# PicoScope® 3000 Series

## PC Oscilloscopes and MSOs

## Programmer's Guide

ps3000apg.en-15


pico
Technology

# Contents



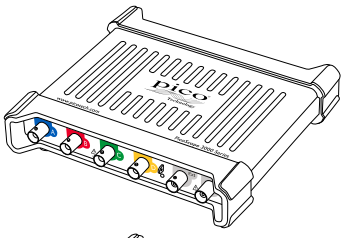
# 1 Introduction

## 1.1 Overview

The PicoScope 3000A, 3000B and 3000D Series Oscilloscopes and MSOs from Pico Technology are a range of high-specification, real-time measuring instruments that connect to the USB port of your computer. The series covers various options of portability, deep memory, fast sampling rates and high bandwidth, making it a highly versatile range that suits a wide range of applications. The range includes Hi-Speed USB 2.0 and SuperSpeed USB 3.0 devices.

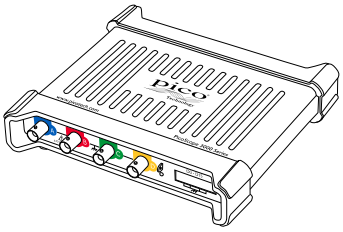
This manual explains how to use the *ps3000a* API (application programming interface) functions to develop your own programs to collect and analyze data from these oscilloscopes.

The information in this manual applies to the following oscilloscopes:



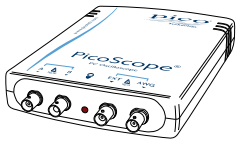
**PicoScope 3203D to 3206D**
**PicoScope 3403D to 3406D**
*USB 3.0 2-channel and 4-channel oscilloscopes*

3000D models have an arbitrary waveform generator.



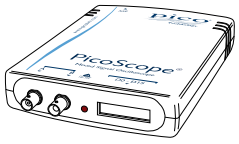
**PicoScope 3203D MSO to 3206D MSO**
**PicoScope 3403D MSO to 3406D MSO**
*USB 3.0 mixed-signal oscilloscopes*

3000D MSO models have 2 or 4 analog inputs, 16 digital inputs and an arbitrary waveform generator.



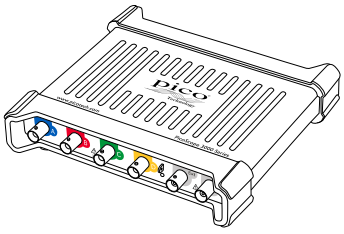
**PicoScope 3204A/B to 3207A/B**
*High-speed 2-channel oscilloscopes (discontinued)*

3000A Series models have a function generator; 3000B Series models have an arbitrary waveform generator.



**PicoScope 3204 MSO to 3206 MSO**
*USB 2.0 mixed-signal oscilloscopes (discontinued)*

3000 MSO models have 2 or 4 analog inputs, 16 digital inputs and an arbitrary waveform generator.



**PicoScope 3404A/B to 3406A/B**
*High-speed 4-channel oscilloscopes (discontinued)*

3000A Series models have a function generator; 3000B Series models have an arbitrary waveform generator.

For information on any of the above oscilloscopes, refer to the data sheets on our website.

For programming information on PicoScope 3000 Series oscilloscopes and MSOs not listed above, refer to the *PicoScope 3000 Series Programmer's Guide* available from www.picotech.com.

## 1.2 License agreement

**Grant of license.** The material contained in this release is licensed, not sold. Pico Technology Limited ('Pico') grants a license 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 and agree to abide by these conditions.

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

**Copyright.** The software in this release is for use only with Pico products or with data collected using Pico products. You may copy and distribute the SDK without restriction, as long as you do not remove any Pico Technology copyright statements. The example programs in the SDK may be modified, copied and distributed for the purpose of developing programs to collect data using Pico products.

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

**Fitness for purpose.** No two applications are the same, so Pico cannot guarantee that its equipment or software is suitable for a given application. It is therefore the user's responsibility to ensure that the product is suitable for the user's application.

**Mission-critical applications.** Because the software runs on a computer that may be running other software products, and may be subject to interference from these other products, this license specifically excludes usage in 'mission-critical' applications, for example life-support systems.

**Viruses.** This software was continuously monitored for viruses during production. However, the user is responsible for virus checking the software once it is installed.

**Support.** No software is ever error-free, but if you are dissatisfied with the performance of this software, please contact our technical support staff.

**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.

## 1.3 Trademarks

**Pico Technology** and **PicoScope** are trademarks of Pico Technology Limited, registered in the United Kingdom and other countries.

**PicoScope** and **Pico Technology** are registered in the U.S. Patent and Trademark Office.

**Windows** and **Visual Basic for Applications** are registered trademarks or trademarks of Microsoft Corporation in the USA and other countries.

# 2 Programming the PicoScope 3000 Series (A API) oscilloscopes

The `ps3000a.dll` dynamic link library (DLL) in the SDK allows you to program any supported oscilloscope using standard C function calls.

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

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

Numerous example programs are included in the SDK. These demonstrate how to use the functions of the driver software in each of the modes available.

## 2.1 Drivers

Your application communicates with two drivers—`ps3000a.dll` and `picoipp.dll`—which are supplied in 32-bit and 64-bit versions. `ps3000a.dll` exports the *ps3000a* 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 two DLLs depend on a low-level (kernel) driver called `WinUsb.sys`. This is installed by the SDK and configured when you plug the oscilloscope into each USB port for the first time.

## 2.2 Minimum PC requirements

To ensure that your PicoScope operates correctly, 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 multicore processor.

| Item | Specification |
|---|---|
| **Operating system** | Windows 7, 8 or 10 (32-bit or 64-bit)<br>Or Linux<br>Or OS X (Mac) |
| **Processor** | As required by operating system |
| **Memory** | |
| **Free disk space** | |
| **Ports** | USB 2.0 port |

## 2.3 USB port requirements

The *ps3000a* driver offers four different methods of recording data, all of which support both USB 1.1, USB 2.0, and USB 3.0 connections. The USB 2.0 oscilloscopes are Hi-Speed devices, so transfer rate will not increase by using USB 3.0, but it will decrease when using USB 1.1. The USB 3.0 oscilloscopes are SuperSpeed devices, so should be used with a USB 3.0 port for optimal performance.

# 3 Device features

## 3.1 Power options

PicoScope 3000 Series oscilloscopes can be powered in several ways depending on the model:

| | USB 2.0 cable | USB 2.0 double-headed cable | USB 3.0 cable | USB 2.0 cable + power supply |
|---|---|---|---|---|
| **PicoScope 3200A & 3200B** 2-channel USB 2.0 oscilloscopes | ✓ | | | |
| **PicoScope 3400A & 3400B** 4-channel USB 2.0 oscilloscopes | | ✓ | | ✓ |
| **PicoScope 3207A & 3207B** 2-channel USB 3.0 oscilloscopes | | ✓ | ✓ | |
| **PicoScope 3200D MSO** 2-channel USB 3.0 MSOs | | | | |
| **PicoScope 3200D** 2-channel USB 3.0 oscilloscopes | | | | |
| **PicoScope 3400D MSO** 4-channel USB 3.0 MSOs | | ✓ | ✓ | ✓ |
| **PicoScope 3400D** 4-channel USB 3.0 oscilloscopes | | | | |

**Data retention**

If the power source is changed (power supply connected or disconnected) while the oscilloscope is in operation, any unsaved data is lost. The application must then reconfigure the oscilloscope before data capture can continue.

**API functions**

The following functions are used to control the flexible power feature:

- ps3000aChangePowerSource
- ps3000aCurrentPowerSource

If you want the device to run on USB power only, instruct the driver by calling ps3000aChangePowerSource after calling ps3000aOpenUnit. If you call ps3000aOpenUnit without the power supply connected, the driver returns either PICO_POWER_SUPPLY_NOT_CONNECTED (for 4-channel scopes) or PICO_USB3_0_DEVICE_NON_USB3_0_PORT (for 2-channel USB 3.0 scopes plugged into a non-USB 3.0 port).

If the supply is connected or disconnected during use, the driver returns the relevant status code and you must then call ps3000aChangePowerSource before you can continue running the scope.

# 3.2 Voltage ranges

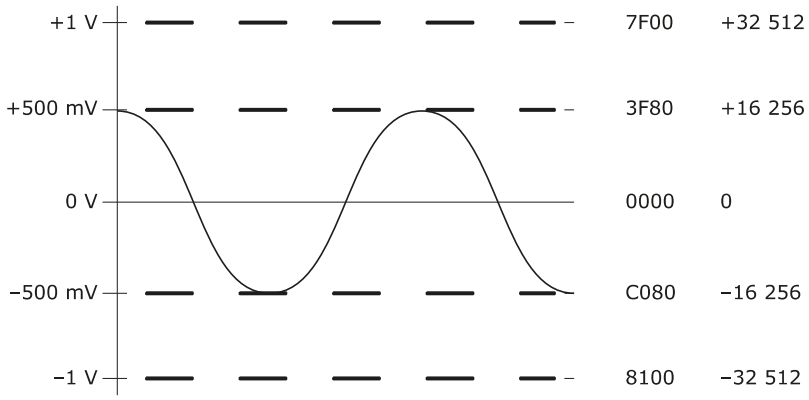
### Analog input channels

You can set a device input channel to any voltage range from ±50 mV to ±20 V with ps3000aSetChannel. Each sample is scaled to 16 bits so that the values returned to your application are as follows:

| Function | Voltage | Value returned | |
|---|---|---|---|
| | | decimal | hex |
| ps3000aMaximumValue | maximum | 32 512 | 7F00 |
| | 0 V | 0 | 0000 |
| ps3000aMinimumValue | minimum | −32 512 | 8100 |

### Example

1. Call ps3000aSetChannel with range set to PS3000A_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

The PicoScope 3000 Series D models have an external trigger input (marked **Ext**). This external trigger input is scaled to a 16-bit value as follows:

| Constant | Voltage | Value returned | |
|---|---|---|---|
| | | decimal | hex |
| PS3000A_EXT_MAX_VALUE | +5 V | +32 767 | 7FFF |
| | 0 V | 0 | 0000 |
| PS3000A_EXT_MIN_VALUE | - 5 V | −32 767 | 8001 |

# 3.3 MSO digital data

**Applicability: mixed-signal oscilloscope (MSO) devices only**

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

The data from each port is returned in a separate buffer that is set up by the ps3000aSetDataBuffer and ps3000aSetDataBuffers 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 while the upper 8 bits of the word are undefined.

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

**Retrieving stored digital data**

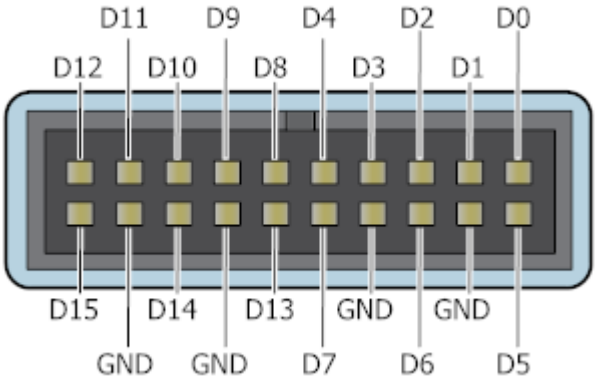
The following C code snippet shows how to combine data from the two 8-bit ports into a single 16-bit word, and then how to extract individual bits from the 16-bit word.

```
// 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,
// then OR with shifted Port 1 bits to get 16-bit word
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;
}
```

## 3.4 MSO digital connector

The PicoScope 3000 Series and 3000D Series MSOs have a digital input connector. The following pinout of the 20-pin IDC header plug is drawn as you look at the front panel of the device.




## 3.5 Triggering

PicoScope oscilloscopes can either start collecting data immediately, or be programmed to wait for a **trigger** event to occur. In both cases you need to use the trigger function ps3000aSetSimpleTrigger, which in turn calls:

- ps3000aSetTriggerChannelConditions
- ps3000aSetTriggerChannelDirections
- ps3000aSetTriggerChannelProperties

These can also be called individually, rather than using ps3000aSetSimpleTrigger, in order to set up advanced trigger types such as pulse width.

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 up to four inputs using the logic trigger function.

The driver supports these triggering methods:

- Simple edge
- Advanced edge
- Windowing
- Pulse width
- Logic
- Delay
- Drop-out
- Runt

The pulse width, delay and drop-out triggering methods additionally require the use of the pulse width qualifier function, ps3000aSetPulseWidthQualifier.

# 3.6 Timebases

The API allows you to select one of $2^{32}$ different timebases*. 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. ps3000aGetTimebase will tell you the sampling interval for a given timebase number.

**PicoScope 3000A and 3000B Series 2-Channel USB 2.0 Oscilloscopes**

| Timebase (n) | Sample interval formula | Sample interval | Notes |
|---|---|---|---|
| 0 | $2^n$ / 500,000,000 | 2 ns | Only one channel enabled |
| 1 | | 4 ns | |
| 2 | | 8 ns | |
| 3<br>...<br>$2^{32}-1$ | (n–2) / 62,500,000 | 16 ns<br>...<br>~ 68.7 s | |

**PicoScope 3000 Series USB 2.0 MSOs**

| Timebase (n) | Sample interval formula | Sample interval | Notes |
|---|---|---|---|
| 0 | $2^n$ / 500,000,000 | 2 ns | No more than one analog channel and one digital port enabled |
| 1 | | 4 ns | |
| 2<br>...<br>$2^{32}-1$ | (n–1) / 125,000,000 | 8 ns<br>...<br>~ 34.4 s | |

**PicoScope 3000A and 3000B Series 4-Channel USB 2.0 Oscilloscopes**
**PicoScope 3207A and 3207B USB 3.0 Oscilloscopes**
**PicoScope 3000D Series USB 3.0 Oscilloscopes and MSOs**

| Timebase (n) | Sample interval formula | Sample interval | Notes |
|---|---|---|---|
| 0 | $2^n$ / 1,000,000,000 | 1 ns | Only one analog channel enabled |
| 1 | | 2 ns | No more than two analog channels or digital ports enabled |
| 2 | | 4 ns | No more than four analog channels or digital ports enabled |
| 3<br>...<br>$2^{32}-1$ | (n–2) / 125,000,000 | 8 ns<br>...<br>~ 34.4 s | |

* The fastest timebase available depends on the number of channels and digital ports enabled, as specified in the data sheet. In streaming mode it also depends on the oscilloscope model.

# 3.7 Sampling modes

PicoScope oscilloscopes can run in various **sampling modes:**

- **Block mode.** In this mode, the scope stores data in its 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 capture is started, 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 buffer memory. This enables long periods of slow data collection for chart recorder and data-logging applications. Streaming mode supports downsampling and triggering, while providing fast streaming at up these rates:

| Number of active channels or ports* | Max. sampling rate (min. sample time) | |
|---|---|---|
| | USB 2.0 | USB 3.0 |
| 1 | 31.25 MS/s (32 ns) | 125 MS/s (8 ns) |
| 2 | 15.625 MS/s (64 ns) | 62.5 MS/s (16 ns) |
| 3 or 4 | 7.8125 MS/s (128 ns) | 31.25 MS/s (32 ns) |
| More than 4 | | 15.625 MS/s (64 ns) |

*Note: A port is a block of 8 digital channels, available on MSOs only.

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.

In programming environments not supporting callbacks, you may poll the driver in block mode or use one of the wrapper functions provided.

### 3.7.1 Block mode

In **block mode**, the computer prompts the 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. If three or four channels are enabled, each receives a quarter of the memory. These calculations are handled transparently by the driver. The block size also depends on the number of memory segments in use (see `ps3000aMemorySegments`).

  For the PicoScope 3000 and 3000D Series MSOs, the memory is shared between the digital ports and analog channels. If one or more analog channels is enabled at the same time as one or more digital ports, the memory per channel is one quarter of the buffer size.

- **Sampling rate.** A *ps3000a* oscilloscope can sample at a number of different rates according to the selected timebase and the combination of channels that are enabled. See the *PicoScope 3000 Series Data Sheet* 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 `ps3000aRunBlock`, `ps3000aStop` and `ps3000aGetValues`.

- **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 `ps3000aMemorySegments`.

- **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 or the power source is changed (for flexible power devices).

