostTriggerSamples`. If `autoStop` is false, the scope will collect data continuously using the buffer as a first-in first-out (FIFO) memory.

Applicability	Streaming mode

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`* sampleInterval`, on entry, the requested time interval between samples; on exit, the actual time interval used.

`sampleIntervalTimeUnits`, the unit of time used for `sampleInterval`. Use one of these values:
`PS2000A_FS`
`PS2000A_PS`
`PS2000A_NS`
`PS2000A_US`
`PS2000A_MS`
`PS2000A_S`

`maxPreTriggerSamples`, the maximum number of raw samples before a trigger event for each enabled channel.

`maxPostTriggerSamples`, the maximum number of raw samples after a trigger event for each enabled channel.

`autoStop`, a flag that specifies if the streaming should stop when all of `maxSamples = maxPreTriggerSamples + maxPostTriggerSamples` have been captured. This can only happen after a trigger event.

`downSampleRatio`, see `ps2000aGetValues()`

`downSampleRatioMode`, see [`ps2000aGetValues()`]

`overviewBufferSize`, the size of the overview buffers. These are temporary buffers used for storing the data before returning it to the application. The size is the same as the `bufferLth` value passed to [`ps2000aSetDataBuffer()`].

Returns	PICO_OK PICO_INVALID_HANDLE PICO_ETS_MODE_SET PICO_USER_CALLBACK PICO_NULL_PARAMETER PICO_INVALID_PARAMETER PICO_STREAMING_FAILED PICO_NOT_RESPONDING PICO_TRIGGER_ERROR PICO_INVALID_SAMPLE_INTERVAL PICO_INVALID_BUFFER PICO_DRIVER_FUNCTION PICO_FW_FAIL PICO_MEMORY

3.39 ps2000aSetChannel() – set up input channel

```
PICO_STATUS ps2000aSetChannel
(
  int16_t             handle,
  PS2000A_CHANNEL     channel,
  int16_t             enabled,
  PS2000A_COUPLING    type,
  PS2000A_RANGE       range,
  float               analogOffset
)
```

This function specifies whether an input channel is to be enabled, its input coupling type, voltage range, analog offset.

Applicability	All modes

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`channel`, the channel to be configured. The values are:
- `PS2000A_CHANNEL_A`: Channel A input
- `PS2000A_CHANNEL_B`: Channel B input
- `PS2000A_CHANNEL_C`: Channel C input
- `PS2000A_CHANNEL_D`: Channel D input

`enabled`, `TRUE` to enable the channel, `FALSE` to disable it.

`type`, the impedance and coupling type. The values are:
- `PS2000A_AC`: 1 megohm impedance, AC coupling. The channel accepts input frequencies from about 1 hertz up to its maximum analog bandwidth.
- `PS2000A_DC`: 1 megohm impedance, DC coupling. The channel accepts all input frequencies from zero (DC) up to its maximum analog bandwidth.

`range`, the input voltage range:

`PS2000A_20MV`:	±20 mV	`PS2000A_500MV`:	±500 mV	`PS2000A_5V`:	±5 V
`PS2000A_50MV`:	±50 mV	`PS2000A_1V`:	±1 V	`PS2000A_10V`:	±10 V
`PS2000A_100MV`:	±100 mV	`PS2000A_2V`:	±2 V	`PS2000A_20V`:	±20 V
`PS2000A_200MV`:	±200 mV				

`analogOffset`, a voltage to add to the input channel before digitization. The allowable range of offsets can be obtained from `ps2000aGetAnalogueOffset()` and depends on the input range selected for the channel. This argument is ignored if the device is a PicoScope 2205 MSO.

Returns	`PICO_OK`
	`PICO_USER_CALLBACK`
	`PICO_INVALID_HANDLE`
	`PICO_INVALID_CHANNEL`
	`PICO_INVALID_VOLTAGE_RANGE`
	`PICO_INVALID_COUPLING`
	`PICO_INVALID_ANALOGUE_OFFSET`
	`PICO_DRIVER_FUNCTION`

3.40 ps2000aSetDataBuffer() – register data buffer with driver

```
PICO_STATUS ps2000aSetDataBuffer
(
  int16_t                handle,
  int32_t                channel,
  int16_t              * buffer,
  int32_t                bufferLth,
  uint32_t               segmentIndex,
  PS2000A_RATIO_MODE     mode
)
```

This function tells the driver where to store the data, either unprocessed or downsampled, that will be returned after the next call to one of the `ps2000aGetValues...()` functions. The function only allows you to specify a single buffer, so for aggregation mode, which requires two buffers, you need to call `ps2000aSetDataBuffers()` instead.

You must allocate memory for the buffer before calling this function.

Applicability	Block, rapid block and streaming modes. All downsampling modes except aggregation.

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`channel`, the channel you want to use with the buffer. Use one of these values:
`PS2000A_CHANNEL_A`
`PS2000A_CHANNEL_B`
`PS2000A_CHANNEL_C`
`PS2000A_CHANNEL_D`
`PS2000A_DIGITAL_PORT0 = 0x80` (MSO models only)
`PS2000A_DIGITAL_PORT1 = 0x81` (MSO models only)

`buffer`, pointer to the buffer. Each sample written to the buffer will be a 16-bit ADC count scaled according to the selected voltage range.

`bufferLth`, the size of the `buffer` array

`segmentIndex`, the number of the memory segment to be used

`mode`, the downsampling mode. See `ps2000aGetValues()` for the available modes, but note that a single call to `ps2000aSetDataBuffer()` can only associate one buffer with one downsampling mode. If you intend to call `ps2000aGetValues()` with more than one downsampling mode activated, then you must call `ps2000aSetDataBuffer()` several times to associate a separate buffer with each downsampling mode.

Returns	`PICO_OK`
	`PICO_INVALID_HANDLE`
	`PICO_INVALID_CHANNEL`
	`PICO_RATIO_MODE_NOT_SUPPORTED`
	`PICO_SEGMENT_OUT_OF_RANGE`
	`PICO_DRIVER_FUNCTION`
	`PICO_INVALID_PARAMETER`

3.41 ps2000aSetDataBuffers() – register aggregated data buffers with driver

```
PICO_STATUS ps2000aSetDataBuffers
(
  int16_t               handle,
  int32_t               channel,
  int16_t             * bufferMax,
  int16_t             * bufferMin,
  int32_t               bufferLth,
  uint32_t              segmentIndex,
  PS2000A_RATIO_MODE    mode
)
```

This function tells the driver the location of one or two buffers for receiving data. You need to allocate memory for the buffers before calling this function. If you do not need two buffers because you are not using aggregate mode, you can optionally use ps2000aSetDataBuffer() instead.

Applicability	Block and streaming modes with aggregation.

Arguments

handle, device identifier returned by ps2000aOpenUnit().

channel, the channel for which you want to set the buffers. Use one of these constants:
  PS2000A_CHANNEL_A
  PS2000A_CHANNEL_B
  PS2000A_CHANNEL_C
  PS2000A_CHANNEL_D
  PS2000A_DIGITAL_PORT0 = 0x80 (MSO models only)
  PS2000A_DIGITAL_PORT1 = 0x81 (MSO models only)

bufferMax, a user-allocated buffer to receive the maximum data values in aggregation mode, or the non-aggregated values otherwise. Each value is a 16-bit ADC count scaled according to the selected voltage range.

bufferMin, a user-allocated buffer to receive the minimum data values in aggregation mode. Not normally used in other modes, but you can direct the driver to write non-aggregated values to this buffer by setting bufferMax to NULL. To enable aggregation, the downsampling ratio and mode must be set appropriately when calling one of the ps2000aGetValues...() functions.

bufferLth, the size of the bufferMax and bufferMin arrays.

segmentIndex, the number of the memory segment to be used.

mode, see ps2000aGetValues().

Returns	PICO_OK PICO_INVALID_HANDLE PICO_INVALID_CHANNEL PICO_RATIO_MODE_NOT_SUPPORTED PICO_SEGMENT_OUT_OF_RANGE PICO_DRIVER_FUNCTION PICO_INVALID_PARAMETER

3.42 ps2000aSetDigitalAnalogTriggerOperand() – set up combined analog/digital trigger

```
PICO_STATUS ps2000aSetDigitalAnalogTriggerOperand
(
  int16_t                   handle,
  PS2000A_TRIGGER_OPERAND   operand
)
```

Mixed-signal (MSO) models in this series have two independent triggers, one for the analog input channels and another for the digital inputs. This function defines how the two triggers are combined.

Applicability	MSO models only

Arguments

handle, device identifier returned by ps2000aOpenUnit().

operand, one of the following constants:
  PS2000A_OPERAND_NONE, ignore the trigger settings
  PS2000A_OPERAND_OR, fire when either trigger is activated
  PS2000A_OPERAND_AND, fire when both triggers are activated
  PS2000A_OPERAND_THEN, fire when one trigger is activated followed by the other

Returns	PICO_OK PICO_INVALID_HANDLE PICO_DRIVER_FUNCTION PICO_NOT_USED PICO_INVALID_PARAMETER

3.43 ps2000aSetDigitalPort() – set up digital input

```
PICO_STATUS ps2000aSetDigitalPort
(
  int16_t                handle,
  PS2000A_DIGITAL_PORT   port,
  int16_t                enabled,
  int16_t                logiclevel
)
```

This function is used to enable the digital ports of an MSO and set the logic level (the voltage point at which the state transitions from 0 to 1).

Applicability	MSO devices only. Block and streaming modes with aggregation. Not compatible with ETS mode.

Arguments

`handle,` device identifier returned by `ps2000aOpenUnit()`.

`port,` the digital port to be configured:
  `PS2000A_DIGITAL_PORT0 = 0x80` (D0 to D7)
  `PS2000A_DIGITAL_PORT1 = 0x81` (D8 to D15)

`enabled,` whether or not to enable the port. Enabling a digital port allows the scope to collect data from the port and to trigger on the port. Enabling a digital port may also increase the memory usage of the scope (see Block mode). The values are:
  `TRUE:` enable
  `FALSE:` do not enable

`logiclevel,` the logic threshold voltage
  Range: –32767 (–5 V) to 32767 (+5 V).

Returns	`PICO_OK`
	`PICO_INVALID_HANDLE`
	`PICO_INVALID_CHANNEL`
	`PICO_RATIO_MODE_NOT_SUPPORTED`
	`PICO_SEGMENT_OUT_OF_RANGE`
	`PICO_DRIVER_FUNCTION`
	`PICO_INVALID_PARAMETER`

3.44 ps2000aSetEts() – set up equivalent-time sampling

```
PICO_STATUS ps2000aSetEts
(
  int16_t                  handle,
  PS2000A_ETS_MODE         mode,
  int16_t                  etsCycles,
  int16_t                  etsInterleave,
  int32_t                * sampleTimePicoseconds
)
```

This function is used to enable or disable ETS (equivalent-time sampling) and to set the ETS parameters. See ETS overview for an explanation of ETS mode.

Applicability	Block mode only. On MSOs, ETS mode not available when digital port(s) enabled.

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`mode`, the ETS mode. Use one of these values:

`PS2000A_ETS_OFF`:	disables ETS
`PS2000A_ETS_FAST`:	enables ETS and provides `etsCycles` of data, which may contain data from previously returned cycles
`PS2000A_ETS_SLOW`:	enables ETS and provides fresh data every `etsCycles`. This mode takes longer to provide each data set, but the data sets are more stable and are guaranteed to contain only new data.

`etsCycles`, the number of cycles to store. The computer can then select `etsInterleave` cycles to give the most uniform spread of samples. Maximum values are:

`500` for the PicoScope 2206B, 2206B MSO, 2207B, 2207B MSO, 2208B, 2208B MSO, 2405A, 2406B, 2407B, 2408B
`PS2206_MAX_ETS_CYCLES` for the PicoScope 2206, 2206A
`PS2207_MAX_ETS_CYCLES` for the PicoScope 2207, 2207A
`PS2208_MAX_ETS_CYCLES` for the PicoScope 2208, 2208A

`etsInterleave`, the number of waveforms to combine into a single ETS capture. Maximum values are:

`40` for the PicoScope 2206B, 2206B MSO, 2207B, 2207B MSO, 2208B, 2208B MSO, 2405A, 2406B, 2407B, 2408B
`PS2206_MAX_INTERLEAVE` for the PicoScope 2206, 2206A
`PS2207_MAX_INTERLEAVE` for the PicoScope 2207, 2207A
`PS2208_MAX_INTERLEAVE` for the PicoScope 2208, 2208A

`* sampleTimePicoseconds`, on exit, the effective sampling interval of the ETS data. For example, if the captured sample time is 4 ns and `etsInterleave` is 10, then the effective sample time in ETS mode is 400 ps.

Returns	`PICO_OK`
	`PICO_USER_CALLBACK`
	`PICO_INVALID_HANDLE`
	`PICO_INVALID_PARAMETER`
	`PICO_DRIVER_FUNCTION`

3.45 ps2000aSetEtsTimeBuffer() – set up 64-bit buffer for ETS timings

```
PICO_STATUS ps2000aSetEtsTimeBuffer
(
  int16_t       handle,
  int64_t     * buffer,
  int32_t       bufferLth
)
```

This function tells the driver where to find your application's ETS time buffers. These buffers contain the 64-bit timing information for each ETS sample after you run a block-mode ETS capture.

Applicability	ETS mode only. If your programming language does not support 64-bit data, use the 32-bit version `ps2000aSetEtsTimeBuffers()` instead.

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`* buffer`, an array of 64-bit words, each representing the time in femtoseconds (10^{-15} s) at which the sample was captured.

`bufferLth`, the size of the buffer array.

Returns	`PICO_OK`
	`PICO_INVALID_HANDLE`
	`PICO_NULL_PARAMETER`
	`PICO_DRIVER_FUNCTION`

3.46 ps2000aSetEtsTimeBuffers() – set up 32-bit buffers for ETS timings

```
PICO_STATUS ps2000aSetEtsTimeBuffers
(
  int16_t       handle,
  uint32_t    * timeUpper,
  uint32_t    * timeLower,
  int32_t       bufferLth
)
```

This function tells the driver where to find your application's ETS time buffers. These buffers contain the timing information for each ETS sample after you run a block-mode ETS capture. There are two buffers containing the upper and lower 32-bit parts of the timing information, to allow programming languages that do not support 64-bit data to retrieve the timings.

Applicability	ETS mode only. If your programming language supports 64-bit data then you can use `ps2000aSetEtsTimeBuffer()` instead.

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`* timeUpper`, an array of 32-bit words, each representing the upper 32 bits of the time in femtoseconds (10^{-15} s) at which the sample was captured

`* timeLower`, an array of 32-bit words, each representing the lower 32 bits of the time in femtoseconds (10^{-15} s) at which the sample was captured

`bufferLth`, the size of the `timeUpper` and `timeLower` arrays.

Returns	`PICO_OK`
	`PICO_INVALID_HANDLE`
	`PICO_NULL_PARAMETER`
	`PICO_DRIVER_FUNCTION`

3.47 ps2000aSetNoOfCaptures() – set number of captures to collect in one run

```
PICO_STATUS ps2000aSetNoOfCaptures
(
  int16_t     handle,
  uint32_t    nCaptures
)
```

This function sets the number of captures to be collected in one run of rapid block mode. If you do not call this function before a run, the driver will capture only one waveform. Once a value has been set, the value remains constant unless changed.

Applicability	Rapid block mode

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`nCaptures`, the number of waveforms to capture in one run.

Returns	`PICO_OK`
	`PICO_INVALID_HANDLE`
	`PICO_INVALID_PARAMETER`
	`PICO_DRIVER_FUNCTION`

3.48 ps2000aSetOutputEdgeDetect() – enable or disable state trigger edge-detection

```
PICO_STATUS ps2000aSetOutputEdgeDetect
(
  int16_t     handle,
  int16_t     state
)
```

This function tells the device whether or not to wait for an edge on the trigger input when one of the 'level' or 'window' trigger types is in use. By default the device waits for an edge on the trigger input before firing the trigger. If you switch off edge detect mode, the device will trigger continually for as long as the trigger input remains in the specified state.

You can query the state of this flag by calling `ps2000aQueryOutputEdgeDetect()`.

Applicability	Level and window trigger types

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`state`, a flag that specifies the trigger behavior:

`0` : do not wait for a signal transition

`<> 0` : wait for a signal transition (default)

Returns	`PICO_OK`

3.49 ps2000aSetPulseWidthDigitalPortProperties() – set pulse-width triggering on digital inputs

```
PICO_STATUS ps2000aSetPulseWidthDigitalPortProperties
(
  int16_t                               handle,
  PS2000A_DIGITAL_CHANNEL_DIRECTIONS  * directions
  int16_t                               nDirections
)
```

This function will set the individual digital channels' pulse-width trigger directions. Each trigger direction consists of a channel name and a direction. If the channel is not included in the array of PS2000A_DIGITAL_CHANNEL_DIRECTIONS the driver assumes the digital channel's pulse-width trigger direction is PS2000A_DIGITAL_DONT_CARE.

Applicability	All modes. MSO models only.

Arguments

handle, device identifier returned by ps2000aOpenUnit().

* directions, a pointer to an array of PS2000A_DIGITAL_CHANNEL_DIRECTIONS structures describing the requested properties. The array can contain a single element describing the properties of one channel, or a number of elements describing several digital channels. If directions is NULL, digital pulse-width triggering is switched off. A digital channel that is not included in the array will be set to PS2000A_DIGITAL_DONT_CARE.

nDirections, the number of digital channel directions being passed to the driver.

Returns	PICO_OK PICO_INVALID_HANDLE PICO_DRIVER_FUNCTION PICO_INVALID_DIGITAL_CHANNEL PICO_INVALID_DIGITAL_TRIGGER_DIRECTION

3.50 ps2000aSetPulseWidthQualifier() – set up pulse width triggering

```
PICO_STATUS ps2000aSetPulseWidthQualifier
(
  int16_t                      handle,
  PS2000A_PWQ_CONDITIONS     * conditions,
  int16_t                      nConditions,
  PS2000A_THRESHOLD_DIRECTION  direction,
  uint32_t                     lower,
  uint32_t                     upper,
  PS2000A_PULSE_WIDTH_TYPE     type
)
```

This function sets up pulse-width qualification, which can be used on its own for pulse-width triggering or combined with threshold triggering, level triggering or window triggering to produce more complex triggers. The pulse-width qualifier is set by defining one or more structures that are then ORed together. Each structure is itself the AND of the states of one or more of the inputs. This AND-OR logic allows you to create any possible Boolean function of the scope's inputs.

Note: The oscilloscope contains a single pulse-width counter. It is possible to include multiple channels in a pulse-width qualifier but the same pulse-width counter will apply to all of them. The counter starts when your selected trigger condition occurs, and the scope then triggers if the trigger condition ends after a time that satisfies the pulse-width condition.