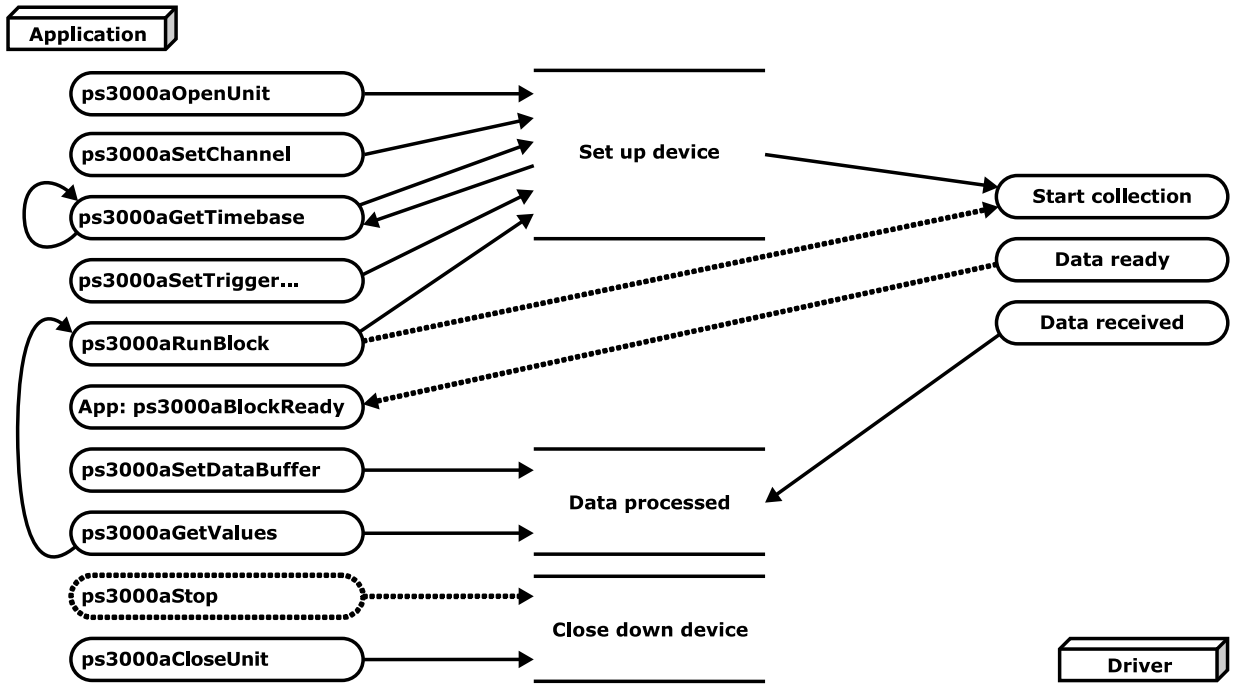
See Using block mode for programming details.

### 3.7.1.1 Using block mode

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

1. Open the oscilloscope using ps3000aOpenUnit.
2. Select channel ranges and AC/DC coupling using ps3000aSetChannel. All channels are enabled by default, so if you wish to allocate the buffer memory to fewer channels you must disable those that are not required.
3. *[MSOs only]* Set the digital port using ps3000aSetDigitalPort.
4. Using ps3000aGetTimebase, select timebases until the required number of nanoseconds per sample is located.
5. Use the trigger setup functions ps3000aSetTriggerChannelConditionsV2, ps3000aSetTriggerChannelDirections and ps3000aSetTriggerChannelProperties to set up the trigger if required.
6. *[MSOs only]* Use the trigger setup functions ps3000aSetTriggerDigitalPortProperties to set up the digital trigger if required.
7. Start the oscilloscope running using ps3000aRunBlock.
8. Wait until the oscilloscope is ready using the ps3000aBlockReady callback (or poll using ps3000aIsReady).
9. Use ps3000aSetDataBuffer 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 6.
10. Transfer the block of data from the oscilloscope using ps3000aGetValues.
11. Display the data.
12. Repeat steps 7 to 11.
13. Stop the oscilloscope using ps3000aStop.
14. Request new views of stored data using different downsampling parameters: see Retrieving stored data.
15. Close the oscilloscope using ps3000aCloseUnit.



### 3.7.1.2 Asynchronous calls in block mode

ps3000aGetValues may take a long time to complete if a large amount of data is being collected. For example, it can take several seconds to retrieve the full 512 M samples from a PicoScope 3206D using a USB 3.0 connection, or several minutes on USB 1.1. To avoid hanging the calling thread, it is possible to call ps3000aGetValuesAsync instead. This immediately returns control to the calling thread, which then has the option of waiting for the data or calling ps3000aStop to abort the operation.

## 3.7.2 Rapid block mode

In normal block mode, the oscilloscope collects 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 at a time with the minimum time between waveforms. It reduces the gap from milliseconds to less than 2 microseconds (on fastest timebase).

See Using rapid block mode for details.

### 3.7.2.1 Using rapid block mode

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

**Without aggregation**

1. Open the oscilloscope using `ps3000aOpenUnit`.
2. Select channel ranges and AC/DC coupling using `ps3000aSetChannel`.
3. *[MSOs only]* Set the digital port using `ps3000aSetDigitalPort`.
4. Set the number of memory segments equal to or greater than the number of captures required using `ps3000aMemorySegments`. Use `ps3000aSetNoOfCaptures` before each run to specify the number of waveforms to capture.
5. Using `ps3000aGetTimebase`, select timebases until the required sampling interval is located. The function will indicate the number of samples per channel available for each segment. If you do not need to know the segment size limit (because you are capturing a small number of samples) you can optionally call this function before step 4.
6. Use the trigger setup functions `ps3000aSetTriggerChannelConditionsV2`, `ps3000aSetTriggerChannelDirections` and `ps3000aSetTriggerChannelProperties` to set up the trigger if required.
7. *[MSOs only]* Use the trigger setup functions `ps3000aSetTriggerDigitalPortProperties` to set up the digital trigger if required.
8. Start the oscilloscope running using `ps3000aRunBlock`.
9. Wait until the oscilloscope is ready using the `ps3000aIsReady` or wait on the callback function.
10. Use `ps3000aSetDataBuffer` 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 when doing multiple captures, you can call this function outside the loop, after step 7.
11. Transfer the blocks of data from the oscilloscope using `ps3000aGetValuesBulk`.
12. Retrieve the time offset for each data segment using `ps3000aGetValuesTriggerTimeOffsetBulk64`.
13. Display the data.
14. Repeat steps 8 to 13 if necessary.
15. Stop the oscilloscope using `ps3000aStop`.
16. Close the oscilloscope using `ps3000aCloseUnit`.

**With aggregation**

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

10a. Call ps3000aSetDataBuffer or (ps3000aSetDataBuffers) to set up one pair of buffers for every waveform segment required.
11a. Call ps3000aGetValuesBulk for each pair of buffers.
12a. Retrieve the time offset for each data segment using ps3000aGetValuesTriggerTimeOffsetBulk64.

Continue from step 13 above.

### 3.7.2.2 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 number of captures required)

```
// Set the number of waveforms to 100
ps3000aSetNoOfCaptures(handle, 100);

pParameter = false;

ps3000aRunBlock
(
  handle,
  0,                  // noOfPreTriggerSamples
  10000,              // noOfPostTriggerSamples
  1,                  // timebase to be used
  1,                  // not used
  &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 (int32_t i = 0; i < 10; i++)
{
   for (int32_t c = PS3000A_CHANNEL_A; c <= PS3000A_CHANNEL_B; c++)
   {
      ps3000aSetDataBuffer
      (
         handle,
         c,
         buffer[c][i],
         MAX_SAMPLES,
         i
         PS3000A_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.

```
ps3000aGetValuesBulk
(
  handle,
  &noOfSamples,            // set to MAX_SAMPLES on entry
  0,                       // fromSegmentIndex
  9,                       // toSegmentIndex
  1,                       // downsampling ratio
  PS3000A_RATIO_MODE_NONE, // downsampling ratio mode
  overflow                 // an array of size 10 int16_t
)
```

Comments: the number of samples could be up to `noOfPreTriggerSamples + noOfPostTriggerSamples`, the values set in `ps3000aRunBlock`. The samples are always returned from the first sample taken, unlike the `ps3000aGetValues` function which allows the sample index to be set. The above segments start at 0 and finish at 9 inclusive. It is possible for the segment index to wrap around from the last segment to the first segment and end at `toSegmentIndex`, if for example `fromSegmentIndex` is 98 and `toSegmentIndex` is 7.

```
ps3000aGetValuesTriggerTimeOffsetBulk64
(
  handle,
  times,
  timeUnits,
  0,
  9
)
```

Comments: the above segments start at 0 and finish at 9 inclusive. As mentioned in the previous comment, it is possible for the segment index to wrap around from the last segment to the first segment and continue until `toSegmentIndex`.

### 3.7.2.3 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 100
ps3000aSetNoOfCaptures(handle, 100);

pParameter = false;
ps3000aRunBlock
(
  handle,
  0,                  // noOfPreTriggerSamples,
  1000000,            // noOfPostTriggerSamples,
  1,                  // timebase to be used,
  1,                  // not used
  &timeIndisposedMs,
  0,                  // segment index
  lpReady,
  &pParameter
);
```

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

```
for (int32_t segment = 10; segment < 20; segment++)
{
  for (int32_t c = PS3000A_CHANNEL_A; c <= PS3000A_CHANNEL_D; c++)
  {
    ps3000aSetDataBuffers
    (
      handle,
      c,
      bufferMax[c],
      bufferMin[c],
      MAX_SAMPLES,
      segment,
      PS3000A_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 samples.

```
    ps3000aGetValues
    (
      handle,
      0,
      &noOfSamples,        // set to MAX_SAMPLES on entry
      1000,
      downSampleRatioMode, // set to RATIO_MODE_AGGREGATE
      index,
      overflow
    );

    ps3000aGetTriggerTimeOffset64
    (
      handle,
      &time,
      &timeUnits,
      index
    )
  }
```

Comments: each waveform is retrieved one at a time from the driver with an aggregation of 1000.

## 3.7.3 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 trigger functions and ps3000aSetEts.

**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 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.

**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 ps3000aBlockReady callback function when a new waveform is ready for collection. Your application should then call ps3000aGetValues to retrieve the waveform from the data buffer and the sample times from the ETS times buffer.

| | |
|---|---|
| **Applicability** | Available in block mode only.<br>Not suitable for one-shot (non-repetitive) signals.<br>Aggregation is not supported.<br>Edge-triggering only.<br>Auto trigger delay (`autoTriggerMilliseconds`) is ignored.<br>Digital ports (on MSOs) cannot be used in ETS mode.<br>Refer to product specifications for availability of ETS triggering on specific devices. |

### 3.7.3.1 Using ETS mode

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

When using ETS mode you must consider if a digital port has previously been active. If it has, call ps3000aSetDigitalPort and ps3000aSetTriggerDigitalPortProperties to ensure these are not active when using ETS.

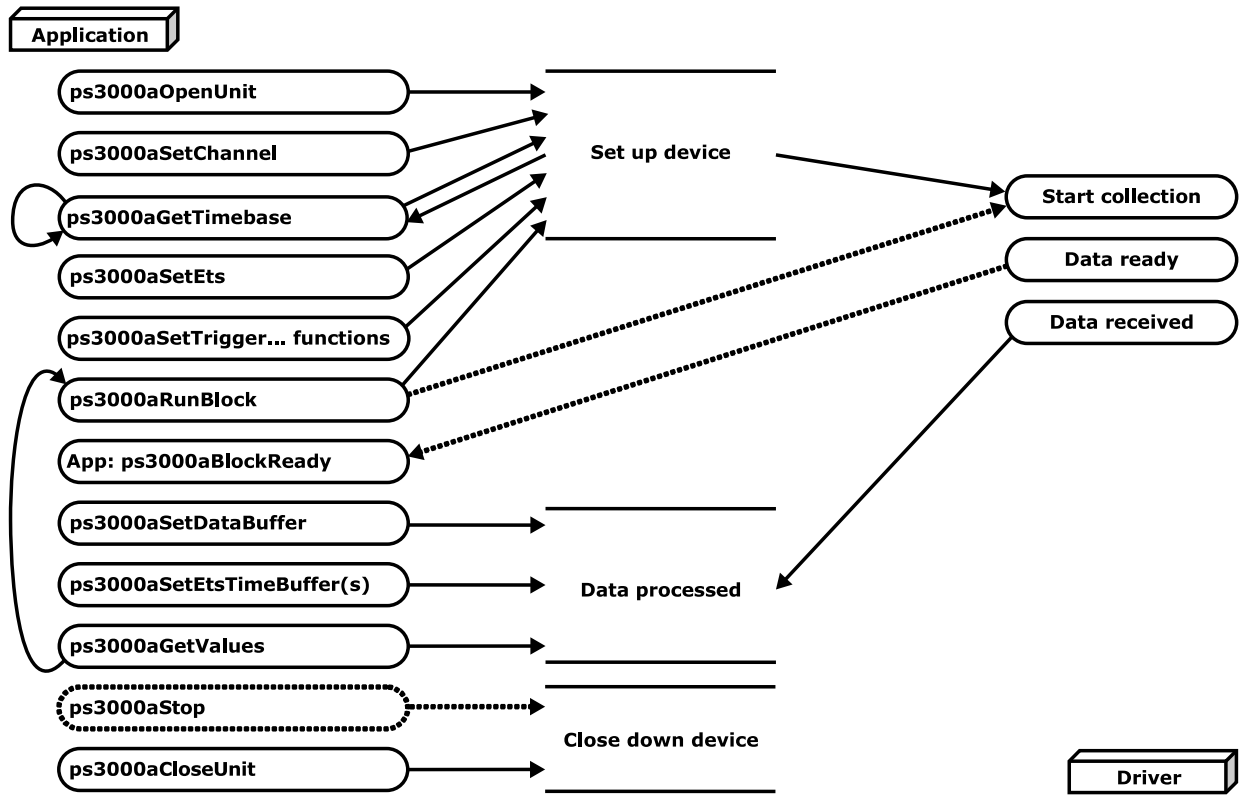
1. Open the oscilloscope using ps3000aOpenUnit.
2. Select channel ranges and AC/DC coupling using ps3000aSetChannel.
3. Use ps3000aSetEts to enable ETS and set the parameters.
4. Use ps3000aGetTimebase to verify the number of samples to be collected.
5. Use the trigger setup functions ps3000aSetTriggerChannelConditionsV2, ps3000aSetTriggerChannelDirections and ps3000aSetTriggerChannelProperties to set up the trigger if required.
6. Start the oscilloscope running using ps3000aRunBlock.
7. Wait until the oscilloscope is ready using the ps3000aBlockReady callback (or poll using ps3000aIsReady).
8. Use ps3000aSetDataBuffer to tell the driver where your memory buffer is.

8a. Use ps3000aSetEtsTimeBuffer or ps3000aSetEtsTimeBuffers to tell the driver where to store the sample times.

9. Transfer the block of data from the oscilloscope using ps3000aGetValues.
10. Display the data.
11. While you want to collect updated captures, repeat steps 7 to 10.
12. Repeat steps 6 to 11.

13. Stop the oscilloscope using ps3000aStop.
14. Close the oscilloscope using ps3000aCloseUnit.



## 3.7.4 Streaming mode

**Streaming mode** can capture data without the gaps that occur between blocks when using block mode. Streaming mode supports downsampling and triggering, while providing fast streaming (for example, with USB 2.0, at up to 31.25 MS/s or 32 ns per sample) when one channel is active, depending on the computer's performance. 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 then only one buffer is used per channel. When aggregation is set above 1 then two buffers (maximum and minimum) per channel are used.
- **Memory segmentation.** The memory can be divided into segments to reduce the latency of data transfers to the PC. However, this increases the risk of losing data if the PC cannot keep up with the device's sampling rate.

See Using streaming mode for programming details when using the API. When using the wrapper DLL, see Using the wrapper functions for streaming data capture.

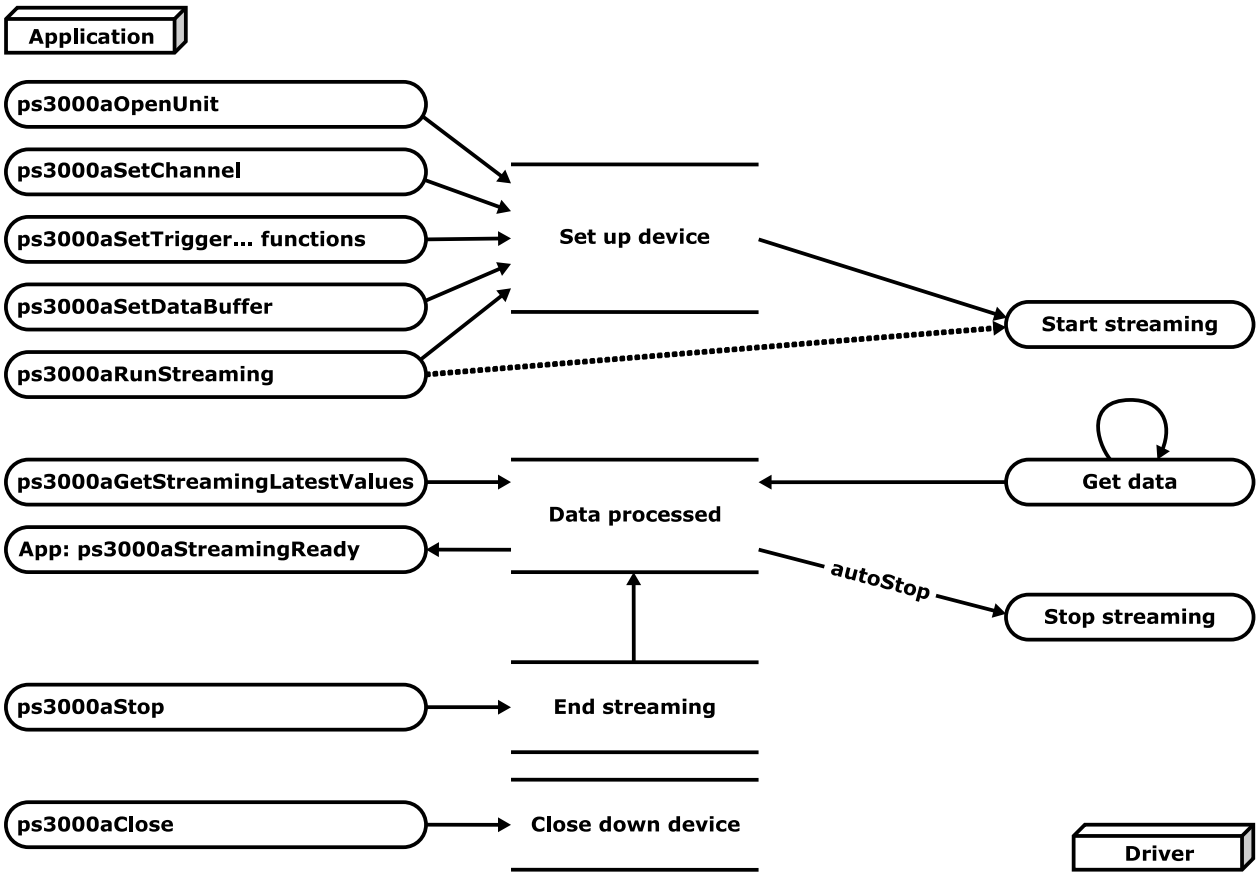
### 3.7.4.1 Using streaming mode

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

1. Open the oscilloscope using `ps3000aOpenUnit`.
2. Select channels, ranges and AC/DC coupling using `ps3000aSetChannel`.
3. *[MSOs only]* Set the digital port using `ps3000aSetDigitalPort`.
4. Use the trigger setup functions `ps3000aSetTriggerChannelConditionsV2`, `ps3000aSetTriggerChannelDirections` and `ps3000aSetTriggerChannelProperties` to set up the trigger if required.
5. *[MSOs only]* Use the trigger setup functions `ps3000aSetTriggerDigitalPortProperties` to set up the digital trigger if required.
6. Call `ps3000aSetDataBuffer` to tell the driver where your data buffer is.
7. Set up aggregation and start the oscilloscope running using `ps3000aRunStreaming`.
8. Call `ps3000aGetStreamingLatestValues` to get data.
9. 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.
10. Call `ps3000aStop`, even if `autoStop` is enabled.
11. Request new views of stored data using different downsampling parameters: see Retrieving stored data.

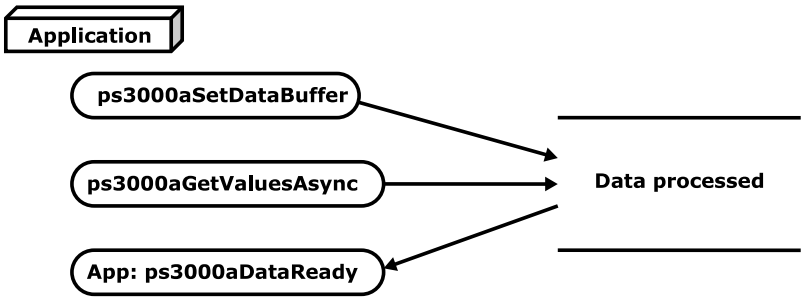
12. Close the oscilloscope using ps3000aCloseUnit.



### 3.7.5 Retrieving stored data

You can collect data from the *ps3000a* driver with a different downsampling factor when ps3000aRunBlock or ps3000aRunStreaming has already been called and has successfully captured all the data. Use ps3000aGetValuesAsync.




## 3.8 Combining several oscilloscopes

It is possible to collect data using up to 64 PicoScope oscilloscopes at the same time, depending on the capabilities of the PC. Each oscilloscope must be connected to a separate USB port. ps3000aOpenUnit 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 ps3000aBlockReady(...)
// Define callback function specific to application

handle1 = ps3000aOpenUnit
handle2 = ps3000aOpenUnit

ps3000aSetChannel(handle1)
// Set up unit 1
ps3000aSetDigitalPort   // MSO models only
ps3000aRunBlock(handle1)

ps3000aSetChannel(handle2)
// Set up unit 2
ps3000aSetDigitalPort   // MSO models only
ps3000aRunBlock(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

ps3000aCloseUnit(handle1)
ps3000aCloseUnit(handle2)
```