Applicability	All modes

Arguments

handle, device identifier returned by ps2000aOpenUnit().

* conditions, an array of PS2000A_PWQ_CONDITIONS structures specifying the conditions that should be applied to each channel. In the simplest case, the array consists of a single element. When there are several elements, the overall trigger condition is the logical OR of all the elements. Since each element can combine a number of input conditions using AND logic, this AND-OR logic enables you to create a qualifier based on any possible Boolean function of the inputs. If conditions is NULL, the pulse-width qualifier is not used.

nConditions, the number of elements in the conditions array. If nConditions is zero then the pulse-width qualifier is not used.
Range: 0 to PS2000A_MAX_PULSE_WIDTH_QUALIFIER_COUNT.

direction, the direction of the signal required for the pulse width trigger to fire. See PS2000A_THRESHOLD_DIRECTION constants for the list of possible values. Each channel of the oscilloscope (except the **EXT** input, if present) has two thresholds for each direction—for example, PS2000A_RISING and PS2000A_RISING_LOWER— so that one can be used for the pulse-width qualifier and the other for the level trigger. The driver will not let you use the same threshold for both triggers; so, for example, you cannot use PS2000A_RISING as the direction argument for both ps2000aSetTriggerConditions() and ps2000aSetPulseWidthQualifier() at the same time. There is no such restriction when using window triggers.

lower, the lower limit of the pulse-width counter, measured in sample periods

upper, the upper limit of the pulse-width counter, measured in sample periods. This parameter is used only when the type is set to PS2000A_PW_TYPE_IN_RANGE or PS2000A_PW_TYPE_OUT_OF_RANGE.

`type`, the pulse-width type, one of these constants:

- `PS2000A_PW_TYPE_NONE`: do not use the pulse width qualifier
- `PS2000A_PW_TYPE_LESS_THAN`: pulse width less than `lower`
- `PS2000A_PW_TYPE_GREATER_THAN`: pulse width greater than `lower`
- `PS2000A_PW_TYPE_IN_RANGE`: pulse width between `lower` and `upper`
- `PS2000A_PW_TYPE_OUT_OF_RANGE`: pulse width not between `lower` and `upper`

Returns	`PICO_OK`
	`PICO_INVALID_HANDLE`
	`PICO_USER_CALLBACK`
	`PICO_CONDITIONS`
	`PICO_PULSE_WIDTH_QUALIFIER`
	`PICO_DRIVER_FUNCTION`

3.50.1 PS2000A_PWQ_CONDITIONS structure

A structure of this type is passed to `ps2000aSetPulseWidthQualifier()` in the `conditions` argument to specify a set of trigger conditions. It is defined as follows:

```
typedef struct tPS2000APwqConditions
{
  PS2000A_TRIGGER_STATE channelA;
  PS2000A_TRIGGER_STATE channelB;
  PS2000A_TRIGGER_STATE channelC;
  PS2000A_TRIGGER_STATE channelD;
  PS2000A_TRIGGER_STATE external;
  PS2000A_TRIGGER_STATE aux;
  PS2000A_TRIGGER_STATE digital;
} PS2000A_PWQ_CONDITIONS
```

A structure of this type is passed to ps2000aSetPulseWidthQualifier in the conditions argument to specify the pulse-width qualifier conditions for all the trigger sources. Each structure is the logical AND of the states of the scope's inputs. The ps2000aSetPulseWidthQualifier function can OR together a number of these structures to produce the final pulse width qualifier, which can therefore be any possible Boolean function of the scope's inputs.

The structure is byte-aligned. In C++, for example, you should specify this using the `#pragma pack()` instruction.

Elements

`channelA, channelB, channelC, channelD, external`: the type of condition that should be applied to each channel. Use these constants:

`PS2000A_CONDITION_DONT_CARE`
`PS2000A_CONDITION_TRUE`
`PS2000A_CONDITION_FALSE`

The channels that are set to `PS2000A_CONDITION_TRUE` or `PS2000A_CONDITION_FALSE` must all meet their conditions simultaneously to produce a trigger. Channels set to `PS2000A_CONDITION_DONT_CARE` are ignored.

`aux, digital`: not used.

3.51 ps2000aSetSigGenArbitrary() – set up arbitrary waveform generator

```
PICO_STATUS ps2000aSetSigGenArbitrary
(
  int16_t                        handle,
  int32_t                        offsetVoltage,
  uint32_t                       pkToPk
  uint32_t                       startDeltaPhase,
  uint32_t                       stopDeltaPhase,
  uint32_t                       deltaPhaseIncrement,
  uint32_t                       dwellCount,
  int16_t                      * arbitraryWaveform,
  int32_t                        arbitraryWaveformSize,
  PS2000A_SWEEP_TYPE             sweepType,
  PS2000A_EXTRA_OPERATIONS       operation,
  PS2000A_INDEX_MODE             indexMode,
  uint32_t                       shots,
  uint32_t                       sweeps,
  PS2000A_SIGGEN_TRIG_TYPE       triggerType,
  PS2000A_SIGGEN_TRIG_SOURCE     triggerSource,
  int16_t                        extInThreshold
)
```

This function programs the signal generator to produce an arbitrary waveform.

The arbitrary waveform generator uses direct digital synthesis (DDS). It maintains a 32-bit phase accumulator that indicates the present location in the waveform. The top bits of the phase accumulator are used as an index into a buffer containing the arbitrary waveform. The remaining bits act as the fractional part of the index, enabling high-resolution control of output frequency and allowing the generation of lower frequencies.

The phase accumulator initially increments by `startDeltaPhase`. If the AWG is set to sweep mode, the phase increment is increased at specified intervals until it reaches `stopDeltaPhase`. The easiest way to obtain the values of `startDeltaPhase` and `stopDeltaPhase` necessary to generate the desired frequency is to call `ps2000aSigGenFrequencyToPhase()`. Alternatively, see Calculating deltaPhase below for more information on how to calculate these values.

Set up the signal generator before starting data acquisition, particularly if you require it to be triggered during data acquisition.

This document provides some useful guidance on how to call the API functions in order to trigger the signal generator output.

Applicability	All modes

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`offsetVoltage`, the voltage offset, in microvolts, to be applied to the waveform

`pkToPk`, the peak-to-peak voltage, in microvolts, of the waveform signal startDeltaPhase, the initial value added to the phase accumulator as the generator begins to step through the waveform buffer. Calculate this value from the information above, or use ps2000aSigGenFrequencyToPhase().

stopDeltaPhase, the final value added to the phase accumulator before the generator restarts or reverses the sweep. When frequency sweeping is not required, set equal to startDeltaPhase.

deltaPhaseIncrement, the amount added to the delta phase value every time the dwellCount period expires. This determines the amount by which the generator sweeps the output frequency in each dwell period. When frequency sweeping is not required, set to zero.

dwellCount, the time, in multiples of *ddsPeriod*, between successive additions of deltaPhaseIncrement to the delta phase accumulator. This determines the rate at which the generator sweeps the output frequency. Minimum value: PS2000A_MIN_DWELL_COUNT

* arbitraryWaveform, a buffer that holds the waveform pattern as a set of samples equally spaced in time. Each sample is scaled to an output voltage as follows:

$$v_{OUT} = 1\ \mu V \times (\texttt{pkToPk} / 2) \times (sample\_value / 32767) + \texttt{offsetVoltage}$$

and clipped to the overall ±2 V range of the AWG.

arbitraryWaveformSize, the size of the arbitrary waveform buffer, in samples, in the range:
[minArbitraryWaveformSize, maxArbitraryWaveformSize]
where minArbitraryWaveformSize and maxArbitraryWaveformSize are the values returned by ps2000aSigGenArbitraryMinMaxValues().

sweepType, determines whether the startDeltaPhase is swept up to the stopDeltaPhase, or down to it, or repeatedly swept up and down. Use one of these values:
PS2000A_UP
PS2000A_DOWN
PS2000A_UPDOWN
PS2000A_DOWNUP

operation, the type of waveform to be produced, specified by one of the following enumerated types:
PS2000A_ES_OFF, normal AWG operation using the waveform buffer.
PS2000A_WHITENOISE, the signal generator produces white noise and ignores all settings except offsetVoltage and pkToPk.
PS2000A_PRBS, produces a random bitstream with a bit rate specified by the phase accumulator.

indexMode, specifies how the signal will be formed from the arbitrary waveform data. Single and dual index modes are possible. Use one of these constants:
PS2000A_SINGLE
PS2000A_DUAL

shots,
0: sweep the frequency as specified by sweeps
1...PS2000A_MAX_SWEEPS_SHOTS: the number of cycles of the waveform to be produced after a trigger event. sweeps must be zero.
PS2000A_SHOT_SWEEP_TRIGGER_CONTINUOUS_RUN: start and run continuously after trigger occurs (not PicoScope 2205 MSO)

sweeps,
0: produce number of cycles specified by shots

1..PS2000A_MAX_SWEEPS_SHOTS: the number of times to sweep the frequency after a trigger event, according to sweepType. shots must be zero.
PS2000A_SHOT_SWEEP_TRIGGER_CONTINUOUS_RUN: start a sweep and continue after trigger occurs (not PicoScope 2205 MSO)

triggerType, the type of trigger that will be applied to the signal generator:

PS2000A_SIGGEN_RISING	trigger on rising edge
PS2000A_SIGGEN_FALLING	trigger on falling edge
PS2000A_SIGGEN_GATE_HIGH	run while trigger is high
PS2000A_SIGGEN_GATE_LOW	run while trigger is low

A trigger event causes the signal generator to produce the specified number of shots or sweeps.

triggerSource, the source that will trigger the signal generator:

PS2000A_SIGGEN_NONE	run without waiting for trigger
PS2000A_SIGGEN_SCOPE_TRIG	use scope trigger
PS2000A_SIGGEN_EXT_IN	use **EXT** input (if available)
PS2000A_SIGGEN_SOFT_TRIG	wait for software trigger provided by ps2000aSigGenSoftwareControl()
PS2000A_SIGGEN_TRIGGER_RAW	reserved

When triggering is enabled (trigger source set to something other than P2000A_SIGGEN_NONE), either shots or sweeps, but not both, must be non-zero.