# 4 API functions

The *ps3000a* API exports the following 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. An additional set of wrapper functions is provided for use with programming languages that do not support callbacks.

| | |
|---|---|
| ps3000aBlockReady | indicate when block-mode data ready |
| ps3000aChangePowerSource | configure the unit's power source |
| ps3000aCloseUnit | close a scope device |
| ps3000aCurrentPowerSource | indicate the current power state of the device |
| ps3000aDataReady | indicate when post-collection data ready |
| ps3000aEnumerateUnits | find all connected oscilloscopes |
| ps3000aFlashLed | flash the front-panel LED |
| ps3000aGetAnalogueOffset | query the permitted analog offset range |
| ps3000aGetChannelInformation | query which ranges are available on a device |
| ps3000aGetMaxDownSampleRatio | query the aggregation ratio for data |
| ps3000aGetMaxEtsValues | obtain limits for the ETS parameters |
| ps3000aGetMaxSegments | query the maximum number of segments |
| ps3000aGetNoOfCaptures | find out how many captures are available |
| ps3000aGetNoOfProcessedCaptures | query number of captures processed |
| ps3000aGetStreamingLatestValues | get streaming data while scope is running |
| ps3000aGetTimebase | find out what timebases are available |
| ps3000aGetTimebase2 | find out what timebases are available |
| ps3000aGetTriggerInfoBulk | get rapid block trigger timings |
| ps3000aGetTriggerTimeOffset | find out when trigger occurred (32-bit) |
| ps3000aGetTriggerTimeOffset64 | find out when trigger occurred (64-bit) |
| ps3000aGetUnitInfo | read information about scope device |
| ps3000aGetValues | retrieve block-mode data with callback |
| ps3000aGetValuesAsync | retrieve streaming data with callback |
| ps3000aGetValuesBulk | retrieve data in rapid block mode |
| ps3000aGetValuesOverlapped | set up data collection ahead of capture |
| ps3000aGetValuesOverlappedBulk | set up data collection in rapid block mode |
| ps3000aGetValuesTriggerTimeOffsetBulk | get rapid-block waveform timings (32-bit) |
| ps3000aGetValuesTriggerTimeOffsetBulk64 | get rapid-block waveform timings (64-bit) |
| ps3000aHoldOff | not currently used |
| ps3000aIsReady | poll driver in block mode |
| ps3000aIsTriggerOrPulseWidthQualifierEnabled | find out whether trigger is enabled |
| ps3000aMaximumValue | query the max. ADC count in GetValues calls |
| ps3000aMemorySegments | divide scope memory into segments |
| ps3000aMinimumValue | query the min. ADC count in GetValues calls |
| ps3000aNoOfStreamingValues | get number of samples in streaming mode |
| ps3000aOpenUnit | open a scope device |
| ps3000aOpenUnitAsync | open a scope device without waiting |
| ps3000aOpenUnitProgress | check progress of OpenUnit call |
| ps3000aPingUnit | check communication with device |
| ps3000aQueryOutputEdgeDetect | query the output edge detect mode |
| ps3000aRunBlock | start block mode |
| ps3000aRunStreaming | start streaming mode |
| ps3000aSetBandwidthFilter | control the bandwidth limiter |
| ps3000aSetChannel | set up input channels |
| ps3000aSetDataBuffer | register data buffer with driver |
| ps3000aSetDataBuffers | register aggregated data buffers with driver |
| ps3000aSetDigitalPort | enable the digital port and set the logic level |
| ps3000aSetEts | set up equivalent-time sampling |
| ps3000aSetEtsTimeBuffer | set up buffer for ETS timings (64-bit) |
| ps3000aSetEtsTimeBuffers | set up buffer for ETS timings (32-bit) |
| ps3000aSetNoOfCaptures | set number of captures to collect in one run |
| ps3000aSetOutputEdgeDetect | switch output edge detect mode on or off |
| ps3000aSetPulseWidthDigitalPortProperties | set up pulse width triggering on digital port |
| ps3000aSetPulseWidthQualifier | set up pulse width triggering |
| ps3000aSetPulseWidthQualifierV2 | set up pulse width triggering (digital condition) |
| ps3000aSetSigGenArbitrary | set up arbitrary waveform generator |

| | |
|---|---|
| ps3000aSetSigGenBuiltIn | set up standard signal generator |
| ps3000aSetSigGenBuiltInV2 | set up signal generator (double precision) |
| ps3000aSetSigGenPropertiesArbitrary | set arbitrary waveform generator properties |
| ps3000aSetSigGenPropertiesBuiltIn | set signal generator properties |
| ps3000aSetSimpleTrigger | set up level triggers only |
| ps3000aSetTriggerChannelConditions | specify which channels to trigger on |
| ps3000aSetTriggerChannelConditionsV2 | specify trigger channels for MSOs |
| ps3000aSetTriggerChannelDirections | set up signal polarities for triggering |
| ps3000aSetTriggerChannelProperties | set up trigger thresholds |
| ps3000aSetTriggerDelay | set up post-trigger delay |
| ps3000aSetTriggerDigitalPortProperties | set individual digital channels trigger directions |
| ps3000aSigGenArbitraryMinMaxValues | query AWG parameter limits |
| ps3000aSigGenFrequencyToPhase | calculate AWG phase from frequency |
| ps3000aSigGenSoftwareControl | trigger the signal generator |
| ps3000aStop | stop data capture |
| ps3000aStreamingReady | indicate when streaming-mode data ready |

# 4.1 ps3000aBlockReady (callback)

```
typedef void (CALLBACK *ps3000aBlockReady)
(
  int16_t       handle,
  PICO_STATUS   status,
  void        * pParameter
)
```

This callback function is part of your application. You register it with the *ps3000a* driver using ps3000aRunBlock, and the driver calls it back when block-mode data is ready. You can then download the data using ps3000aGetValues.

| | |
|---|---|
| **Applicability** | Block mode only |
| **Arguments** | handle, device identifier returned by ps3000aOpenUnit<br>status, indicates whether an error occurred during collection of the data<br><br>* pParameter, a void pointer passed from ps3000aRunBlock. Your callback function can write to this location to send any data, such as a status flag, back to your application. |
| **Returns** | nothing |

# 4.2 ps3000aChangePowerSource

```
PICO_STATUS ps3000aChangePowerSource
(
  int16_t       handle,
  PICO_STATUS   powerstate
)
```

This function selects the power supply mode. You must call this function if any of the following conditions arises:

- USB power is required
- The power supply is connected or disconnected during use
- A 2-channel USB 3.0 scope is plugged into a USB 2.0 port (indicated if any function returns the PICO_USB3_0_DEVICE_NON_USB3_0_PORT status code)

Whenever the power supply mode is changed, all data and settings in the scope device are lost. You must then reconfigure the device before restarting capture.

| | |
|---|---|
| **Applicability** | All modes. 4-channel and USB 3.0 oscilloscopes only. |
| **Arguments** | handle, device identifier returned by ps3000aOpenUnit<br>powerstate, the required state of the unit. One of the following:<br>PICO_POWER_SUPPLY_CONNECTED – to use power from the external power supply<br>PICO_POWER_SUPPLY_NOT_CONNECTED – to use power from the USB port<br>PICO_USB3_0_DEVICE_NON_USB3_0_PORT – to use power from a non-USB 3.0 port |
| **Returns** | PICO_OK<br>PICO_POWER_SUPPLY_REQUEST_INVALID<br>PICO_INVALID_PARAMETER<br>PICO_NOT_RESPONDING<br>PICO_INVALID_HANDLE |

## 4.3 ps3000aCloseUnit

```
PICO_STATUS ps3000aCloseUnit
(
  int16_t   handle
)
```

This function shuts down an oscilloscope.

| | |
|---|---|
| **Applicability** | All modes |
| **Arguments** | `handle`, the device identifier, returned by `ps3000aOpenUnit`, of the scope device to be closed |
| **Returns** | `PICO_OK`<br>`PICO_HANDLE_INVALID`<br>`PICO_USER_CALLBACK`<br>`PICO_DRIVER_FUNCTION` |

# 4.4 ps3000aCurrentPowerSource

```
PICO_STATUS ps3000aCurrentPowerSource
(
  int16_t   handle
)
```

This function returns the current power state of a 4-channel device. If called for a 2-channel device, it always returns PICO_OK.

| | |
|---|---|
| **Applicability** | All modes. Intended for for 4-channel devices. |
| **Arguments** | handle, device identifier returned by ps3000aOpenUnit |
| **Returns** | PICO_POWER_SUPPLY_CONNECTED – the device is powered by the external power supply<br>PICO_POWER_SUPPLY_NOT_CONNECTED – the device is powered by the USB port<br>PICO_OK – the device has 2 channels |

# 4.5 ps3000aDataReady (callback)

```
typedef void (CALLBACK *ps3000aDataReady)
(
  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 ps3000aGetValuesAsync, and the driver calls your function back when the data is ready.

| | |
|---|---|
| **Applicability** | All modes |
| **Arguments** | handle, device identifier returned by ps3000aOpenUnit<br>status, a PICO_STATUS code returned by the driver<br><br>noOfSamples, the number of samples collected<br><br>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.<br><br>* pParameter, a void pointer passed from ps3000aGetValuesAsync. 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 |

# 4.6 ps3000aEnumerateUnits

```
PICO_STATUS ps3000aEnumerateUnits
(
  int16_t * count,
  int8_t  * serials,
  int16_t * serialLth
)
```

This function counts the number of unopened *ps3000a*-compatible scopes connected to the computer and returns a list of serial numbers as a string. It does not detect devices that have already been opened in another process.

| | |
|---|---|
| **Applicability** | All modes |
| **Arguments** | `* count`, on exit, the number of unopened *ps3000a*-compatible units found<br><br>`* 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.<br><br>`* serialLth`, on entry, the length of the `int8_t` buffer pointed to by `serials`; on exit, the length of the string written to `serials` |
| **Returns** | `PICO_OK`<br>`PICO_BUSY`<br>`PICO_NULL_PARAMETER`<br>`PICO_FW_FAIL`<br>`PICO_CONFIG_FAIL`<br>`PICO_MEMORY_FAIL`<br>`PICO_CONFIG_FAIL_AWG`<br>`PICO_INITIALISE_FPGA` |

# 4.7 ps3000aFlashLed

```
PICO_STATUS ps3000aFlashLed
(
  int16_t   handle,
  int16_t   start
)
```

This function flashes the LED on the front of the scope without blocking the calling thread. Calls to ps3000aRunStreaming and ps3000aRunBlock 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 ps3000aOpenUnit<br>start, the action required: -<br><br>< 0 : flash the LED indefinitely.<br>0 : stop the LED flashing.<br>> 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<br>PICO_HANDLE_INVALID<br>PICO_BUSY<br>PICO_DRIVER_FUNCTION<br>PICO_NOT_RESPONDING |

# 4.8 ps3000aGetAnalogueOffset

```
PICO_STATUS ps3000aGetAnalogueOffset
(
  int16_t            handle,
  PS3000A_RANGE      range,
  PS3000A_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** | AI models |
| **Arguments** | `handle`, device identifier returned by `ps3000aOpenUnit`<br>`range`, the voltage range to be used when gathering the min and max information<br><br>`coupling`, the type of AC/DC coupling used<br><br>`* maximumVoltage`, a pointer to a float, an out parameter set to the maximum voltage allowed for the range, may be `NULL`<br><br>`* minimumVoltage`, a pointer to a float, an out parameter set to the minimum voltage allowed for the range, may be `NULL`<br><br>If both `maximumVoltage` and `minimumVoltage` are set to `NULL` the driver will return `PICO_NULL_PARAMETER`. |
| **Returns** | `PICO_OK`<br>`PICO_INVALID_HANDLE`<br>`PICO_DRIVER_FUNCTION`<br>`PICO_INVALID_VOLTAGE_RANGE`<br>`PICO_NULL_PARAMETER` |

# 4.9 ps3000aGetChannelInformation

```
PICO_STATUS ps3000aGetChannelInformation
(
  int16_t                handle,
  PS3000A_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 `ps3000aOpenUnit`<br>`info`, the type of information required. The following value is currently supported:<br>`PS3000A_CI_RANGES`<br><br>`probe`, not used, must be set to 0<br><br>`* ranges`, an array that will be populated with available `PS3000A_RANGE` values for the given info. If `NULL`, length is set to the number of ranges available.<br><br>`* length`, on input: the length of the ranges array; on output: the number of elements written to ranges array<br><br>`channels`, the channel for which the information is required |
| **Returns** | `PICO_OK`<br>`PICO_HANDLE_INVALID`<br>`PICO_BUSY`<br>`PICO_DRIVER_FUNCTION`<br>`PICO_NOT_RESPONDING`<br>`PICO_NULL_PARAMETER`<br>`PICO_INVALID_CHANNEL`<br>`PICO_INVALID_INFO` |

# 4.10 ps3000aGetMaxDownSampleRatio

```
PICO_STATUS ps3000aGetMaxDownSampleRatio
(
  int16_t              handle,
  uint32_t             noOfUnaggregatedSamples,
  uint32_t           * maxDownSampleRatio,
  PS3000A_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 `ps3000aOpenUnit`<br>`noOfUnaggregatedSamples`, the number of unprocessed samples to be downsampled<br><br>`* maxDownSampleRatio`, the maximum possible downsampling ratio output<br><br>`downSampleRatioMode`, the downsampling mode. See `ps3000aGetValues`.<br><br>`segmentIndex`, the memory segment where the data is stored |
| **Returns** | `PICO_OK`<br>`PICO_INVALID_HANDLE`<br>`PICO_NO_SAMPLES_AVAILABLE`<br>`PICO_NULL_PARAMETER`<br>`PICO_INVALID_PARAMETER`<br>`PICO_SEGMENT_OUT_OF_RANGE`<br>`PICO_TOO_MANY_SAMPLES` |

# 4.11 ps3000aGetMaxEtsValues

```
PICO_STATUS ps3000aGetMaxEtsValues
(
  int16_t   handle,
  int16_t * etsCycles,
  int16_t * etsInterleave
)
```

This function returns the maximum number of cycles and maximum interleaving factor that can be used for the selected scope device in ETS mode. These values are the upper limits for the `etsCycles` and `etsInterleave` arguments supplied to `ps3000SetEts`.

| | |
|---|---|
| **Applicability** | All modes |
| **Arguments** | `handle`, device identifier returned by `ps3000aOpenUnit`<br>`etsCycles`, the maximum value of the `etsCycles` argument supplied to `ps3000SetEts`<br><br>`etsInterleave`, the maximum value of the `etsInterleave` argument supplied to `ps3000SetEts` |
| **Returns** | `PICO_OK`<br>`PICO_INVALID_HANDLE`<br>`PICO_DRIVER_FUNCTION`<br>`PICO_NULL_PARAMETER` - if `etsCycles` and `etsInterleave` are both `NULL` |

# 4.12 ps3000aGetMaxSegments

```
PICO_STATUS ps3000aGetMaxSegments
(
  int16_t    handle,
  uint32_t * maxsegments
 )
```

This function returns the maximum number of segments allowed for the opened device. This number is the maximum value of `nsegments` that can be passed to `ps3000aMemorySegments`.

| | |
|---|---|
| **Applicability** | All modes |
| **Arguments** | `handle`, device identifier returned by `ps3000aOpenUnit`<br>`* maxsegments`, on exit, the maximum number of segments allowed |
| **Returns** | `PICO_OK`<br>`PICO_INVALID_HANDLE`<br>`PICO_DRIVER_FUNCTION`<br>`PICO_NULL_PARAMETER` |

# 4.13 ps3000aGetNoOfCaptures

```
PICO_STATUS ps3000aGetNoOfCaptures
(
  int16_t    handle,
  uint32_t * nCaptures
)
```

This function returns the number of waveforms that the device has captured. It can be called during waveform capture.

It can be called in rapid block mode after ps3000aRunBlock has been called and either the collection completed or the collection of waveforms was interrupted by calling ps3000aStop. The returned value (nCaptures) can then be used to iterate through the number of segments using ps3000aGetValues, or in a single call to ps3000aGetValuesBulk where it is used to calculate the toSegmentIndex parameter.

| | |
|---|---|
| **Applicability** | Rapid block mode |
| **Arguments** | handle, device identifier returned by ps3000aOpenUnit<br>* nCaptures, output: the number of available captures that has been collected from calling ps3000aRunBlock |
| **Returns** | PICO_OK<br>PICO_DRIVER_FUNCTION<br>PICO_INVALID_HANDLE<br>PICO_NOT_RESPONDING<br>PICO_NO_SAMPLES_AVAILABLE<br>PICO_NULL_PARAMETER<br>PICO_INVALID_PARAMETER<br>PICO_SEGMENT_OUT_OF_RANGE<br>PICO_TOO_MANY_SAMPLES |

# 4.14 ps3000aGetNoOfProcessedCaptures

```
PICO_STATUS ps3000aGetNoOfProcessedCaptures
(
  int16_t     handle,
  uint32_t  * nProcessedCaptures
)
```

This function gets the number of captures collected and processed in one run of rapid block mode. It enables your application to start processing captured data while the driver is still transferring later captures from the device to the computer.

The function returns the number of captures the driver has processed since you called ps3000aRunBlock. It is for use in rapid block mode, alongside the ps3000aGetValuesOverlappedBulk function, when the driver is set to transfer data from the device automatically as soon as the ps3000aRunBlock function is called. You can call ps3000aGetNoOfProcessedCaptures during device capture, after collection has completed or after interrupting waveform collection by calling ps3000aStop.

The returned value (nProcessedCaptures) can then be used to iterate through the number of segments using ps3000aGetValues, or in a single call to ps3000aGetValuesBulk, where it is used to calculate the toSegmentIndex parameter.

**When capture is stopped**

If nProcessedCaptures = 0, you will also need to call ps3000aGetNoOfCaptures, in order to determine how many waveform segments were captured, before calling ps3000aGetValues or ps3000aGetValuesBulk.

| | |
|---|---|
| **Applicability** | Rapid block mode |
| **Arguments** | handle, the handle of the device.<br><br>* nProcessedCaptures, on exit, the number of waveforms captured and processed. |
| **Returns** | PICO_OK<br>PICO_INVALID_HANDLE<br>PICO_INVALID_PARAMETER |

# 4.15 ps3000aGetStreamingLatestValues

```
PICO_STATUS ps3000aGetStreamingLatestValues
(
  int16_t                 handle,
  ps3000aStreamingReady   lpPs3000AReady,
  void                  * pParameter
)
```

This function instructs the driver to return the next block of values to your ps3000aStreamingReady callback. You must have previously called ps3000aRunStreaming beforehand to set up streaming.

| | |
|---|---|
| **Applicability** | Streaming mode only |
| **Arguments** | handle, device identifier returned by ps3000aOpenUnit<br>lpPs3000AReady, a pointer to your ps3000aStreamingReady callback<br><br>* pParameter, a void pointer that will be passed to the ps3000aStreamingReady callback. The callback may optionally use this pointer to return information to the application. |
| **Returns** | PICO_OK<br>PICO_POWER_SUPPLY_CONNECTED<br>PICO_POWER_SUPPLY_NOT_CONNECTED<br>PICO_INVALID_HANDLE<br>PICO_NO_SAMPLES_AVAILABLE<br>PICO_INVALID_CALL<br>PICO_BUSY<br>PICO_NOT_RESPONDING<br>PICO_DRIVER_FUNCTION |

# 4.16 ps3000aGetTimebase

```
PICO_STATUS ps3000aGetTimebase
(
  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 will depend on the number of channels enabled by the last call to ps3000aSetChannel.

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 ps3000aGetTimebase2 instead.

To use ps3000aGetTimebase or ps3000aGetTimebase2, 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 `timeIntervalNanoseconds` argument that the function returns is the value that 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 ps3000aOpenUnit<br>`timebase`, see timebase guide. This value can be supplied to ps3000aRunBlock to define the sampling interval.<br><br>`noSamples`, the number of samples required<br><br>`* timeIntervalNanoseconds`, on exit, the time interval between readings at the selected timebase. Use NULL if not required.<br><br>`oversample`, not used<br><br>`* 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.<br><br>`segmentIndex`, the index of the memory segment to use |
| **Returns** | `PICO_OK`<br>`PICO_INVALID_HANDLE`<br>`PICO_TOO_MANY_SAMPLES`<br>`PICO_INVALID_CHANNEL`<br>`PICO_INVALID_TIMEBASE`<br>`PICO_INVALID_PARAMETER`<br>`PICO_SEGMENT_OUT_OF_RANGE`<br>`PICO_DRIVER_FUNCTION` |

# 4.17 ps3000aGetTimebase2

```
PICO_STATUS ps3000aGetTimebase2
(
  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 ps3000aGetTimebase, and returns the time interval as a `float` rather than an `int32_t`. This allows it to return sub-nanosecond time intervals. See ps3000aGetTimebase 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.<br><br>All other arguments: see ps3000aGetTimebase. |
| **Returns** | See ps3000aGetTimebase. |

# 4.18 ps3000aGetTriggerInfoBulk

```
PICO_STATUS ps3000aGetTriggerInfoBulk
(
  int16_t                handle,
  PS3000A_TRIGGER_INFO * triggerInfo,
  uint32_t               fromSegmentIndex,
  uint32_t               toSegmentIndex
)
```

This function returns trigger information in rapid block mode.

| | |
|---|---|
| **Applicability** | Rapid block mode.<br>PicoScope 3207A and 3207B only. |
| **Arguments** | `handle`, device identifier returned by `ps3000aOpenUnit`<br>`triggerInfo`, an array of pointers to `PS3000A_TRIGGER_INFO` structures that, on exit, will contain information on each trigger event. There will be one structure for each segment in the range `[fromSegmentIndex, toSegmentIndex]`.<br><br>`fromSegmentIndex`, the number of the first memory segment for which information is required<br><br>`toSegmentIndex`, the number of the last memory segment for which information is required |
| **Returns** | `PICO_NOT_SUPPORTED_BY_THIS_DEVICE`<br>`PICO_NO_SAMPLES_AVAILABLE`<br>`PICO_NULL_PARAMETER`<br>`PICO_SEGMENT_OUT_OF_RANGE`<br>`PICO_NOT_USED_IN_THIS_CAPTURE_MODE`<br>`PICO_ETS_MODE_SET`<br>`PICO_OK`<br>`PICO_NOT_RESPONDING`<br>`PICO_INVALID_HANDLE`<br>`PICO_DRIVER_FUNCTION` |

# 4.19 ps3000aGetTriggerTimeOffset

```
PICO_STATUS ps3000aGetTriggerTimeOffset
(
  int16_t             handle,
  uint32_t          * timeUpper,
  uint32_t          * timeLower,
  PS3000A_TIME_UNITS * timeUnits,
  uint32_t            segmentIndex
)
```

This function gets the trigger time offset for waveforms obtained in block mode or rapid block mode. The trigger time offset is an adjustment value used for correcting jitter in the waveform, and is intended mainly for applications that wish to display the waveform with reduced jitter. The offset is zero if the waveform crosses the threshold at the trigger sampling instant, or a positive or negative value if jitter correction is required. The value should be added to the nominal trigger time to get the corrected trigger time.

Call this function after data has been captured or when data has been retrieved from a previous capture.

This function is provided for use in programming environments that do not support 64-bit integers. Another version of this function, ps3000aGetTriggerTimeOffset64, is available that returns the time as a single 64-bit value.

| | |
|---|---|
| **Applicability** | Block mode, rapid block mode |
| **Arguments** | `handle`, device identifier returned by ps3000aOpenUnit<br>`* timeUpper`, on exit, the upper 32 bits of the trigger time offset<br><br>`* timeLower`, on exit, the lower 32 bits of the trigger time offset<br><br>`* timeUnits`, returns the time units in which `timeUpper`:`timeLower` is measured. The allowable values are:<br>PS3000A_FS<br>PS3000A_PS<br>PS3000A_NS<br>PS3000A_US<br>PS3000A_MS<br>PS3000A_S<br><br>`segmentIndex`, the number of the memory segment for which the information is required |
| **Returns** | PICO_OK<br>PICO_INVALID_HANDLE<br>PICO_DEVICE_SAMPLING<br>PICO_SEGMENT_OUT_OF_RANGE<br>PICO_NOT_USED_IN_THIS_CAPTURE_MODE<br>PICO_NOT_RESPONDING<br>PICO_NULL_PARAMETER<br>PICO_NO_SAMPLES_AVAILABLE<br>PICO_DRIVER_FUNCTION |

# 4.20 ps3000aGetTriggerTimeOffset64

```
PICO_STATUS ps3000aGetTriggerTimeOffset64
(
  int16_t             handle,
  int64_t           * time,
  PS3000A_TIME_UNITS * timeUnits,
  uint32_t            segmentIndex
)
```

This function gets the trigger time offset for a waveform. It is equivalent to ps3000aGetTriggerTimeOffset except that the time offset is returned as a single 64-bit value instead of two 32-bit values.

| | |
|---|---|
| **Applicability** | Block mode, rapid block mode |
| **Arguments** | handle, device identifier returned by ps3000aOpenUnit<br>* time, on exit, the time at which the trigger point occurred<br><br>* timeUnits,<br>segmentIndex, see ps3000aGetTriggerTimeOffset |
| **Returns** | PICO_OK<br>PICO_INVALID_HANDLE<br>PICO_DEVICE_SAMPLING<br>PICO_SEGMENT_OUT_OF_RANGE<br>PICO_NOT_USED_IN_THIS_CAPTURE_MODE<br>PICO_NOT_RESPONDING<br>PICO_NULL_PARAMETER<br>PICO_NO_SAMPLES_AVAILABLE<br>PICO_DRIVER_FUNCTION |