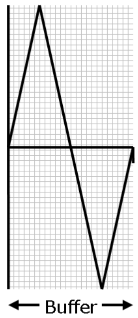
extInThreshold, trigger level, in ADC counts, for external trigger.

Returns	PICO_OK PICO_AWG_NOT_SUPPORTED PICO_BUSY PICO_INVALID_HANDLE PICO_SIG_GEN_PARAM PICO_SHOTS_SWEEPS_WARNING PICO_NOT_RESPONDING PICO_WARNING_EXT_THRESHOLD_CONFLICT PICO_NO_SIGNAL_GENERATOR PICO_SIGGEN_OFFSET_VOLTAGE PICO_SIGGEN_PK_TO_PK PICO_SIGGEN_OUTPUT_OVER_VOLTAGE PICO_DRIVER_FUNCTION PICO_SIGGEN_WAVEFORM_SETUP_FAILED

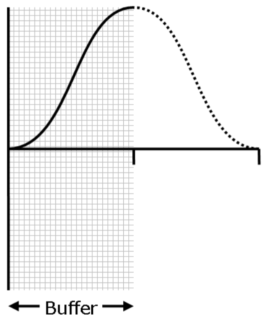
3.51.1 AWG index modes

The arbitrary waveform generator supports **single** and **dual** index modes to help you make the best use of the waveform buffer.

Single mode. The generator outputs the raw contents of the buffer repeatedly. This mode is the only one that can generate asymmetrical waveforms. You can also use this mode for symmetrical waveforms but the dual mode makes more efficient use of the buffer memory.



Dual mode. The generator outputs the contents of the buffer from beginning to end, and then does a second pass in the reverse direction through the buffer. This allows you to specify only the first half of a waveform with twofold symmetry, such as a Gaussian function, and let the generator fill in the other half.



3.51.2 Calculating deltaPhase

The arbitrary waveform generator (AWG) steps through the waveform buffer by adding a *deltaPhase* value between 1 and *phaseAccumulatorSize-1* to the phase accumulator every *ddsPeriod* (*1/ddsFrequency*). If the *deltaPhase* is constant, the generator produces a waveform at a constant frequency that can be calculated as follows:

$$outputFrequency = ddsFrequency \times \left(\frac{deltaPhase}{phaseAccumulatorSize}\right) \times \left(\frac{awgBufferSize}{arbitraryWaveformSize}\right)$$

where:

- *outputFrequency* = repetition rate of the complete arbitrary waveform
- *ddsFrequency* = update rate of DDS counter for each model
- *deltaPhase* = calculated from *startDeltaPhase* and *deltaPhaseIncrement* (we recommend that you use ps2000aSigGenFrequencyToPhase() to calculate *deltaPhase*)
- *phaseAccumulatorSize* = 2^{32} for all models
- *awgBufferSize* = AWG buffer size for each model
- *arbitraryWaveformSize* = length in samples of the user-defined waveform

It is also possible to sweep the frequency by continually modifying the *deltaPhase*. This is done by setting up a `deltaPhaseIncrement` that the oscilloscope adds to the *deltaPhase* at intervals specified by `dwellCount`.

Parameter	**PicoScope 2205 MSO**	**PicoScope 2205A MSO 2206/2206A 2207/2207A 2208/2208A 2405A**	**PicoScope 2206B/2206B MSO 2207B/2207B MSO 2208B/2208B MSO 2406B 2407B 2408B**
phaseAccumulatorSize	2^{32}	2^{32}	2^{32}
ddsFrequency	48 MHz	20 MHz	20 MHz
awgBufferSize	8192 samples	8192 samples	32 768 samples
ddsPeriod (= 1/ddsFrequency)	20.83 ns	50 ns	50 ns

3.52 ps2000aSetSigGenBuiltIn() – set up standard signal generator

```
PICO_STATUS ps2000aSetSigGenBuiltIn
(
  int16_t                       handle,
  int32_t                       offsetVoltage,
  uint32_t                      pkToPk
  int16_t                       waveType
  float                         startFrequency,
  float                         stopFrequency,
  float                         increment,
  float                         dwellTime,
  PS2000A_SWEEP_TYPE            sweepType,
  PS2000A_EXTRA_OPERATIONS      operation,
  uint32_t                      shots,
  uint32_t                      sweeps,
  PS2000A_SIGGEN_TRIG_TYPE      triggerType,
  PS2000A_SIGGEN_TRIG_SOURCE    triggerSource,
  int16_t                       extInThreshold
)
```

This function sets up the signal generator to produce a signal from a list of built-in waveforms. If different start and stop frequencies are specified, the device will sweep either up, down, or up and down.

Set up the signal generator before starting data acquisition, particularly if you require it to be triggered during data acquisition.

Applicability	All modes

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`offsetVoltage`, the voltage offset, in microvolts, to be applied to the waveform

`pkToPk`, the peak-to-peak voltage, in microvolts, of the waveform signal

Note: if the signal voltages described by the combination of `offsetVoltage` and `pkToPk` extend outside the voltage range of the signal generator, the output waveform will be clipped.

`waveType`, the type of waveform to be generated:

`PS2000A_SINE`	sine wave
`PS2000A_SQUARE`	square wave
`PS2000A_TRIANGLE`	triangle wave
`PS2000A_DC_VOLTAGE`	DC voltage
`PS2000A_RAMP_UP`	rising sawtooth
`PS2000A_RAMP_DOWN`	falling sawtooth
`PS2000A_SINC`	sin(x)/x
`PS2000A_GAUSSIAN`	Gaussian
`PS2000A_HALF_SINE`	half (full-wave rectified) sine

`startFrequency`, the frequency that the signal generator will initially produce. Allowable values are between one of these constants:

PS2000A_MIN_FREQUENCY
PS2000A_PRBS_MIN_FREQUENCY

and one of these constants:

PS2000A_SINE_MAX_FREQUENCY
PS2000A_SQUARE_MAX_FREQUENCY
PS2000A_TRIANGLE_MAX_FREQUENCY
PS2000A_SINC_MAX_FREQUENCY
PS2000A_RAMP_MAX_FREQUENCY
PS2000A_HALF_SINE_MAX_FREQUENCY
PS2000A_GAUSSIAN_MAX_FREQUENCY
PS2000A_PRBS_MAX_FREQUENCY

depending on the signal type.

stopFrequency, the frequency at which the sweep reverses direction or returns to the initial frequency

increment, the amount of frequency increase or decrease in sweep mode

dwellTime, the time for which the sweep stays at each frequency, in seconds

sweepType, whether the frequency will sweep from startFrequency to stopFrequency, or in the opposite direction, or repeatedly reverse direction. Use one of these constants:
PS2000A_UP
PS2000A_DOWN
PS2000A_UPDOWN
PS2000A_DOWNUP

operation, the type of waveform to be produced, specified by one of the following enumerated types:
PS2000A_ES_OFF, normal signal generator operation specified by waveType.
PS2000A_WHITENOISE, the signal generator produces white noise and ignores all settings except pkToPk and offsetVoltage.
PS2000A_PRBS, produces a pseudorandom binary sequence at the specified frequency or frequency range *(not available on PicoScope 2205 MSO)*.

shots, see ps2000aSigGenArbitrary()
sweeps, see ps2000aSigGenArbitrary()
triggerType, see ps2000aSigGenArbitrary()
triggerSource, see ps2000aSigGenArbitrary()
extInThreshold, see ps2000aSigGenArbitrary()

Returns	PICO_OK PICO_BUSY PICO_INVALID_HANDLE PICO_SIG_GEN_PARAM PICO_SHOTS_SWEEPS_WARNING PICO_NOT_RESPONDING PICO_WARNING_AUX_OUTPUT_CONFLICT PICO_WARNING_EXT_THRESHOLD_CONFLICT PICO_NO_SIGNAL_GENERATOR PICO_SIGGEN_OFFSET_VOLTAGE PICO_SIGGEN_PK_TO_PK PICO_SIGGEN_OUTPUT_OVER_VOLTAGE

	PICO_DRIVER_FUNCTION PICO_SIGGEN_WAVEFORM_SETUP_FAILED PICO_NOT_RESPONDING

3.53 ps2000aSetSigGenBuiltInV2() – double-precision signal generator setup

```
PICO_STATUS ps2000aSetSigGenBuiltInV2
(
  int16_t                      handle,
  int32_t                      offsetVoltage,
  uint32_t                     pkToPk
  int16_t                      waveType
  double                       startFrequency,
  double                       stopFrequency,
  double                       increment,
  double                       dwellTime,
  PS2000_SWEEP_TYPE            sweepType,
  PS2000_EXTRA_OPERATIONS      operation,
  uint32_t                     shots,
  uint32_t                     sweeps,
  PS2000_SIGGEN_TRIG_TYPE      triggerType,
  PS2000_SIGGEN_TRIG_SOURCE    triggerSource,
  int16_t                      extInThreshold
)
```

This function sets up the signal generator. It differs from `ps2000aSetSigGenBuiltIn()` in having double-precision arguments instead of floats, giving greater resolution when setting the output frequency.

This document provides some useful guidance on how to call the API functions in order to trigger the signal generator output.

Applicability	All modes

Arguments
See `ps2000aSetSigGenBuiltIn()`

Returns	See `ps2000aSetSigGenBuiltIn()`

3.54 ps2000aSetSigGenPropertiesArbitrary() – change AWG properties

```
PICO_STATUS ps2000aSetSigGenPropertiesArbitrary
(
  int16_t                      handle,
  uint32_t                     startDeltaPhase,
  uint32_t                     stopDeltaPhase,
  uint32_t                     deltaPhaseIncrement,
  uint32_t                     dwellCount,
  PS2000A_SWEEP_TYPE           sweepType,
  uint32_t                     shots,
  uint32_t                     sweeps,
  PS2000A_SIGGEN_TRIG_TYPE     triggerType,
  PS2000A_SIGGEN_TRIG_SOURCE   triggerSource,
  int16_t                      extInThreshold
)
```

This function reprograms the arbitrary waveform generator. All values can be reprogrammed while the signal generator is waiting for a trigger.

Applicability	All modes

Arguments
See ps2000aSetSigGenArbitrary().

Returns	PICO_OK if successful PICO_INVALID_HANDLE PICO_NO_SIGNAL_GENERATOR PICO_DRIVER_FUNCTION PICO_AWG_NOT_SUPPORTED PICO_SIGGEN_OFFSET_VOLTAGE PICO_SIGGEN_PK_TO_PK PICO_SIGGEN_OUTPUT_OVER_VOLTAGE PICO_SIG_GEN_PARAM PICO_SHOTS_SWEEPS_WARNING PICO_WARNING_EXT_THRESHOLD_CONFLICT PICO_BUSY PICO_SIGGEN_WAVEFORM_SETUP_FAILED PICO_NOT_RESPONDING

3.55 ps2000aSetSigGenPropertiesBuiltIn() – change standard signal generator properties

```
PICO_STATUS ps2000aSetSigGenPropertiesBuiltIn
(
  int16_t                       handle,
  double                        startFrequency,
  double                        stopFrequency,
  double                        increment,
  double                        dwellTime,
  PS2000A_SWEEP_TYPE            sweepType,
  uint32_t                      shots,
  uint32_t                      sweeps,
  PS2000A_SIGGEN_TRIG_TYPE      triggerType,
  PS2000A_SIGGEN_TRIG_SOURCE    triggerSource,
  int16_t                       extInThreshold
)
```

This function reprograms the signal generator. Values can be changed while the signal generator is waiting for a trigger.

Applicability	All modes

Arguments
See ps2000aSetSigGenBuiltIn().

Returns	PICO_OK if successful PICO_INVALID_HANDLE PICO_NO_SIGNAL_GENERATOR PICO_DRIVER_FUNCTION PICO_WARNING_EXT_THRESHOLD_CONFLICT PICO_SIGGEN_OFFSET_VOLTAGE PICO_SIGGEN_PK_TO_PK PICO_SIGGEN_OUTPUT_OVER_VOLTAGE PICO_SIG_GEN_PARAM PICO_SHOTS_SWEEPS_WARNING PICO_WARNING_EXT_THRESHOLD_CONFLICT PICO_BUSY PICO_SIGGEN_WAVEFORM_SETUP_FAILED PICO_NOT_RESPONDING

3.56 ps2000aSetSimpleTrigger() – set up level triggers

```
PICO_STATUS ps2000aSetSimpleTrigger
(
  int16_t                       handle,
  int16_t                       enable,
  PS2000A_CHANNEL               source,
  int16_t                       threshold,
  PS2000A_THRESHOLD_DIRECTION   direction,
  uint32_t                      delay,
  int16_t                       autoTrigger_ms
)
```

This function simplifies arming the trigger. It supports only the LEVEL trigger types on analog channels, and does not allow more than one channel to have a trigger applied to it. Any previous pulse width qualifier is canceled. The trigger threshold includes a small, fixed amount of hysteresis.

Applicability	All modes

Arguments

handle, device identifier returned by ps2000aOpenUnit().

enable, zero to disable the trigger; any non-zero value to set the trigger.

source, the channel on which to trigger.

threshold, the ADC count at which the trigger will fire.

direction, the direction in which the signal must move to cause a trigger. The following directions are supported: ABOVE, BELOW, RISING, FALLING and RISING_OR_FALLING.

delay, the time between the trigger occurring and the first sample being taken. For example, if delay=100, the scope would wait 100 sample periods before sampling.

autoTrigger_ms, the number of milliseconds the device will wait if no trigger occurs. If this is set to zero, the scope device will wait indefinitely for a trigger.

Returns	PICO_OK PICO_INVALID_CHANNEL PICO_INVALID_PARAMETER PICO_MEMORY PICO_CONDITIONS PICO_INVALID_HANDLE PICO_USER_CALLBACK PICO_DRIVER_FUNCTION

3.57 ps2000aSetTriggerChannelConditions() – specify which channels to trigger on

```
PICO_STATUS ps2000aSetTriggerChannelConditions
(
  int16_t                        handle,
  PS2000A_TRIGGER_CONDITIONS   * conditions,
  int16_t                        nConditions
)
```

This function sets up trigger conditions on the scope's analog and digital inputs. The trigger is defined by one or more `PS2000A_TRIGGER_CONDITIONS` structures that are then ORed together. Each structure is itself the AND of the states of one or more of the inputs. This AND–OR logic allows you to create any possible Boolean function of the scope's inputs. (The 16 digital inputs of an MSO count as a unit for the purposes of this function.)

If complex triggering is not required, use `ps2000aSetSimpleTrigger()`.

Applicability	All modes

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`* conditions`, an array of `PS2000A_TRIGGER_CONDITIONS` structures specifying the conditions that should be applied to each channel. In the simplest case, the array consists of a single element. When there is more than one element, the overall trigger condition is the logical OR of all the elements.

`nConditions`, the number of elements in the `conditions` array. If `nConditions` is zero, triggering is switched off.

Returns	`PICO_OK`
	`PICO_INVALID_HANDLE`
	`PICO_USER_CALLBACK`
	`PICO_CONDITIONS`
	`PICO_MEMORY`
	`PICO_DRIVER_FUNCTION`

3.57.1 PS2000A_TRIGGER_CONDITIONS structure

A structure of this type is passed to ps2000aSetTriggerChannelConditions() in the conditions argument to specify the trigger conditions, and is defined as follows:

```
typedef struct tPS2000ATriggerConditions
{
  PS2000A_TRIGGER_STATE     channelA;
  PS2000A_TRIGGER_STATE     channelB;
  PS2000A_TRIGGER_STATE     channelC;
  PS2000A_TRIGGER_STATE     channelD;
  PS2000A_TRIGGER_STATE     external;
  PS2000A_TRIGGER_STATE     aux;
  PS2000A_TRIGGER_STATE     pulseWidthQualifier;
  PS2000A_TRIGGER_STATE     digital;
} PS2000A_TRIGGER_CONDITIONS
```

Each structure is the logical AND of the states of the scope's inputs. The ps2000aSetTriggerChannelConditions() function can OR together a number of these structures to produce the final trigger condition, which can be any possible Boolean function of the scope's inputs. (The 16 digital inputs of an MSO count as a unit for the purposes of this function.)

The structure is byte-aligned. In C++, for example, you should specify this using the `#pragma pack()` instruction.

Elements

`channelA, channelB, channelC, channelD, external, pulseWidthQualifier, digital`: the type of condition that should be applied to each channel. Use these constants:

- `PS2000A_CONDITION_DONT_CARE`
- `PS2000A_CONDITION_TRUE`
- `PS2000A_CONDITION_FALSE`

The channels that are set to `PS2000A_CONDITION_TRUE` or `PS2000A_CONDITION_FALSE` must all meet their conditions simultaneously to produce a trigger. Channels set to `PS2000A_CONDITION_DONT_CARE` are ignored.

`aux`: not used.

3.58 ps2000aSetTriggerChannelDirections() – set up signal polarities for triggering

```
PICO_STATUS ps2000aSetTriggerChannelDirections
(
  int16_t                        handle,
  PS2000A_THRESHOLD_DIRECTION    channelA,
  PS2000A_THRESHOLD_DIRECTION    channelB,
  PS2000A_THRESHOLD_DIRECTION    channelC,
  PS2000A_THRESHOLD_DIRECTION    channelD,
  PS2000A_THRESHOLD_DIRECTION    ext,
  PS2000A_THRESHOLD_DIRECTION    aux
)
```

This function sets the direction of the trigger for each channel.

Applicability	All modes

Arguments

handle, device identifier returned by ps2000aOpenUnit().

channelA, channelB, channelC, channelD, ext, the direction in which the signal must pass through the threshold to activate the trigger. See the table below for allowable values. If using a level trigger in conjunction with a pulse-width trigger, see the description of the direction argument to ps2000aSetPulseWidthQualifier() for more information.

aux: not used.