# 4.21 ps3000aGetUnitInfo

```
PICO_STATUS ps3000aGetUnitInfo
(
  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`, the identifier of the device to query. If an invalid handle is passed, only the driver versions can be read.<br><br>`* string`, on exit, the information string selected specified by the `info` argument. If `string` is NULL, only `requiredSize` is returned.<br><br>`stringLength`, on entry, the maximum number of `int8_t` that may be written to `string`<br><br>`* requiredSize`, on exit, the required length of the `string` array<br><br>`info`, a number specifying what information is required. The possible values are listed in the table below. |
| **Returns** | `PICO_OK`<br>`PICO_INVALID_HANDLE`<br>`PICO_NULL_PARAMETER`<br>`PICO_INVALID_INFO`<br>`PICO_INFO_UNAVAILABLE`<br>`PICO_DRIVER_FUNCTION` |

| info | | Example |
|---|---|---|
| 0 | PICO_DRIVER_VERSION<br>Version number of PicoScope ps3000a DLL | 1.0.0.1 |
| 1 | PICO_USB_VERSION<br>Type of USB connection to device: 1.1, 2.0 or 3.0 | 2.0 |
| 2 | PICO_HARDWARE_VERSION<br>Hardware version of device | 1 |
| 3 | PICO_VARIANT_INFO<br>Variant number of device | 3206B |
| 4 | PICO_BATCH_AND_SERIAL<br>Batch and serial number of device | KJL87/006 |
| 5 | PICO_CAL_DATE<br>Calibration date of device | 30Sep09 |
| 6 | PICO_KERNEL_VERSION<br>Version of kernel driver | 1.0 |
| 7 | PICO_DIGITAL_HARDWARE_VERSION<br>Hardware version of the digital section | 1 |
| 8 | PICO_ANALOGUE_HARDWARE_VERSION<br>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 |

# 4.22 ps3000aGetValues

```
PICO_STATUS ps3000aGetValues
(
  int16_t              handle,
  uint32_t             startIndex,
  uint32_t           * noOfSamples,
  uint32_t             downSampleRatio,
  PS3000A_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 ps3000aSetDataBuffer() or ps3000aSetDataBuffers(). It blocks the calling function while retrieving data.

| | |
|---|---|
| **Applicability** | Block mode, rapid block mode |
| **Arguments** | `handle`, device identifier returned by `ps3000aOpenUnit`<br>`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.<br><br>`* 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`.<br><br>`downSampleRatio`, the downsampling factor that will be applied to the raw data<br><br>`downSampleRatioMode`, which downsampling mode to use. The available values are: -<br>`PS3000A_RATIO_MODE_NONE` (`downSampleRatio` is ignored)<br>`PS3000A_RATIO_MODE_AGGREGATE`<br>`PS3000A_RATIO_MODE_AVERAGE`<br>`PS3000A_RATIO_MODE_DECIMATE`<br><br>`AGGREGATE`, `AVERAGE`, `DECIMATE` are single-bit constants that can be ORed to apply multiple downsampling modes to the same data<br><br>`segmentIndex`, the zero-based number of the memory segment where the data is stored<br><br>`* 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<br>PICO_INVALID_HANDLE<br>PICO_POWER_SUPPLY_CONNECTED<br>PICO_POWER_SUPPLY_NOT_CONNECTED<br>PICO_NO_SAMPLES_AVAILABLE<br>PICO_DEVICE_SAMPLING<br>PICO_NULL_PARAMETER<br>PICO_SEGMENT_OUT_OF_RANGE<br>PICO_STARTINDEX_INVALID<br>PICO_ETS_NOT_RUNNING<br>PICO_BUFFERS_NOT_SET<br>PICO_INVALID_PARAMETER<br>PICO_TOO_MANY_SAMPLES<br>PICO_DATA_NOT_AVAILABLE<br>PICO_STARTINDEX_INVALID<br>PICO_INVALID_SAMPLERATIO<br>PICO_INVALID_CALL<br>PICO_NOT_RESPONDING<br>PICO_MEMORY<br>PICO_RATIO_MODE_NOT_SUPPORTED<br>PICO_DRIVER_FUNCTION |
|---|---|

## 4.22.1 Downsampling modes

Various methods of data reduction, or **downsampling**, are possible with PicoScope oscilloscopes. The downsampling is done at high speed by dedicated hardware inside the scope, making your application faster and more responsive than if you had to do all the data processing in software.

You specify the downsampling mode when you call one of the data collection functions such as ps3000aGetValues. The following modes are available:

| | |
|---|---|
| PS3000A_RATIO_MODE_NONE | No downsampling. Returns the raw data values. |
| PS3000A_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. |
| PS3000A_RATIO_MODE_DECIMATE | Reduces every block of *n* values to just the first value in the block, discarding all the other values. |
| PS3000A_RATIO_MODE_AVERAGE | Reduces every block of *n* values to a single value representing the average (arithmetic mean) of all the values. |

# 4.23 ps3000aGetValuesAsync

```
PICO_STATUS ps3000aGetValuesAsync
(
  int16_t              handle,
  uint32_t             startIndex,
  uint32_t             noOfSamples,
  uint32_t             downSampleRatio,
  PS3000A_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 device (in block mode) or the driver (in streaming mode) after data collection has stopped. It returns the data using a callback.

| | |
|---|---|
| **Applicability** | Streaming mode and block mode |
| **Arguments** | handle, device identifier returned by ps3000aOpenUnit<br>startIndex,<br>noOfSamples,<br>downSampleRatio,<br>downSampleRatioMode,<br>segmentIndex: see ps3000aGetValues<br><br>* lpDataReady, a pointer to the user-supplied function that will be called when the data is ready. This will be ps3000aDataReady for block-mode data or ps3000aStreamingReady for streaming-mode data.<br><br>* pParameter, a void pointer that will be passed to the callback function. The data type is determined by the application. |
| **Returns** | PICO_OK<br>PICO_POWER_SUPPLY_CONNECTED<br>PICO_POWER_SUPPLY_NOT_CONNECTED<br>PICO_INVALID_HANDLE<br>PICO_NO_SAMPLES_AVAILABLE<br>PICO_DEVICE_SAMPLING<br>PICO_NULL_PARAMETER<br>PICO_STARTINDEX_INVALID<br>PICO_SEGMENT_OUT_OF_RANGE<br>PICO_INVALID_PARAMETER<br>PICO_DATA_NOT_AVAILABLE<br>PICO_INVALID_SAMPLERATIO<br>PICO_INVALID_CALL<br>PICO_DRIVER_FUNCTION |

# 4.24 ps3000aGetValuesBulk

```
PICO_STATUS ps3000aGetValuesBulk
(
  int16_t              handle,
  uint32_t           * noOfSamples,
  uint32_t             fromSegmentIndex,
  uint32_t             toSegmentIndex,
  uint32_t             downSampleRatio,
  PS3000A_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.

| | |
|---|---|
| **Applicability** | Rapid block mode |
| **Arguments** | `handle`, device identifier returned by `ps3000aOpenUnit`<br>`* 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.<br><br>`fromSegmentIndex`, the first segment from which the waveform should be retrieved<br><br>`toSegmentIndex`, the last segment from which the waveform should be retrieved<br><br>`downSampleRatio`,<br>`downSampleRatioMode`, see `ps3000aGetValues`<br><br>`* 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 `ps3000aGetValues`. |
| **Returns** | `PICO_OK`<br>`PICO_POWER_SUPPLY_CONNECTED`<br>`PICO_POWER_SUPPLY_NOT_CONNECTED`<br>`PICO_INVALID_HANDLE`<br>`PICO_INVALID_PARAMETER`<br>`PICO_INVALID_SAMPLERATIO`<br>`PICO_ETS_NOT_RUNNING`<br>`PICO_BUFFERS_NOT_SET`<br>`PICO_TOO_MANY_SAMPLES`<br>`PICO_SEGMENT_OUT_OF_RANGE`<br>`PICO_NO_SAMPLES_AVAILABLE`<br>`PICO_NOT_RESPONDING`<br>`PICO_DRIVER_FUNCTION` |

# 4.25 ps3000aGetValuesOverlapped

```
PICO_STATUS ps3000aGetValuesOverlapped
(
  int16_t              handle,
  uint32_t             startIndex,
  uint32_t           * noOfSamples,
  uint32_t             downSampleRatio,
  PS3000A_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 ps3000aRunBlock. The advantage of this function is that the driver makes contact with the scope only once, when you call ps3000aRunBlock, compared with the two contacts that occur when you use the conventional ps3000aRunBlock, ps3000aGetValues calling sequence. This slightly reduces the dead time between successive captures in block mode.

After calling ps3000aRunBlock, you can optionally use ps3000aGetValues to request further copies of the data. This might be required if you wish to display the data with different data reduction settings.

See also: Using the GetValuesOverlapped functions.

| | |
|---|---|
| **Applicability** | Block mode |
| **Arguments** | handle, device identifier returned by ps3000aOpenUnit<br>startIndex,<br>* noOfSamples,<br>downSampleRatio,<br>downSampleRatioMode,<br>segmentIndex: see ps3000aGetValues<br>* overflow, see ps3000aGetValuesBulk |
| **Returns** | PICO_OK<br>PICO_POWER_SUPPLY_CONNECTED<br>PICO_POWER_SUPPLY_NOT_CONNECTED<br>PICO_INVALID_HANDLE<br>PICO_INVALID_PARAMETER<br>PICO_DRIVER_FUNCTION |

# 4.26 ps3000aGetValuesOverlappedBulk

```
PICO_STATUS ps3000aGetValuesOverlappedBulk
(
  int16_t              handle,
  uint32_t             startIndex,
  uint32_t           * noOfSamples,
  uint32_t             downSampleRatio,
  PS3000A_RATIO_MODE   downSampleRatioMode,
  uint32_t             fromSegmentIndex,
  uint32_t             toSegmentIndex,
  int16_t            * overflow
)
```

This function requests data from multiple segments in rapid block mode. It is similar to calling ps3000aGetValuesOverlapped multiple times, but more efficient.

See also: Using the GetValuesOverlapped functions.

| | |
|---|---|
| **Applicability** | Rapid block mode |
| **Arguments** | handle, device identifier returned by ps3000aOpenUnit<br>startIndex,<br>* noOfSamples,<br>downSampleRatio,<br>downSampleRatioMode: see ps3000aGetValues<br><br>fromSegmentIndex,<br>toSegmentIndex,<br>* overflow: see ps3000aGetValuesBulk |
| **Returns** | PICO_OK<br>PICO_POWER_SUPPLY_CONNECTED<br>PICO_POWER_SUPPLY_NOT_CONNECTED<br>PICO_INVALID_HANDLE<br>PICO_INVALID_PARAMETER<br>PICO_DRIVER_FUNCTION |

### 4.26.1 Using the GetValuesOverlapped functions

1. Open the oscilloscope using ps3000aOpenUnit.
2. Select channel ranges and AC/DC coupling using ps3000aSetChannel.
3. Using ps3000aGetTimebase, select timebases until the required sampling interval is located.
4. Use the trigger setup functions ps3000aSetTriggerChannelDirections and ps3000aSetTriggerChannelProperties to set up the trigger if required.
5. Use ps3000aSetDataBuffer to tell the driver where your memory buffer is.
6. Set up the transfer of the block of data from the oscilloscope using ps3000aGetValuesOverlapped.
7. Start the oscilloscope running using ps3000aRunBlock.
8. Wait until the oscilloscope is ready using the ps3000aBlockReady callback (or poll using ps3000aIsReady).
9. Display the data.
10. Repeat steps 7 to 9 if needed.
11. Stop the oscilloscope using ps3000aStop.

A similar procedure can be used with rapid block mode using the ps3000aGetValuesOverlappedBulk function.

# 4.27 ps3000aGetValuesTriggerTimeOffsetBulk

```
PICO_STATUS ps3000aGetValuesTriggerTimeOffsetBulk
(
  int16_t              handle,
  uint32_t           * timesUpper,
  uint32_t           * timesLower,
  PS3000A_TIME_UNITS * timeUnits,
  uint32_t             fromSegmentIndex,
  uint32_t             toSegmentIndex
)
```

This function retrieves the trigger time offset for multiple waveforms obtained in block mode or rapid block mode. It is a more efficient alternative to calling ps3000aGetTriggerTimeOffset once for each waveform required. See ps3000aGetTriggerTimeOffset for an explanation of trigger time offsets.

There is another version of this function, ps3000aGetValuesTriggerTimeOffsetBulk64, that returns trigger time offsets as 64-bit values instead of pairs of 32-bit values.

| **Applicability** | Block mode, rapid block mode |
|---|---|
| **Arguments** | |
| handle, device identifier returned by ps3000aOpenUnit<br>* timesUpper, * timesLower, two arrays of integers. On exit, they hold the most significant 32 bits and least significant 32 bits of the trigger time offset for each requested segment index. timesUpper[0] and timesLower[0] hold the fromSegmentIndex time offset and the last timesUpper and timesLower elements hold the toSegmentIndex time offset. The arrays must be long enough to hold the number of requested times.<br><br>* timeUnits, an array of integers. On exit, timeUnits[0] contains the time unit for fromSegmentIndex and the last element contains the time unit for toSegmentIndex. Refer to ps3000aGetTriggerTimeOffset for allowable values. The array must be long enough to hold the number of requested times.<br><br>fromSegmentIndex, the first segment for which the time offset is required<br><br>toSegmentIndex, the last segment for which the time offset is required. If toSegmentIndex is less than fromSegmentIndex, the driver will wrap around from the last segment to the first. | |
| **Returns** | PICO_OK<br>PICO_POWER_SUPPLY_CONNECTED<br>PICO_POWER_SUPPLY_NOT_CONNECTED<br>PICO_INVALID_HANDLE<br>PICO_NOT_USED_IN_THIS_CAPTURE_MODE<br>PICO_NOT_RESPONDING<br>PICO_NULL_PARAMETER<br>PICO_DEVICE_SAMPLING<br>PICO_SEGMENT_OUT_OF_RANGE<br>PICO_NO_SAMPLES_AVAILABLE<br>PICO_DRIVER_FUNCTION |

# 4.28 ps3000aGetValuesTriggerTimeOffsetBulk64

```
PICO_STATUS ps3000aGetValuesTriggerTimeOffsetBulk64
(
  int16_t               handle,
  int64_t             * times,
  PS3000A_TIME_UNITS  * timeUnits,
  uint32_t              fromSegmentIndex,
  uint32_t              toSegmentIndex
)
```

This function is equivalent to ps3000aGetValuesTriggerTimeOffsetBulk but retrieves the trigger time offsets as 64-bit values instead of pairs of 32-bit values.

| | |
|---|---|
| **Applicability** | Block mode, rapid block mode |
| **Arguments** | handle, device identifier returned by ps3000aOpenUnit<br>* times, an array of integers. On exit, this holds the trigger time offset for each requested segment index. Each value is equivalent to the timesUpper:timesLower value returned by ps3000aGetValuesTriggerTimeOffsetBulk. See the description of that function for more information.<br><br>* timeUnits,<br>fromSegmentIndex,<br>toSegmentIndex, see ps3000aGetValuesTriggerTimeOffsetBulk |
| **Returns** | PICO_OK<br>PICO_POWER_SUPPLY_CONNECTED<br>PICO_POWER_SUPPLY_NOT_CONNECTED<br>PICO_INVALID_HANDLE<br>PICO_NOT_USED_IN_THIS_CAPTURE_MODE<br>PICO_NOT_RESPONDING<br>PICO_NULL_PARAMETER<br>PICO_DEVICE_SAMPLING<br>PICO_SEGMENT_OUT_OF_RANGE<br>PICO_NO_SAMPLES_AVAILABLE<br>PICO_DRIVER_FUNCTION |

# 4.29 ps3000aHoldOff

```
PICO_STATUS ps3000aHoldOff
(
  int16_t                handle,
  uint64_t               holdoff,
  PS3000A_HOLDOFF_TYPE   type
)
```

This function is for backward compatibility only and does nothing.

| | |
|---|---|
| **Applicability** | None |
| **Arguments** | `handle`, device identifier returned by `ps3000aOpenUnit`<br>`holdoff`, not used<br>`type`, not used |
| **Returns** | Undefined |

# 4.30 ps3000aIsReady

```
PICO_STATUS ps3000aIsReady
(
  int16_t   handle,
  int16_t * ready
)
```

This function may be used instead of a callback function to receive data from ps3000aRunBlock. To use this method, pass a NULL pointer as the lpReady argument to ps3000aRunBlock. 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 ps3000aOpenUnit<br>* ready, output: indicates the state of the collection. If zero, the device is still collecting. If non-zero, the device has finished collecting and ps3000aGetValues can be used to retrieve the data. |
| **Returns** | PICO_OK<br>PICO_INVALID_HANDLE<br>PICO_DRIVER_FUNCTION<br>PICO_NULL_PARAMETER<br>PICO_NO_SAMPLES_AVAILABLE<br>PICO_CANCELLED<br>PICO_NOT_RESPONDING |

# 4.31 ps3000aIsTriggerOrPulseWidthQualifierEnabled

```
PICO_STATUS ps3000aIsTriggerOrPulseWidthQualifierEnabled
(
  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 ps3000aRunBlock or ps3000aRunStreaming. |
| **Arguments** | handle, device identifier returned by ps3000aOpenUnit<br>* triggerEnabled, on exit, indicates whether the trigger will successfully be set when ps3000aRunBlock or ps3000aRunStreaming is called. A non-zero value indicates that the trigger is set, zero that the trigger is not set.<br><br>* pulseWidthQualifierEnabled, on exit, indicates whether the pulse width qualifier will successfully be set when ps3000aRunBlock or ps3000aRunStreaming 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<br>PICO_INVALID_HANDLE<br>PICO_NULL_PARAMETER<br>PICO_DRIVER_FUNCTION |

# 4.32 ps3000aMaximumValue

```
PICO_STATUS ps3000aMaximumValue
(
  int16_t   handle,
  int16_t * value
)
```

This function returns the maximum ADC count returned by calls to get values.

| | |
|---|---|
| **Applicability** | All modes |
| **Arguments** | `handle`, device identifier returned by `ps3000aOpenUnit`<br>`* value`, returns the maximum ADC value |
| **Returns** | `PICO_OK`<br>`PICO_USER_CALLBACK`<br>`PICO_INVALID_HANDLE`<br>`PICO_TOO_MANY_SEGMENTS`<br>`PICO_MEMORY`<br>`PICO_DRIVER_FUNCTION` |

# 4.33 ps3000aMemorySegments

```
PICO_STATUS ps3000aMemorySegments
(
  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** | All modes |
| **Arguments** | `handle`, device identifier returned by `ps3000aOpenUnit`<br>`nSegments`, the number of segments required, from 1 to the value of `maxsegments` returned by `ps3000aGetMaxSegments`<br><br>`* nMaxSamples`, on exit, the number of samples available in each segment. This is the total number over all channels, so if more than one channel is in use, the number of samples available to each channel is `nMaxSamples` divided by 2 (for 2 channels) or 4 (for 3 or 4 channels). |
| **Returns** | `PICO_OK`<br>`PICO_USER_CALLBACK`<br>`PICO_INVALID_HANDLE`<br>`PICO_TOO_MANY_SEGMENTS`<br>`PICO_MEMORY`<br>`PICO_DRIVER_FUNCTION` |

# 4.34 ps3000aMinimumValue

```
PICO_STATUS ps3000aMinimumValue
(
  int16_t   handle,
  int16_t * value
)
```

This function returns the minimum ADC count returned by calls to ps3000aGetValues and related functions

| | |
|---|---|
| **Applicability** | All modes |
| **Arguments** | handle, device identifier returned by ps3000aOpenUnit<br>* value, returns the minimum ADC value |
| **Returns** | PICO_OK<br>PICO_USER_CALLBACK<br>PICO_INVALID_HANDLE<br>PICO_TOO_MANY_SEGMENTS<br>PICO_MEMORY<br>PICO_DRIVER_FUNCTION |

# 4.35 ps3000aNoOfStreamingValues

```
PICO_STATUS ps3000aNoOfStreamingValues
(
  int16_t    handle,
  uint32_t * noOfValues
)
```

This function returns the number of samples available after data collection in streaming mode. Call it after calling ps3000aStop. The maximum number possible is the sum of the maxPreTriggerSamples + maxPostTriggerSamples arguments passed to ps3000aRunStreaming.

| | |
|---|---|
| **Applicability** | Streaming mode |
| **Arguments** | handle, device identifier returned by ps3000aOpenUnit<br>* noOfValues, on exit, the number of samples |
| **Returns** | PICO_OK<br>PICO_INVALID_HANDLE<br>PICO_NULL_PARAMETER<br>PICO_NO_SAMPLES_AVAILABLE<br>PICO_NOT_USED<br>PICO_BUSY<br>PICO_DRIVER_FUNCTION |

# 4.36 ps3000aOpenUnit

```
PICO_STATUS ps3000aOpenUnit
(
  int16_t * handle,
  int8_t  * serial
)
```

This function opens a PicoScope 3000 Series oscilloscope attached to the computer. The maximum number of units that can be opened depends on the operating system, the kernel driver and the computer.

If the function returns PICO_POWER_SUPPLY_NOT_CONNECTED, call ps3000aChangePowerSource to switch from the external power supply to USB power. If the return value is PICO_USB3_0_DEVICE_NON_USB3_0_PORT, call ps3000aChangePowerSource to tell the driver to power the device from a USB 2.0 or USB 1.1 port.

| | |
|---|---|
| **Applicability** | All modes |
| **Arguments** | * handle, on exit, the result of the attempt to open a scope:<br>−1 : if the scope fails to open<br>0 : if no scope is found<br>> 0 : a number that uniquely identifies the scope<br>If a valid handle is returned, it must be used in all subsequent calls to API functions to identify this scope.<br><br>* 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<br>PICO_OS_NOT_SUPPORTED<br>PICO_OPEN_OPERATION_IN_PROGRESS<br>PICO_EEPROM_CORRUPT<br>PICO_KERNEL_DRIVER_TOO_OLD<br>PICO_FPGA_FAIL<br>PICO_MEMORY_CLOCK_FREQUENCY<br>PICO_FW_FAIL<br>PICO_MAX_UNITS_OPENED<br>PICO_NOT_FOUND (if the specified unit was not found)<br>PICO_NOT_RESPONDING<br>PICO_MEMORY_FAIL<br>PICO_ANALOG_BOARD<br>PICO_CONFIG_FAIL_AWG<br>PICO_INITIALISE_FPGA<br>PICO_POWER_SUPPLY_NOT_CONNECTED (if the device is a 4-channel scope with no power supply connected)<br>PICO_USB3_0_DEVICE_NON_USB3_0_PORT (if the device is a 2-channel USB 3.0 scope connected to a non-USB 3.0 port) |

# 4.37 ps3000aOpenUnitAsync

```
PICO_STATUS ps3000aOpenUnitAsync
(
  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 ps3000aOpenUnitProgress until that function returns a non-zero value.

| | |
|---|---|
| **Applicability** | All modes |
| **Arguments** | * status, a status code:<br>0 if the open operation was disallowed because another open operation is in progress<br>1 if the open operation was successfully started<br><br>* serial, see ps3000aOpenUnit |
| **Returns** | PICO_OK<br>PICO_OPEN_OPERATION_IN_PROGRESS<br>PICO_OPERATION_FAILED |

# 4.38 ps3000aOpenUnitProgress

```
PICO_STATUS ps3000aOpenUnitProgress
(
  int16_t * handle,
  int16_t * progressPercent,
  int16_t * complete
)
```

This function checks on the progress of a request made to ps3000aOpenUnitAsync to open a scope.

If the function returns PICO_POWER_SUPPLY_NOT_CONNECTED or PICO_USB3_0_DEVICE_NON_USB3_0_PORT, call ps3000aChangePowerSource to select a new power source.

| | |
|---|---|
| **Applicability** | Use after ps3000aOpenUnitAsync |
| **Arguments** | * handle, see ps3000aOpenUnit. This handle is valid only if the function returns PICO_OK.<br><br>* progressPercent, on exit, the percentage progress towards opening the scope. 100% implies that the open operation is complete.<br><br>* complete, set to 1 when the open operation has finished |
| **Returns** | PICO_OK<br>PICO_NULL_PARAMETER<br>PICO_OPERATION_FAILED<br>PICO_POWER_SUPPLY_NOT_CONNECTED<br>PICO_USB3_0_DEVICE_NON_USB3_0_PORT |

# 4.39 ps3000aPingUnit

```
PICO_STATUS ps3000aPingUnit
(
  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 `ps3000aOpenUnit` |
| **Returns** | `PICO_OK`<br>`PICO_INVALID_HANDLE`<br>`PICO_DRIVER_FUNCTION`<br>`PICO_BUSY`<br>`PICO_NOT_RESPONDING`<br>`PICO_POWER_SUPPLY_UNDERVOLTAGE`<br>`PICO_POWER_SUPPLY_NOT_CONNECTED`<br>`PICO_POWER_SUPPLY_CONNECTED`<br>`PICO_USB3_0_DEVICE_NON_USB3_0_PORT` |

# 4.40 ps3000aRunBlock

```
PICO_STATUS ps3000aRunBlock
(
  int16_t             handle,
  int32_t             noOfPreTriggerSamples,
  int32_t             noOfPostTriggerSamples,
  uint32_t            timebase,
  int16_t             oversample,
  int32_t           * timeIndisposedMs,
  uint32_t            segmentIndex,
  ps3000aBlockReady   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 `ps3000aOpenUnit`<br>`noOfPreTriggerSamples`, the number of samples to return before the trigger event. If no trigger has been set, then this argument is added to `noOfPostTriggerSamples` to give the maximum number of data points (samples) to collect.<br><br>`noOfPostTriggerSamples`, the number of samples to return after the trigger event. If no trigger event has been set, then this argument is added to `noOfPreTriggerSamples` to give the maximum number of data points to collect. If a trigger condition has been set, this specifies the number of data points to collect after a trigger has fired, and the number of samples to be collected is:<br><br>`noOfPreTriggerSamples + noOfPostTriggerSamples`<br><br>`timebase`, a number in the range 0 to $2^{32}-1$. See the guide to calculating timebase values. In ETS mode this argument is ignored and the driver chooses the timebase automatically.<br><br>`oversample`, not used<br><br>`* timeIndisposedMs`, on exit, the time, in milliseconds, that the scope will spend collecting samples. This does not include any auto trigger timeout. If this pointer is null, nothing will be written here.<br><br>`segmentIndex`, zero-based, specifies which memory segment to use<br><br>`lpReady`, a pointer to the `ps3000aBlockReady` callback function that the driver will call when the data has been collected. To use the `ps3000aIsReady` polling method instead of a callback function, set this pointer to NULL.<br><br>`* pParameter`, a void pointer that is passed to the `ps3000aBlockReady` callback function. The callback can use this pointer to return arbitrary data to the application. | |
| **Returns** | `PICO_OK` |

| | |
|---|---|
| | PICO_POWER_SUPPLY_CONNECTED<br>PICO_POWER_SUPPLY_NOT_CONNECTED<br>PICO_BUFFERS_NOT_SET (in overlapped mode)<br>PICO_INVALID_HANDLE<br>PICO_USER_CALLBACK<br>PICO_SEGMENT_OUT_OF_RANGE<br>PICO_INVALID_CHANNEL<br>PICO_INVALID_TRIGGER_CHANNEL<br>PICO_INVALID_CONDITION_CHANNEL<br>PICO_TOO_MANY_SAMPLES<br>PICO_INVALID_TIMEBASE<br>PICO_NOT_RESPONDING<br>PICO_CONFIG_FAIL<br>PICO_INVALID_PARAMETER<br>PICO_NOT_RESPONDING<br>PICO_TRIGGER_ERROR<br>PICO_DRIVER_FUNCTION<br>PICO_FW_FAIL<br>PICO_NOT_ENOUGH_SEGMENTS (in bulk mode)<br>PICO_PULSE_WIDTH_QUALIFIER<br>PICO_SEGMENT_OUT_OF_RANGE (in overlapped mode)<br>PICO_STARTINDEX_INVALID (in overlapped mode)<br>PICO_INVALID_SAMPLERATIO (in overlapped mode)<br>PICO_CONFIG_FAIL |

# 4.41 ps3000aRunStreaming

```
PICO_STATUS ps3000aRunStreaming
(
  int16_t             handle,
  uint32_t          * sampleInterval,
  PS3000A_TIME_UNITS  sampleIntervalTimeUnits,
  uint32_t            maxPreTriggerSamples,
  uint32_t            maxPostTriggerSamples,
  int16_t             autoStop,
  uint32_t            downSampleRatio,
  PS3000A_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 ps3000aGetStreamingLatestValues to retrieve the data. See Using streaming mode for a step-by-step guide to this process.

Whether a trigger is set or not, the total number of samples stored in the driver is always maxPreTriggerSamples + maxPostTriggerSamples. If autoStop is false, this becomes the maximum number of samples without downsampling.

| **Applicability** | Streaming mode |
|---|---|
| **Arguments** | |
| handle, device identifier returned by ps3000aOpenUnit<br>* sampleInterval, on entry, the requested time interval between samples, in units of sampleIntervalTimeUnits; on exit, the actual time interval used.<br><br>sampleIntervalTimeUnits, the unit of time used for sampleInterval. Use one of these enumerated types:<br>PS3000A_FS<br>PS3000A_PS<br>PS3000A_NS<br>PS3000A_US<br>PS3000A_MS<br>PS3000A_S<br><br>maxPreTriggerSamples, the maximum number of raw samples before a trigger event for each enabled channel.<br><br>maxPostTriggerSamples, the maximum number of raw samples after a trigger event for each enabled channel.<br><br>autoStop, a flag that specifies if the streaming should stop when all of maxPreTriggerSamples + maxPostTriggerSamples have been captured.<br><br>downSampleRatio,<br>downSampleRatioMode: see ps3000aGetValues<br><br>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 ps3000aSetDataBuffer. | |
| **Returns** | PICO_OK |

| | |
|---|---|
| | PICO_INVALID_HANDLE<br>PICO_ETS_MODE_SET<br>PICO_USER_CALLBACK<br>PICO_NULL_PARAMETER<br>PICO_INVALID_PARAMETER<br>PICO_STREAMING_FAILED<br>PICO_NOT_RESPONDING<br>PICO_POWER_SUPPLY_CONNECTED<br>PICO_POWER_SUPPLY_NOT_CONNECTED<br>PICO_TRIGGER_ERROR<br>PICO_INVALID_SAMPLE_INTERVAL<br>PICO_INVALID_BUFFER<br>PICO_DRIVER_FUNCTION<br>PICO_FW_FAIL<br>PICO_MEMORY |

# 4.42 ps3000aSetBandwidthFilter

```
PICO_STATUS ps3000aSetBandwidthFilter
(
  int16_t                    handle,
  PS3000A_CHANNEL            channel,
  PS3000A_BANDWIDTH_LIMITER  bandwidth
)
```

This function sets the bandwidth limiter for a specified channel.

| | |
|---|---|
| **Applicability** | All modes. PicoScope 3400, 3000D, and 3000D MSO scopes only. |
| **Arguments** | `handle`, device identifier returned by `ps3000aOpenUnit`<br>`channel`, the channel to be configured. Use one of the following enumerated types:<br>`PS3000A_CHANNEL_A`: Channel A input<br>`PS3000A_CHANNEL_B`: Channel B input<br>`PS3000A_CHANNEL_C`: Channel C input (if present)<br>`PS3000A_CHANNEL_D`: Channel D input (if present)<br><br>`bandwidth`, either one of these values:<br>`PS3000A_BW_FULL`<br>`PS3000A_BW_20MHZ` |
| **Returns** | `PICO_OK`<br>`PICO_INVALID_HANDLE`<br>`PICO_INVALID_CHANNEL`<br>`PICO_INVALID_BANDWIDTH` |

# 4.43 ps3000aSetChannel

```
PICO_STATUS ps3000aSetChannel
(
  int16_t            handle,
  PS3000A_CHANNEL    channel,
  int16_t            enabled,
  PS3000A_COUPLING   type,
  PS3000A_RANGE      range,
  float              analogueOffset
)
```

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

| | |
|---|---|
| **Applicability** | All modes |
| **Arguments** | `handle`, device identifier returned by `ps3000aOpenUnit`<br>`channel`, the channel to be configured. Use one of the following enumerated types:<br>`PS3000A_CHANNEL_A`: Channel A input<br>`PS3000A_CHANNEL_B`: Channel B input<br>`PS3000A_CHANNEL_C`: Channel C input<br>`PS3000A_CHANNEL_D`: Channel D input<br><br>`enabled`, whether or not to enable the channel (`TRUE` or `FALSE`)<br><br>`type`, the impedance and coupling type. The values are:<br>`PS3000A_AC`: 1 megohm impedance, AC coupling. The channel accepts input frequencies from about 1 hertz up to its maximum –3 dB analog bandwidth.<br>`PS3000A_DC`: 1 megohm impedance, DC coupling. The scope accepts all input frequencies from zero (DC) up to its maximum –3 dB analog bandwidth.<br><br>`range`, the input voltage range, one of these enumerated types:<br>`PS3000A_50MV`: ±50 mV<br>`PS3000A_100MV`: ±100 mV<br>`PS3000A_200MV`: ±200 mV<br>`PS3000A_500MV`: ±500 mV<br>`PS3000A_1V`: ±1 V<br>`PS3000A_2V`: ±2 V<br>`PS3000A_5V`: ±5 V<br>`PS3000A_10V`: ±10 V<br>`PS3000A_20V`: ±20 V<br><br>`analogueOffset`, a voltage to add to the input channel before digitization. The allowable range of offsets depends on the input range selected for the channel, as obtained from `ps3000aGetAnalogueOffset`. |
| **Returns** | `PICO_OK`<br>`PICO_USER_CALLBACK`<br>`PICO_INVALID_HANDLE`<br>`PICO_INVALID_CHANNEL`<br>`PICO_INVALID_VOLTAGE_RANGE`<br>`PICO_INVALID_COUPLING`<br>`PICO_INVALID_ANALOGUE_OFFSET`<br>`PICO_DRIVER_FUNCTION` |

# 4.44 ps3000aSetDataBuffer