Returns	PICO_OK PICO_INVALID_HANDLE PICO_USER_CALLBACK PICO_INVALID_PARAMETER

PS2000A_THRESHOLD_DIRECTION **constants**

Constant	Trigger type	Direction
PS2000A_ABOVE	gated	above the upper threshold
PS2000A_ABOVE_LOWER	gated	above the lower threshold
PS2000A_BELOW	gated	below the upper threshold
PS2000A_BELOW_LOWER	gated	below the lower threshold
PS2000A_RISING	threshold	rising edge, using upper threshold
PS2000A_RISING_LOWER	threshold	rising edge, using lower threshold
PS2000A_FALLING	threshold	falling edge, using upper threshold
PS2000A_FALLING_LOWER	threshold	falling edge, using lower threshold
PS2000A_RISING_OR_FALLING	threshold	either edge
PS2000A_INSIDE	window-qualified	inside window
PS2000A_OUTSIDE	window-qualified	outside window
PS2000A_ENTER	window	entering the window
PS2000A_EXIT	window	leaving the window
PS2000A_ENTER_OR_EXIT	window	entering or leaving the window
PS2000A_NONE	none	none

3.59 ps2000aSetTriggerChannelProperties() – set up trigger thresholds

```
PICO_STATUS ps2000aSetTriggerChannelProperties
(
  int16_t                               handle,
  PS2000A_TRIGGER_CHANNEL_PROPERTIES  * channelProperties,
  int16_t                               nChannelProperties,
  int16_t                               auxOutputEnable,
  int32_t                               autoTriggerMilliseconds
)
```

This function is used to enable or disable triggering on the analog channels and set its parameters.

Applicability	All modes

Arguments

handle, device identifier returned by ps2000aOpenUnit().

channelProperties, a pointer to an array of PS2000A_TRIGGER_CHANNEL_PROPERTIES structures describing the requested properties. The array can contain a single element describing the properties of one channel or a number of elements describing several channels. If NULL is passed, triggering on analog channels is switched off.

nChannelProperties, the length of the channelProperties array. If zero, triggering on analog channels is switched off.

auxOutputEnable, not used.

autoTriggerMilliseconds, the number of milliseconds for which the scope device will wait for a trigger before timing out. If this argument is set to zero, the scope device will wait indefinitely for a trigger. In block mode, the capture cannot finish until a trigger event or auto-trigger timeout has occurred. In streaming mode the device always starts collecting data as soon as ps2000aRunStreaming() is called but does not start counting post-trigger samples until it detects a trigger event or auto-trigger timeout.

Returns	PICO_OK PICO_INVALID_HANDLE PICO_USER_CALLBACK PICO_TRIGGER_ERROR PICO_MEMORY PICO_INVALID_TRIGGER_PROPERTY PICO_DRIVER_FUNCTION PICO_INVALID_PARAMETER

3.59.1 PS2000A_TRIGGER_CHANNEL_PROPERTIES structure

A structure of this type is passed to `ps2000aSetTriggerChannelProperties()` in the `channelProperties` argument to specify the trigger mechanism, and is defined as follows:

```
typedef struct tPS2000ATriggerChannelProperties
{
  int16_t                      thresholdUpper;
  uint16_t                     thresholdUpperHysteresis;
  int16_t                      thresholdLower;
  uint16_t                     thresholdLowerHysteresis;
  PS2000A_CHANNEL              channel;
  PS2000A_THRESHOLD_MODE       thresholdMode;
} PS2000A_TRIGGER_CHANNEL_PROPERTIES
```

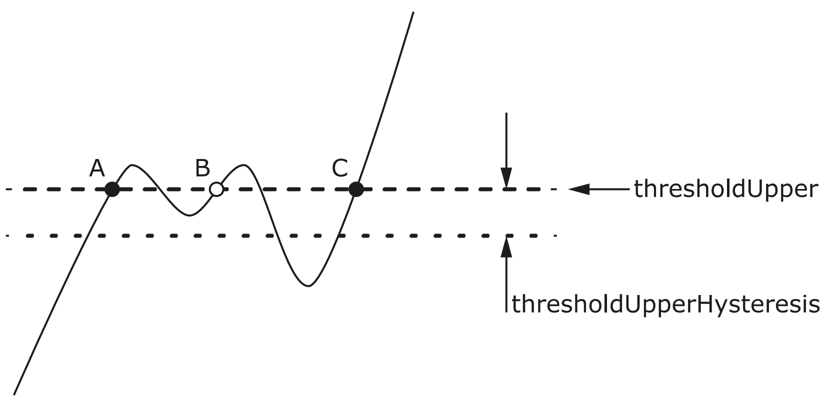
The structure is byte-aligned. In C++, for example, you should specify this using the `#pragma pack()` instruction.

Upper and lower thresholds
The digital triggering hardware in your PicoScope has two independent trigger thresholds called *upper* and *lower*. For some trigger types you can freely choose which threshold to use. The table in `ps2000aSetTriggerChannelDirections()` shows which thresholds are available for use with which trigger types. Dual thresholds are used for pulse-width triggering, when one threshold applies to the level trigger and the other to the pulse-width qualifier; and for window triggering, when the two thresholds define the upper and lower limits of the window.

Each threshold has its own trigger and hysteresis settings.

Hysteresis
Each trigger threshold (*upper* and *lower*) has an accompanying parameter called *hysteresis*. This defines a second threshold at a small offset from the main threshold. The trigger fires when the signal crosses the trigger threshold, but will not fire again until the signal has crossed the hysteresis threshold and then returned to cross the trigger threshold. The double-threshold mechanism prevents noise on the signal from causing unwanted trigger events.

For a rising-edge trigger the hysteresis threshold is below the trigger threshold. After one trigger event, the signal must fall below the hysteresis threshold before the trigger is enabled for the next event. Conversely, for a falling-edge trigger, the hysteresis threshold is always above the trigger threshold. After a trigger event, the signal must rise above the hysteresis threshold before the trigger is enabled for the next event.



Hysteresis – The trigger fires at **A** as the signal rises past the trigger threshold. It does not fire at **B** because the signal has not yet dipped below the hysteresis threshold. The trigger fires again at **C** after the signal has dipped below the hysteresis threshold and risen again past the trigger threshold.

Elements

`thresholdUpper`, the upper threshold at which the trigger fires. This is scaled in 16-bit ADC counts at the currently selected range for that channel.

`thresholdUpperHysteresis`, the distance between the upper trigger threshold and the upper hysteresis threshold, scaled in 16-bit counts.

`thresholdLower`, `thresholdLowerHysteresis`, the settings for the lower threshold: see `thresholdUpper` and `thresholdUpperHysteresis`.

`channel`, the channel to which the properties apply. This can be one of the four input channels listed under `ps2000aSetChannel()`, or `PS2000A_TRIGGER_EXT` for the **EXT** input fitted to some models.

`thresholdMode`, either a level or window trigger. Use one of these constants:

`PS2000A_LEVEL`
`PS2000A_WINDOW`

3.60 ps2000aSetTriggerDelay() – set up post-trigger delay

```
PICO_STATUS ps2000aSetTriggerDelay
(
  int16_t     handle,
  uint32_t    delay
)
```

This function sets the post-trigger delay, which causes capture to start a defined time after the trigger event.

Applicability	All modes (but `delay` is ignored in streaming mode)

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`delay`, the time between the trigger occurring and the first sample. For example, if `delay`=100 then the scope would wait 100 sample periods before sampling. At a timebase of 1 GS/s, or 1 ns per sample, the total delay would then be 100 x 1 ns = 100 ns.

Range: 0 to `MAX_DELAY_COUNT`.

Returns	`PICO_OK`
	`PICO_INVALID_HANDLE`
	`PICO_USER_CALLBACK`
	`PICO_DRIVER_FUNCTION`

3.61 ps2000aSetTriggerDigitalPortProperties() – set up digital channel trigger directions

```
PICO_STATUS ps2000aSetTriggerDigitalPortProperties
(
  int16_t                               handle,
  PS2000A_DIGITAL_CHANNEL_DIRECTIONS  * directions,
  int16_t                               nDirections
)
```

This function sets trigger directions for one or more digital channels.

Applicability	All modes

Arguments

handle, device identifier returned by ps2000aOpenUnit().

* directions, a pointer to an array of PS2000A_DIGITAL_CHANNEL_DIRECTIONS structures describing the requested properties. The array can contain a single element describing the properties of one channel, or a number of elements describing several digital channels. If directions is NULL, triggering on digital inputs is switched off. A digital channel that is not included in the array is set to PS2000A_DIGITAL_DONT_CARE.

nDirections, the number of digital channel directions being passed to the driver.