```
PICO_STATUS ps3000aSetDataBuffer
(
  int16_t             handle,
  PS3000A_CHANNEL     channel,
  int16_t           * buffer,
  int32_t             bufferLth,
  uint32_t            segmentIndex,
  PS3000A_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 `GetValues` functions. The function allows you to specify only a single buffer, so for aggregation mode, which requires two buffers, you need to call `ps3000aSetDataBuffers` 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 `ps3000aOpenUnit`<br>`channel`, the channel you want to use with the buffer. Use one of these enumerated types:<br>`PS3000A_CHANNEL_A`<br>`PS3000A_CHANNEL_B`<br>`PS3000A_CHANNEL_C`<br>`PS3000A_CHANNEL_D`<br><br>To set the buffer for a digital port, use one of these enumerated types:<br>`PS3000A_DIGITAL_PORT0 = 0x80`<br>`PS3000A_DIGITAL_PORT1 = 0x81`<br><br>`* buffer`, the location of the buffer<br><br>`bufferLth`, the size of the `buffer` array<br><br>`segmentIndex`, the number of the memory segment to be used<br><br>`mode`, the downsampling mode. See `ps3000aGetValues` for the available modes, but note that a single call to `ps3000aSetDataBuffer` can only associate one buffer with one downsampling mode. If you intend to call `ps3000aGetValues` with more than one downsampling mode activated, then you must call `ps3000aSetDataBuffer` several times to associate a separate buffer with each downsampling mode. |
| **Returns** | `PICO_OK`<br>`PICO_INVALID_HANDLE`<br>`PICO_INVALID_CHANNEL`<br>`PICO_RATIO_MODE_NOT_SUPPORTED`<br>`PICO_SEGMENT_OUT_OF_RANGE`<br>`PICO_DRIVER_FUNCTION`<br>`PICO_INVALID_PARAMETER` |

# 4.45 ps3000aSetDataBuffers

```
PICO_STATUS ps3000aSetDataBuffers
(
  int16_t             handle,
  PS3000A_CHANNEL     channel,
  int16_t           * bufferMax,
  int16_t           * bufferMin,
  int32_t             bufferLth,
  uint32_t            segmentIndex,
  PS3000A_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, then you can optionally use ps3000aSetDataBuffer instead.

| | |
|---|---|
| **Applicability** | Block and streaming modes with aggregation. |
| **Arguments** | handle, device identifier returned by ps3000aOpenUnit<br>channel, the channel for which you want to set the buffers. Use one of these constants:<br>PS3000A_CHANNEL_A<br>PS3000A_CHANNEL_B<br>PS3000A_CHANNEL_C<br>PS3000A_CHANNEL_D<br><br>To set the buffer for a digital port, use one of these enumerated types:<br>PS3000A_DIGITAL_PORT0 = 0x80<br>PS3000A_DIGITAL_PORT1 = 0x81<br><br>* bufferMax, a buffer to receive the maximum data values in aggregation mode, or the non-aggregated values otherwise<br><br>* bufferMin, a buffer to receive the minimum aggregated data values. Not used in other downsampling modes.<br><br>bufferLth, the size of the bufferMax and bufferMin arrays<br><br>segmentIndex, the number of the memory segment to be used<br><br>mode, see ps3000aGetValues |
| **Returns** | PICO_OK<br>PICO_INVALID_HANDLE<br>PICO_INVALID_CHANNEL<br>PICO_RATIO_MODE_NOT_SUPPORTED<br>PICO_SEGMENT_OUT_OF_RANGE<br>PICO_DRIVER_FUNCTION<br>PICO_INVALID_PARAMETER |

# 4.46 ps3000aSetDigitalPort

```
PICO_STATUS ps3000aSetDigitalPort
(
  int16_t                handle,
  PS3000A_DIGITAL_PORT   port,
  int16_t                enabled,
  int16_t                logiclevel
)
```

This function is used to enable the digital port and set the logic level (the voltage at which the state transitions from 0 to 1).

| | |
|---|---|
| **Applicability** | Block and streaming modes with aggregation.<br>MSOs only. |
| **Arguments** | handle, device identifier returned by ps3000aOpenUnit<br>port, identifies the port for digital data:<br>PS3000A_DIGITAL_PORT0 = 0x80 (digital channels 0–7)<br>PS3000A_DIGITAL_PORT1 = 0x81 (digital channels 8–15)<br><br>enabled, whether or not to enable the channel. The values are:<br>TRUE: enable<br>FALSE: do not enable<br><br>logiclevel, the voltage at which the state transitions between 0 and 1. Range: –32767 (–5 V) to 32767 (5 V). |
| **Returns** | PICO_OK<br>PICO_INVALID_HANDLE<br>PICO_INVALID_CHANNEL<br>PICO_RATIO_MODE_NOT_SUPPORTED<br>PICO_SEGMENT_OUT_OF_RANGE<br>PICO_DRIVER_FUNCTION<br>PICO_INVALID_PARAMETER |

# 4.47 ps3000aSetEts

```
PICO_STATUS ps3000aSetEts
(
  int16_t            handle,
  PS3000A_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 |
|---|---|
| **Arguments** | |
| `handle`, device identifier returned by `ps3000aOpenUnit`<br>`mode`, the ETS mode. Use one of these values:<br>`PS3000A_ETS_OFF` - disables ETS<br>`PS3000A_ETS_FAST` - enables ETS and provides `etsCycles` of data, which may contain data from previously returned cycles<br>`PS3000A_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.<br><br>`etsCycles`, the number of cycles to store: the driver then selects `etsInterleave` cycles to give the most uniform spread of samples. Range: between two and five times the value of `etsInterleave`, and not more than the `etsCycles` value returned by `ps3000aGetMaxEtsValues`.<br><br>`etsInterleave`, the number of waveforms to combine into a single ETS capture. The maximum allowed value for the selected device is returned by `ps3000aGetMaxEtsValues` in the `etsInterleave` argument.<br><br>`* 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, the effective sample time in ETS mode is 400 ps. | |
| **Returns** | `PICO_OK`<br>`PICO_USER_CALLBACK`<br>`PICO_INVALID_HANDLE`<br>`PICO_INVALID_PARAMETER`<br>`PICO_DRIVER_FUNCTION` |

# 4.48 ps3000aSetEtsTimeBuffer

```
PICO_STATUS ps3000aSetEtsTimeBuffer
(
  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.<br><br>If your programming language does not support 64-bit data, use the 32-bit version ps3000aSetEtsTimeBuffers instead. |
| **Arguments** | handle, device identifier returned by ps3000aOpenUnit<br>* buffer, an array of 64-bit words, each representing the time in femtoseconds ($10^{-15}$ seconds) at which the sample was captured<br><br>bufferLth, the size of the buffer array |
| **Returns** | PICO_OK<br>PICO_INVALID_HANDLE<br>PICO_NULL_PARAMETER<br>PICO_DRIVER_FUNCTION |

# 4.49 ps3000aSetEtsTimeBuffers

```
PICO_STATUS ps3000aSetEtsTimeBuffers
(
  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.<br><br>If your programming language supports 64-bit data then you can use ps3000aSetEtsTimeBuffer instead. |
| **Arguments** | handle, device identifier returned by ps3000aOpenUnit<br>* timeUpper, an array of 32-bit words, each representing the upper 32 bits of the time in femtoseconds ($10^{-15}$ seconds) at which the sample was captured<br><br>* timeLower, an array of 32-bit words, each representing the lower 32 bits of the time in femtoseconds ($10^{-15}$ seconds) at which the sample was captured<br><br>bufferLth, the size of the timeUpper and timeLower arrays |
| **Returns** | PICO_OK<br>PICO_INVALID_HANDLE<br>PICO_NULL_PARAMETER<br>PICO_DRIVER_FUNCTION |

# 4.50 ps3000aSetNoOfCaptures

```
PICO_STATUS ps3000aSetNoOfCaptures
(
  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 `ps3000aOpenUnit`<br>`nCaptures`, the number of waveforms to capture in one run |
| **Returns** | `PICO_OK`<br>`PICO_INVALID_HANDLE`<br>`PICO_INVALID_PARAMETER`<br>`PICO_DRIVER_FUNCTION` |

# 4.51 ps3000aSetPulseWidthDigitalPortProperties

```
PICO_STATUS ps3000aSetPulseWidthDigitalPortProperties
(
  int16_t                                handle,
  PS3000A_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 PS3000A_DIGITAL_CHANNEL_DIRECTIONS the driver assumes the digital channel's pulse-width trigger direction is PS3000A_DIGITAL_DONT_CARE.

| | |
|---|---|
| **Applicability** | All modes.<br>PicoScope 3000D MSO models only. |
| **Arguments** | handle, device identifier returned by ps3000aOpenUnit<br>* directions, a pointer to an array of PS3000A_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 PS3000A_DIGITAL_DONT_CARE.<br><br>nDirections, the number of digital channel directions being passed to the driver |
| **Returns** | PICO_OK<br>PICO_INVALID_HANDLE<br>PICO_DRIVER_FUNCTION<br>PICO_INVALID_DIGITAL_CHANNEL<br>PICO_INVALID_DIGITAL_TRIGGER_DIRECTION |

# 4.52 ps3000aSetPulseWidthQualifier

```
PICO_STATUS ps3000aSetPulseWidthQualifier
(
  int16_t                      handle,
  PS3000A_PWQ_CONDITIONS     * conditions,
  int16_t                      nConditions,
  PS3000A_THRESHOLD_DIRECTION  direction,
  uint32_t                     lower,
  uint32_t                     upper,
  PS3000A_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 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.

| **Applicability** | All modes |
|---|---|
| **Arguments** | |
| `handle`, device identifier returned by `ps3000aOpenUnit`<br>`* conditions`, an array of `PS3000A_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. If `conditions` is `NULL` then the pulse-width qualifier is not used.<br><br>`nConditions`, the number of elements in the `conditions` array. If `nConditions` is zero then the pulse-width qualifier is not used.<br>Range: 0 to `PS3000A_MAX_PULSE_WIDTH_QUALIFIER_COUNT`.<br><br>`direction`, the direction of the signal required for the pulse width trigger to fire. See `PS3000A_THRESHOLD_DIRECTION constants` for the list of possible values. Each channel of the oscilloscope (except the EXT input) has two thresholds for each direction—for example, `PS3000A_RISING` and `PS3000A_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 `PS3000A_RISING` as the `direction` argument for both `ps3000aSetTriggerConditions` and `ps3000aSetPulseWidthQualifier` at the same time. There is no such restriction when using window triggers.<br><br>`lower`, the lower limit of the pulse-width counter, measured in samples<br><br>`upper`, the upper limit of the pulse-width counter, measured in samples. This parameter is used only when the type is set to `PS3000A_PW_TYPE_IN_RANGE` or `PS3000A_PW_TYPE_OUT_OF_RANGE`. | |
| **Arguments** | `type`, the pulse-width type, one of these constants:<br>`PS3000A_PW_TYPE_NONE`: do not use the pulse width qualifier<br>`PS3000A_PW_TYPE_LESS_THAN`: pulse width less than `lower`<br>`PS3000A_PW_TYPE_GREATER_THAN`: pulse width greater than `lower`<br>`PS3000A_PW_TYPE_IN_RANGE`: pulse width between `lower` and `upper`<br>`PS3000A_PW_TYPE_OUT_OF_RANGE`: pulse width not between `lower` and `upper` |
| **Returns** | `PICO_OK`<br>`PICO_INVALID_HANDLE`<br>`PICO_USER_CALLBACK` |

| | |
|---|---|
| | PICO_CONDITIONS<br>PICO_PULSE_WIDTH_QUALIFIER<br>PICO_DRIVER_FUNCTION |

*Note: using this function the driver will convert the PS3000A_PWQ_CONDITIONS into a PS3000A_PWQ_CONDITIONS_V2 and will set the condition for digital to PS3000A_DIGITAL_DONT_CARE.

## 4.52.1 PS3000A_PWQ_CONDITIONS structure

A structure of this type is passed to ps3000aSetPulseWidthQualifier in the conditions argument to specify the trigger conditions. It is defined as follows:

```
typedef struct tPS3000APwqConditions
{
  PS3000A_TRIGGER_STATE channelA;
  PS3000A_TRIGGER_STATE channelB;
  PS3000A_TRIGGER_STATE channelC;
  PS3000A_TRIGGER_STATE channelD;
  PS3000A_TRIGGER_STATE external;
  PS3000A_TRIGGER_STATE aux;
} PS3000A_PWQ_CONDITIONS
```

Each structure is the logical AND of the states of the scope's inputs. The ps3000aSetPulseWidthQualifier 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.

| | |
|---|---|
| **Applicability** | All models* |
| **Elements** | channelA, channelB, channelC**, channelD**, external, the type of condition that should be applied to each channel. Use these constants: -<br>PS3000A_CONDITION_DONT_CARE<br>PS3000A_CONDITION_TRUE<br>PS3000A_CONDITION_FALSE<br><br>The channels that are set to PS3000A_CONDITION_TRUE or PS3000A_CONDITION_FALSE must all meet their conditions simultaneously to produce a trigger. Channels set to PS3000A_CONDITION_DONT_CARE are ignored.<br><br>aux, not used |

*Note: using this function the driver will convert the PS3000A_PWQ_CONDITIONS into a PS3000A_PWQ_CONDITIONS_V2 and will set the condition for digital to PS3000A_DIGITAL_DONT_CARE.

**Note: applicable to 4-channel oscilloscopes only.

# 4.53 ps3000aSetPulseWidthQualifierV2

```
PICO_STATUS ps3000aSetPulseWidthQualifierV2
(
  int16_t                        handle,
  PS3000A_PWQ_CONDITIONS_V2    * conditions,
  int16_t                        nConditions,
  PS3000A_THRESHOLD_DIRECTION    direction,
  uint32_t                       lower,
  uint32_t                       upper,
  PS3000A_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 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.

| **Applicability** | All modes |
|---|---|
| **Arguments** | |
| `handle`, device identifier returned by `ps3000aOpenUnit`<br>`* conditions`, an array of `PS3000A_PWQ_CONDITIONS_V2` 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. If `conditions` is `NULL` then the pulse-width qualifier is not used.<br><br>`nConditions`, the number of elements in the `conditions` array. If `nConditions` is zero then the pulse-width qualifier is not used.<br>Range: 0 to `PS3000A_MAX_PULSE_WIDTH_QUALIFIER_COUNT`.<br><br>`direction`, the direction of the signal required for the pulse width trigger to fire. See `PS3000A_THRESHOLD_DIRECTION constants` for the list of possible values. Each channel of the oscilloscope (except the EXT input) has two thresholds for each direction—for example, `PS3000A_RISING` and `PS3000A_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 `PS3000A_RISING` as the `direction` argument for both `ps3000aSetTriggerConditionsV2` and `ps3000aSetPulseWidthQualifierV2` at the same time. There is no such restriction when using window triggers.<br><br>`lower`, the lower limit of the pulse-width counter, measured in samples<br><br>`upper`, the upper limit of the pulse-width counter, measured in samples. This parameter is used only when the type is set to `PS3000A_PW_TYPE_IN_RANGE` or `PS3000A_PW_TYPE_OUT_OF_RANGE`. | |
| **Arguments** | `type`, the pulse-width type, one of these constants:<br>`PS3000A_PW_TYPE_NONE`: do not use the pulse width qualifier<br>`PS3000A_PW_TYPE_LESS_THAN`: pulse width less than `lower`<br>`PS3000A_PW_TYPE_GREATER_THAN`: pulse width greater than `lower`<br>`PS3000A_PW_TYPE_IN_RANGE`: pulse width between `lower` and `upper`<br>`PS3000A_PW_TYPE_OUT_OF_RANGE`: pulse width not between `lower` and `upper` |
| **Returns** | `PICO_OK` |

| | |
|---|---|
| | PICO_INVALID_HANDLE<br>PICO_USER_CALLBACK<br>PICO_CONDITIONS<br>PICO_PULSE_WIDTH_QUALIFIER<br>PICO_DRIVER_FUNCTION |

## 4.53.1 PS3000A_PWQ_CONDITIONS_V2 structure

A structure of this type is passed to ps3000aSetPulseWidthQualifierV2 in the conditions argument to specify the trigger conditions. It is defined as follows:

```
typedef struct tPS3000APwqConditionsV2
{
  PS3000A_TRIGGER_STATE channelA;
  PS3000A_TRIGGER_STATE channelB;
  PS3000A_TRIGGER_STATE channelC;
  PS3000A_TRIGGER_STATE channelD;
  PS3000A_TRIGGER_STATE external;
  PS3000A_TRIGGER_STATE aux;
  PS3000A_TRIGGER_STATE digital;
} PS3000A_PWQ_CONDITIONS_V2
```

Each structure is the logical AND of the states of the scope's inputs. The ps3000aSetPulseWidthQualifierV2 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.

| | |
|---|---|
| **Applicability** | All models |
| **Elements** | channelA, channelB, channelC*, channelD*, external, the type of condition that should be applied to each channel. Use these constants: -<br>PS3000A_CONDITION_DONT_CARE<br>PS3000A_CONDITION_TRUE<br>PS3000A_CONDITION_FALSE<br><br>The channels that are set to PS3000A_CONDITION_TRUE or PS3000A_CONDITION_FALSE must all meet their conditions simultaneously to produce a trigger. Channels set to PS3000A_CONDITION_DONT_CARE are ignored.<br><br>aux, not used |

*Note: applicable to 4-channel analog devices only.