Returns	PICO_OK PICO_INVALID_HANDLE PICO_DRIVER_FUNCTION PICO_INVALID_DIGITAL_CHANNEL PICO_INVALID_DIGITAL_TRIGGER_DIRECTION

3.61.1 PS2000A_DIGITAL_CHANNEL_DIRECTIONS structure

A structure of this type is passed to `ps2000aSetTriggerDigitalPortProperties()` in the `directions` argument to specify the trigger mechanism, and is defined as follows:

```
pragma pack(1)
typedef struct tPS2000ADigitalChannelDirections
{
   PS2000A_DIGITAL_CHANNEL   channel;
   PS2000A_DIGITAL_DIRECTION direction;
} PS2000A_DIGITAL_CHANNEL_DIRECTIONS;
#pragma pack()

typedef enum enPS2000ADigitalChannel
{
   PS2000A_DIGITAL_CHANNEL_0,
   PS2000A_DIGITAL_CHANNEL_1,
   PS2000A_DIGITAL_CHANNEL_2,
   PS2000A_DIGITAL_CHANNEL_3,
   PS2000A_DIGITAL_CHANNEL_4,
   PS2000A_DIGITAL_CHANNEL_5,
   PS2000A_DIGITAL_CHANNEL_6,
   PS2000A_DIGITAL_CHANNEL_7,
   PS2000A_DIGITAL_CHANNEL_8,
   PS2000A_DIGITAL_CHANNEL_9,
   PS2000A_DIGITAL_CHANNEL_10,
   PS2000A_DIGITAL_CHANNEL_11,
   PS2000A_DIGITAL_CHANNEL_12,
   PS2000A_DIGITAL_CHANNEL_13,
   PS2000A_DIGITAL_CHANNEL_14,
   PS2000A_DIGITAL_CHANNEL_15,
   PS2000A_DIGITAL_CHANNEL_16,
   PS2000A_DIGITAL_CHANNEL_17,
   PS2000A_DIGITAL_CHANNEL_18,
   PS2000A_DIGITAL_CHANNEL_19,
   PS2000A_DIGITAL_CHANNEL_20,
   PS2000A_DIGITAL_CHANNEL_21,
   PS2000A_DIGITAL_CHANNEL_22,
   PS2000A_DIGITAL_CHANNEL_23,
   PS2000A_DIGITAL_CHANNEL_24,
   PS2000A_DIGITAL_CHANNEL_25,
   PS2000A_DIGITAL_CHANNEL_26,
   PS2000A_DIGITAL_CHANNEL_27,
   PS2000A_DIGITAL_CHANNEL_28,
   PS2000A_DIGITAL_CHANNEL_29,
   PS2000A_DIGITAL_CHANNEL_30,
   PS2000A_DIGITAL_CHANNEL_31,
   PS2000A_MAX_DIGITAL_CHANNELS
} PS2000A_DIGITAL_CHANNEL;
```



```
typedef enum enPS2000ADigitalDirection
{
   PS2000A_DIGITAL_DONT_CARE,
   PS2000A_DIGITAL_DIRECTION_LOW,
   PS2000A_DIGITAL_DIRECTION_HIGH,
   PS2000A_DIGITAL_DIRECTION_RISING,
   PS2000A_DIGITAL_DIRECTION_FALLING,
   PS2000A_DIGITAL_DIRECTION_RISING_OR_FALLING,
   PS2000A_DIGITAL_MAX_DIRECTION
} PS2000A_DIGITAL_DIRECTION;
```

The structure is byte-aligned. In C++, for example, you should specify this using the `#pragma pack()` instruction.

3.62 ps2000aSigGenArbitraryMinMaxValues() – query AWG parameter limits

```
PICO_STATUS ps2000aSigGenArbitraryMinMaxValues
(
  int16_t       handle,
  int16_t     * minArbitraryWaveformValue,
  int16_t     * maxArbitraryWaveformValue,
  uint32_t    * minArbitraryWaveformSize,
  uint32_t    * maxArbitraryWaveformSize
)
```

This function returns the range of possible sample values and waveform buffer sizes that can be supplied to ps2000aSetSigGenArbitrary() for setting up the arbitrary waveform generator (AWG). These values may vary between models.

Applicability	All models with AWG

Arguments

handle, device identifier returned by ps2000aOpenUnit().

minArbitraryWaveformValue, on exit, the lowest sample value allowed in the arbitraryWaveform buffer supplied to ps2000aSetSigGenArbitrary().

maxArbitraryWaveformValue, on exit, the highest sample value allowed in the arbitraryWaveform buffer supplied to ps2000aSetSigGenArbitrary().

minArbitraryWaveformSize, on exit, the minimum value allowed for the arbitraryWaveformSize argument supplied to ps2000aSetSigGenArbitrary().

maxArbitraryWaveformSize, on exit, the maximum value allowed for the arbitraryWaveformSize argument supplied to ps2000aSetSigGenArbitrary().

Returns	PICO_OK PICO_NOT_SUPPORTED_BY_THIS_DEVICE, if the device does not have an arbitrary waveform generator PICO_NULL_PARAMETER, if all the parameter pointers are NULL PICO_INVALID_HANDLE PICO_DRIVER_FUNCTION

3.63 ps2000aSigGenFrequencyToPhase() – calculate AWG phase from frequency

```
PICO_STATUS ps2000aSigGenFrequencyToPhase
(
  int16_t               handle,
  double                frequency,
  PS2000A_INDEX_MODE    indexMode,
  uint32_t              bufferLength,
  uint32_t            * phase
)
```

This function converts a frequency to a phase count for use with the arbitrary waveform generator setup functions `ps2000aSetSigGenArbitrary()` and `ps2000aSetSigGenPropertiesArbitrary()`. The value returned depends on the length of the buffer, the index mode passed and the device model.

Applicability	All models with AWG

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`frequency`, the required AWG output frequency.

`indexMode`, see `ps2000aSetSigGenArbitrary()`.

`bufferLength`, the number of samples in the AWG buffer.

`phase`, on exit, the `deltaPhase` argument to be sent to the AWG setup function.

Returns	`PICO_OK`
	`PICO_NOT_SUPPORTED_BY_THIS_DEVICE`, if the device does not have an AWG
	`PICO_SIGGEN_FREQUENCY_OUT_OF_RANGE`, if the frequency is out of range
	`PICO_NULL_PARAMETER`, if `phase` is a NULL pointer
	`PICO_SIG_GEN_PARAM`, if `indexMode` or `bufferLength` is out of range
	`PICO_INVALID_HANDLE`
	`PICO_DRIVER_FUNCTION`

3.64 ps2000aSigGenSoftwareControl() – trigger the signal generator

```
PICO_STATUS ps2000aSigGenSoftwareControl
(
  int16_t      handle,
  int16_t      state
)
```

This function causes a trigger event, or starts and stops gating, for the signal generator. Use it as follows:

1. Call `ps2000aSetSigGenBuiltIn()` or `ps2000aSetSigGenArbitrary()` to set up the signal generator, setting the `triggerSource` argument to `PS2000A_SIGGEN_SOFT_TRIG`.

2. (a) If you set the signal generator `triggerType` to edge triggering (`PS2000A_SIGGEN_RISING` or `PS2000A_SIGGEN_FALLING`), call `ps2000aSigGenSoftwareControl()` once to trigger the signal generator.
(b) If you set the signal generator `triggerType` to gated-low triggering (`PS2000A_SIGGEN_GATE_LOW`), call `ps2000aSigGenSoftwareControl()` with `state=0` to start the sweep and then again with `state=1` to stop it.
(c) If you set the signal generator `triggerType` to gated-high triggering (`PS2000A_SIGGEN_GATE_HIGH`), call `ps2000aSigGenSoftwareControl()` with `state=1` to start the sweep and then again with `state=0` to stop it.

Generating continuous output runs

- If `shots` is set to `PS2000A_SHOT_SWEEP_TRIGGER_CONTINUOUS_RUN` in `ps2000aSetSigGenBuiltIn()` or `ps2000aSetSigGenArbitrary()`, and `triggerType` to `PS2000A_SIGGEN_GATE_HIGH`, then `state=1` will cause the signal generator to output, while `state=0` will cause it to stop.

- If `shots` is set to `PS2000A_SHOT_SWEEP_TRIGGER_CONTINUOUS_RUN` in `ps2000aSetSigGenBuiltIn()` or `ps2000aSetSigGenArbitrary()` and `trigType` is set to `PS2000A_SIGGEN_GATE_LOW`, the signal generator starts to output immediately. Setting `state=1` will cause it to stop.

- Trying to set a specific number of shots and then attempting to use a gate will cause the call to `ps2000aSetSigGenBuiltIn()` or `ps2000aSetSigGenArbitrary()` to return an error.

Applicability	Use with `ps2000aSetSigGenArbitrary()` or `ps2000aSetSigGenBuiltIn()`.

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`state`, specifies whether to start or stop the sweep (see note 2 above). Effective only when the signal generator `triggerType` is set to `PS2000A_SIGGEN_GATE_HIGH` or `PS2000A_SIGGEN_GATE_LOW`. Ignored for other trigger types.

Returns	PICO_OK PICO_INVALID_HANDLE PICO_NO_SIGNAL_GENERATOR PICO_SIGGEN_TRIGGER_SOURCE PICO_DRIVER_FUNCTION PICO_NOT_RESPONDING

3.65 ps2000aStop() – stop data capture

```
PICO_STATUS ps2000aStop
(
  int16_t      handle
)
```

This function stops the scope device while it is waiting for a trigger or capturing data.

- In block mode, you can optionally call `ps2000aStop()` to terminate the current capture. Any data in the buffer will be invalid.
- In rapid block mode, you can optionally call `ps2000aStop()` to terminate the sequence of captures. Any completed captures will contain valid data but no further captures will be made.
- In streaming mode, calling `ps2000aStop()` is the usual way to terminate data capture. If this function is called before a trigger event occurs, the oscilloscope may not contain valid data. If capture has already started, the buffer will contain valid data.

Applicability	All modes

Arguments
`handle`, device identifier returned by `ps2000aOpenUnit()`.

Returns	`PICO_OK`
	`PICO_INVALID_HANDLE`
	`PICO_USER_CALLBACK`
	`PICO_DRIVER_FUNCTION`

3.66 ps2000aStreamingReady() – find out if streaming-mode data ready

```
typedef void (CALLBACK *ps2000aStreamingReady)
(
  int16_t        handle,
  int32_t        noOfSamples,
  uint32_t       startIndex,
  int16_t        overflow,
  uint32_t       triggerAt,
  int16_t        triggered,
  int16_t        autoStop,
  void         * pParameter
)
```

This callback function is part of your application. You register it with the driver using `ps2000aGetStreamingLatestValues()`, and the driver calls it back when streaming-mode data is ready. You can then download the data using the `ps2000aGetValuesAsync()` function.

The function should do nothing more than copy the data to another buffer within your application. To maintain the best application performance, the function should return as quickly as possible without attempting to process or display the data.

Applicability	Streaming mode only

Arguments

`handle`, device identifier returned by `ps2000aOpenUnit()`.

`noOfSamples`, the number of samples to collect.

`startIndex`, an index to the first valid sample in the buffer. This is the buffer that was previously passed to `ps2000aSetDataBuffer()`.

`overflow`, returns a set of flags that indicate whether an overvoltage has occurred on any of the channels. It is a bit pattern with bit 0 corresponding to Channel A.

`triggerAt`, an index to the buffer indicating the location of the trigger point relative to `startIndex`. The trigger point is therefore at `StartIndex + triggerAt`. This parameter is valid only when `triggered` is non-zero.

`triggered`, a flag indicating whether a trigger occurred. If non-zero, a trigger occurred at the location indicated by `triggerAt`.

`autoStop`, the flag that was set in the call to `ps2000aRunStreaming()`.

`* pParameter`, a void pointer passed from `ps2000aGetStreamingLatestValues()`. The callback function can write to this location to send any data, such as a status flag, back to the application.

Returns	nothing

3.67 Wrapper functions

The Software Development Kits (SDKs) for PicoScope devices contain wrapper dynamic link library (DLL) files in the `lib` subdirectory of your SDK installation for 32-bit and 64-bit systems. The wrapper functions provided by the wrapper DLLs are for use with programming languages such as MathWorks MATLAB, National Instruments LabVIEW and Microsoft Excel VBA that do not support features of the C programming language such as callback functions.

The source code contained in the `Wrapper` projects contains a description of the functions and the input and output parameters.

Below we explain the sequence of calls required to capture data in streaming mode using the wrapper API functions.

The `ps2000aWrap.dll` wrapper DLL has a callback function for streaming data collection that copies data from the driver buffer specified to a temporary application buffer of the same size. To do this it must be registered with the wrapper and the channel must be specified as being enabled. You should process the data in the temporary application buffer accordingly, for example by copying the data into a large array.

Procedure:

1. Open the oscilloscope using `ps2000aOpenUnit()`.

1a. Inform the wrapper of the number of channels on the device by calling `setChannelCount`.

2. Select channels, ranges and AC/DC coupling using `ps2000aSetChannel()`.

2a. Inform the wrapper which channels have been enabled by calling `setEnabledChannels`.

3. [MSOs only] Set the digital port using `ps2000aSetDigitalPort()`.

3a. [MSOs only] Inform the wrapper which digital ports have been enabled by calling `setEnabledDigitalPorts`.

4. Use the appropriate trigger setup functions. For programming languages that do not support structures, use the wrapper's advanced trigger setup functions.

5. [MSOs only] Use the trigger setup function `ps2000aSetTriggerDigitalPortProperties()` to set up the digital trigger if required.

6. Call `ps2000aSetDataBuffer()` (or for aggregated data collection `ps2000aSetDataBuffers()`) to tell the driver where your data buffer(s) is(are).

6a. Register the data buffer(s) with the wrapper and set the application buffer(s) into which the data will be copied.

    For analog channels: Call `setAppAndDriverBuffers` (or `setMaxMinAppAndDriverBuffers` for aggregated data collection).

    [MSOs Only] For digital ports: Call `setAppAndDriverDigiBuffers` (or `setMaxMinAppAndDriverDigiBuffers` for aggregated data collection).

7. Start the oscilloscope running using `ps2000aRunStreaming()`.

8. Loop and call `GetStreamingLatestValues` and `IsReady` to get data and flag when the wrapper is ready for data to be retrieved.

8a. Call the wrapper's `AvailableData` function to obtain information on the number of samples collected and the start index in the buffer.

8b. Call the wrapper's `IsTriggerReady` function for information on whether a trigger has occurred and the trigger index relative to the start index in the buffer.

9. Process data returned to your application data buffers.

10. Call `AutoStopped` if the `autoStop` parameter has been set to `TRUE` in the call to `ps2000aRunStreaming()`.

11. Repeat steps 8 to 10 until `AutoStopped` returns true or you wish to stop data collection.

12. Call `ps2000aStop`, even if the `autoStop` parameter was set to `TRUE`.

13. To disconnect a device, call `ps2000aCloseUnit()`.

4 Further information

4.1 Driver status codes

Every function in the ps2000a driver returns a **driver status code** from the list of `PICO_STATUS` values in `PicoStatus.h`, which is included in the `inc` folder of the PicoSDK installation.

4.2 Enumerated types and constants

Enumerated types and constants are defined in `ps2000aApi.h`, which is included in the SDK under the `inc` folder. We recommend that you refer to these constants by name unless your programming language allows only numerical values.

4.3 Numeric data types

Here is a list of the numeric data types used in the PicoScope 2000 Series A API:

Type	Bits	Signed or unsigned?
`int8_t`	8	signed
`int16_t`	16	signed
`uint16_t`	16	unsigned
`enum`	32	enumerated
`int32_t`	32	signed
`uint32_t`	32	unsigned
`float`	32	signed (IEEE 754 binary32)
`double`	64	signed (IEEE 754 binary64)
`int64_t`	64	signed
`uint64_t`	64	unsigned

5 Glossary

AC/DC control. Each channel can be set to either AC coupling or DC coupling. With DC coupling, the voltage displayed on the screen is equal to the true voltage of the signal. With AC coupling, any DC component of the signal is filtered out, leaving only the variations in the signal (the AC component).

Aggregation. This is the data-reduction method used by the PicoScope 2000 Series (A API) scopes. For each block of consecutive samples, the scope transmits only the minimum and maximum samples over the USB port to the PC. In streaming mode you can set the number of samples in each block, called the downsampling ratio, when you call `ps2000aRunStreaming()` for real-time capture, and when you call `ps2000aGetStreamingLatestValues()` to obtain post-processed data. In block mode you can specify the downsampling ratio when calling `ps2000aGetValues()`. In rapid block mode you can specify the ratio when calling `ps2000aGetValuesBulk()`.

Block mode. A sampling mode in which the computer prompts the oscilloscope to collect a block of data into its internal memory before stopping the oscilloscope and transferring the whole block into computer memory. This mode of operation is effective when the input signal being sampled contains high frequencies. Note: To avoid aliasing effects, the maximum input frequency must be less than half the sampling rate.

Buffer size. The size, in samples, of the oscilloscope buffer memory. The buffer memory is used by the oscilloscope to temporarily store data before transferring it to the PC.

Callback. A call made by the ps2000a driver to a function in your application. It indicates that data is ready for collection.

ETS. Equivalent Time Sampling. ETS constructs a picture of a repetitive signal by accumulating information over many similar wave cycles. This means the oscilloscope can capture fast-repeating signals that have a higher frequency than the maximum sampling rate. Note: ETS cannot be used for one-shot or non-repetitive signals.

External trigger. This is the BNC socket marked **EXT** on some PicoScope oscilloscopes. A pulse fed into this input can be used to start data capture.

Maximum sampling rate. A figure indicating the maximum number of samples the oscilloscope is capable of acquiring per second. Maximum sample rates are given in MS/s (megasamples per second) or GS/s (gigasamples per second). The higher the sampling capability of the oscilloscope, the more accurate the representation of the high frequencies in a fast signal.

MSO (mixed-signal oscilloscope). An oscilloscope that has both analog and digital inputs.

Signal generator. This is a feature of some oscilloscopes that can generate a signal for test purposes. The signal generator output is the BNC socket marked **AWG** or **GEN** on the oscilloscope. If you connect a BNC cable between this and one of the channel inputs, you can send a signal into one of the channels. It can generate a sine, square, triangle or arbitrary wave of fixed or swept frequency.

Streaming mode. A sampling mode in which the oscilloscope samples data and returns it to the computer in an unbroken stream. This mode of operation is effective when the input signal being sampled contains only low frequencies.

Timebase. A function within the PicoScope device that controls the time between samples. This time is programmable.

USB 1.1. An early version of the Universal Serial Bus standard found on older PCs. Although your PicoScope will work with a USB 1.1 port, it will operate much more slowly than with a USB 2.0 or 3.0 port.

USB 2.0. Universal Serial Bus (High Speed). A standard port used to connect external devices to PCs. The high-speed data connection provided by a USB 2.0 port enables your PicoScope to achieve its maximum performance.

USB 3.0. A faster version of the Universal Serial Bus standard. Your PicoScope is fully compatible with USB 3.0 ports and will operate with the same performance as on a USB 2.0 port.

Vertical resolution. A value, in bits, indicating the degree of precision with which the oscilloscope can turn input voltages into digital values.

Voltage range. The voltage range is the difference between the maximum and minimum voltages that can be accurately captured by the oscilloscope.

Index

S

T

U

V

W

United Kingdom global headquarters:
Pico Technology
James House
Colmworth Business Park
St. Neots
Cambridgeshire
PE19 8YP
United Kingdom

Tel: +44 (0) 1480 396 395

sales@picotech.com
support@picotech.com

www.picotech.com

United States regional office:
Pico Technology
320 N Glenwood Blvd
Tyler
TX 75702
United States

Tel: +1 800 591 2796

sales@picotech.com
support@picotech.com

Asia-Pacific regional office:
Pico Technology
Room 2252, 22/F, Centro
568 Hengfeng Road
Zhabei District
Shanghai 200070
PR China

Tel: +86 21 2226-5152

pico.asia-pacific@picotech.com

ps2000apg.en-12