# 4.54 ps3000aSetSigGenArbitrary

```
PICO_STATUS ps3000aSetSigGenArbitrary
(
  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,
  PS3000A_SWEEP_TYPE         sweepType,
  PS3000A_EXTRA_OPERATIONS   operation,
  PS3000A_INDEX_MODE         indexMode,
  uint32_t                   shots,
  uint32_t                   sweeps,
  PS3000A_SIGGEN_TRIG_TYPE   triggerType,
  PS3000A_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 `ps3000aSigGenFrequencyToPhase`. Alternatively, see Calculating deltaPhase below for more information on how to calculate these values.

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

| **Applicability** | All modes. All models with AWG. |
|---|---|
| **Arguments** | |
| `handle`, device identifier returned by `ps3000aOpenUnit`<br>`offsetVoltage`, the voltage offset, in microvolts, to be applied to the waveform<br><br>`pkToPk`, the peak-to-peak voltage, in microvolts, of the waveform signal. Note that 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.<br><br>`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 `ps3000aSigGenFrequencyToPhase`. | |

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 units of *dacPeriod*, between successive additions of deltaPhaseIncrement to the delta phase accumulator. This determines the rate at which the generator sweeps the output frequency.

Minimum value: PS3000A_MIN_DWELL_COUNT

* arbitraryWaveform, a buffer that holds the waveform pattern as a set of samples equally spaced in time. If pkToPk is set to its maximum (4 V) and offsetVoltage is set to 0 V:
a sample of minArbitraryWaveformValue corresponds to - 2 V
a sample of maxArbitraryWaveformValue corresponds to +2 V
where minArbitraryWaveformValue and maxArbitraryWaveformValue are the values returned by ps3000aSigGenArbitraryMinMaxValues.

arbitraryWaveformSize, the size of the arbitrary waveform buffer, in samples, in the range:
[minArbitraryWaveformSize, maxArbitraryWaveformSize]
where minArbitraryWaveformSize and maxArbitraryWaveformSize are the values returned by ps3000aSigGenArbitraryMinMaxValues.

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 enumerated types: -
PS3000A_UP
PS3000A_DOWN
PS3000A_UPDOWN
PS3000A_DOWNUP

operation, the type of waveform to be produced, specified by one of the following enumerated types:
PS3000A_ES_OFF, normal signal generator operation specified by wavetype.
PS3000A_WHITENOISE, the signal generator produces white noise and ignores all settings except pkToPk and offsetVoltage.
PS3000A_PRBS, produces a pseudorandom random binary sequence with a bit rate specified by the start and stop frequency.

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:
PS3000A_SINGLE
PS3000A_DUAL

shots,
sweeps,
triggerType,
triggerSource,
extInThreshold: see ps3000aSigGenBuiltIn

| **Returns** | PICO_OK<br>PICO_AWG_NOT_SUPPORTED<br>PICO_POWER_SUPPLY_CONNECTED<br>PICO_POWER_SUPPLY_NOT_CONNECTED<br>PICO_BUSY<br>PICO_INVALID_HANDLE |
|---|---|

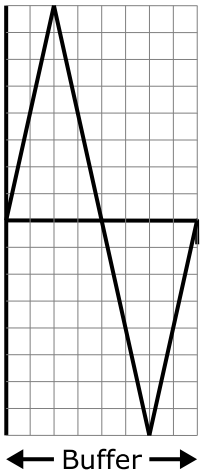
  

| | |
|---|---|
| | PICO_SIG_GEN_PARAM<br>PICO_SHOTS_SWEEPS_WARNING<br>PICO_NOT_RESPONDING<br>PICO_WARNING_EXT_THRESHOLD_CONFLICT<br>PICO_NO_SIGNAL_GENERATOR<br>PICO_SIGGEN_OFFSET_VOLTAGE<br>PICO_SIGGEN_PK_TO_PK<br>PICO_SIGGEN_OUTPUT_OVER_VOLTAGE<br>PICO_DRIVER_FUNCTION<br>PICO_SIGGEN_WAVEFORM_SETUP_FAILED |

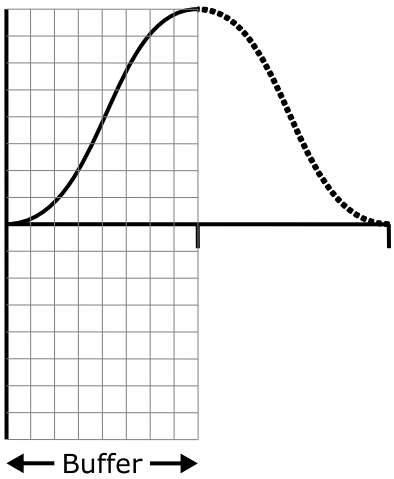
## 4.54.1 AWG index modes

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.



## 4.54.2 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 *dacPeriod* (*1 / dacFrequency*). If the *deltaPhase* is constant, the generator produces a waveform at a constant frequency that can be calculated as follows:

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

where:

| | |
|---|---|
| *outputFrequency* | = repetition rate of the complete arbitrary waveform |
| *dacFrequency* | = DAC update rate for specific oscilloscope model (see data sheet) |
| *deltaPhase* | = calculated from `startDeltaPhase` and `deltaPhaseIncrement` (we recommend that you use `ps3000aSigGenFrequencyToPhase` to calculate *deltaPhase*) |
| *phaseAccumulatorSize* | = $2^{32}$ for all models |
| *awgBufferSize* | = AWG buffer size for specific oscilloscope model (see data sheet) |
| *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`.

# 4.55 ps3000aSetSigGenBuiltIn

```
PICO_STATUS ps3000aSetSigGenBuiltIn
(
  int16_t                    handle,
  int32_t                    offsetVoltage,
  uint32_t                   pkToPk,
  PS3000A_WAVE_TYPE          waveType,
  float                      startFrequency,
  float                      stopFrequency,
  float                      increment,
  float                      dwellTime,
  PS3000A_SWEEP_TYPE         sweepType,
  PS3000A_EXTRA_OPERATIONS   operation,
  uint32_t                   shots,
  uint32_t                   sweeps,
  PS3000A_SIGGEN_TRIG_TYPE   triggerType,
  PS3000A_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.

| **Applicability** | All models |
|---|---|

**Arguments**

`handle`, device identifier returned by `ps3000aOpenUnit`

`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 that 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.

| | |
|---|---|
| `PS3000A_SINE` | sine wave |
| `PS3000A_SQUARE` | square wave |
| `PS3000A_TRIANGLE` | triangle wave |
| `PS3000A_DC_VOLTAGE` | DC voltage |

The following `waveTypes` apply to B and MSO models only.

| | |
|---|---|
| `PS3000A_RAMP_UP` | rising sawtooth |
| `PS3000A_RAMP_DOWN` | falling sawtooth |
| `PS3000A_SINC` | sin (x)/x |
| `PS3000A_GAUSSIAN` | Gaussian |
| `PS3000A_HALF_SINE` | half (full-wave rectified) sine |

`startFrequency`, the frequency that the signal generator will initially produce. For allowable values see `PS3000A_SINE_MAX_FREQUENCY` and related values.

`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:

PS3000A_UP
PS3000A_DOWN
PS3000A_UPDOWN
PS3000A_DOWNUP

operation, the type of waveform to be produced, specified by one of the following enumerated types (MSO and B models only):

PS3000A_ES_OFF, normal signal generator operation specified by wavetype.
PS3000A_WHITENOISE, the signal generator produces white noise and ignores all settings except pkToPk and offsetVoltage.
PS3000A_PRBS, produces a pseudorandom binary sequence with bit rate specified by the start and stop frequencies.

shots,

0: sweep the frequency as specified by sweeps
1...PS3000A_MAX_SWEEPS_SHOTS: the number of cycles of the waveform to be produced after a trigger event. sweeps must be zero.
PS3000A_SHOT_SWEEP_TRIGGER_CONTINUOUS_RUN: start and run continuously after trigger occurs

sweeps,

0: produce number of cycles specified by shots
1..PS3000A_MAX_SWEEPS_SHOTS: the number of times to sweep the frequency after a trigger event, according to sweepType. shots must be zero.
PS3000A_SHOT_SWEEP_TRIGGER_CONTINUOUS_RUN: start a sweep and continue after trigger occurs

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

| | |
|---|---|
| PS3000A_SIGGEN_RISING | trigger on rising edge |
| PS3000A_SIGGEN_FALLING | trigger on falling edge |
| PS3000A_SIGGEN_GATE_HIGH | run while trigger is high |
| PS3000A_SIGGEN_GATE_LOW | run while trigger is low |

triggerSource, the source that will trigger the signal generator:

| | |
|---|---|
| PS3000A_SIGGEN_NONE | run without waiting for trigger |
| PS3000A_SIGGEN_SCOPE_TRIG | use scope trigger |
| PS3000A_SIGGEN_EXT_IN | use EXT input |
| PS3000A_SIGGEN_SOFT_TRIG | wait for software trigger provided by ps3000aSigGenSoftwareControl |
| PS3000A_SIGGEN_TRIGGER_RAW | reserved |

If a trigger source other than P3000A_SIGGEN_NONE is specified, then either shots or sweeps, but not both, must be non-zero.

extInThreshold, sets trigger level for external trigger (see Voltage ranges)

**Returns**

PICO_OK
PICO_BUSY
PICO_POWER_SUPPLY_CONNECTED
PICO_POWER_SUPPLY_NOT_CONNECTED
PICO_INVALID_HANDLE
PICO_SIG_GEN_PARAM
PICO_SHOTS_SWEEPS_WARNING

| | |
|---|---|
| | PICO_NOT_RESPONDING<br>PICO_WARNING_AUX_OUTPUT_CONFLICT<br>PICO_WARNING_EXT_THRESHOLD_CONFLICT<br>PICO_NO_SIGNAL_GENERATOR<br>PICO_SIGGEN_OFFSET_VOLTAGE<br>PICO_SIGGEN_PK_TO_PK<br>PICO_SIGGEN_OUTPUT_OVER_VOLTAGE<br>PICO_DRIVER_FUNCTION<br>PICO_SIGGEN_WAVEFORM_SETUP_FAILED<br>PICO_NOT_RESPONDING |

# 4.56 ps3000aSetSigGenBuiltInV2

```
PICO_STATUS ps3000aSetSigGenBuiltInV2
(
  int16_t                    handle,
  int32_t                    offsetVoltage,
  uint32_t                   pkToPk,
  PS3000A_WAVE_TYPE          waveType,
  double                     startFrequency,
  double                     stopFrequency,
  double                     increment,
  double                     dwellTime,
  PS3000A_SWEEP_TYPE         sweepType,
  PS3000A_EXTRA_OPERATIONS   operation,
  uint32_t                   shots,
  uint32_t                   sweeps,
  PS3000A_SIGGEN_TRIG_TYPE   triggerType,
  PS3000A_SIGGEN_TRIG_SOURCE triggerSource,
  int16_t                    extInThreshold
)
```

This function is an upgraded version of ps3000aSetSigGenBuiltIn with double-precision frequency arguments for more precise control at low frequencies.

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

| | |
|---|---|
| **Applicability** | All models |
| **Arguments** | See ps3000aSetSigGenBuiltIn |
| **Returns** | See ps3000aSetSigGenBuiltIn |

# 4.57 ps3000aSetSigGenPropertiesArbitrary

```
PICO_STATUS ps3000aSetSigGenPropertiesArbitrary
(
  int16_t                    handle,
  uint32_t                   startDeltaPhase,
  uint32_t                   stopDeltaPhase,
  uint32_t                   deltaPhaseIncrement,
  uint32_t                   dwellCount,
  PS3000A_SWEEP_TYPE         sweepType,
  uint32_t                   shots,
  uint32_t                   sweeps,
  PS3000A_SIGGEN_TRIG_TYPE   triggerType,
  PS3000A_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 ps3000aSetSigGenArbitrary |
| **Returns** | 0: if successful.<br>Error code: if failed |

# 4.58 ps3000aSetSigGenPropertiesBuiltIn

```
PICO_STATUS ps3000aSetSigGenPropertiesBuiltIn
(
  int16_t                    handle,
  double                     startFrequency,
  double                     stopFrequency,
  double                     increment,
  double                     dwellTime,
  PS3000A_SWEEP_TYPE         sweepType,
  uint32_t                   shots,
  uint32_t                   sweeps,
  PS3000A_SIGGEN_TRIG_TYPE   triggerType,
  PS3000A_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 ps3000aSetSigGenBuiltIn |
| **Returns** | 0: if successful.<br>Error code: if failed |

# 4.59 ps3000aSetSimpleTrigger

```
PICO_STATUS ps3000aSetSimpleTrigger
(
  int16_t                       handle,
  int16_t                       enable,
  PS3000A_CHANNEL               source,
  int16_t                       threshold,
  PS3000A_THRESHOLD_DIRECTION   direction,
  uint32_t                      delay,
  int16_t                       autoTrigger_ms
)
```

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

| | |
|---|---|
| **Applicability** | All modes |
| **Arguments** | `handle`, device identifier returned by `ps3000aOpenUnit`<br>`enable`, zero to disable the trigger; any other value to set the trigger<br><br>`source`, the channel on which to trigger<br><br>`threshold`, the ADC count at which the trigger will fire<br><br>`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`.<br><br>`delay`, the time between the trigger occurring and the first sample. For example, if `delay = 100`, the scope would wait 100 sample periods before sampling. At a timebase of 500 MS/s, or 2 ns per sample, the total delay would then be 100 x 2 ns = 200 ns. Range: 0 to `MAX_DELAY_COUNT`.<br><br>`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`<br>`PICO_INVALID_CHANNEL`<br>`PICO_INVALID_PARAMETER`<br>`PICO_MEMORY`<br>`PICO_CONDITIONS`<br>`PICO_INVALID_HANDLE`<br>`PICO_USER_CALLBACK`<br>`PICO_DRIVER_FUNCTION` |

# 4.60 ps3000aSetTriggerChannelConditions

```
PICO_STATUS ps3000aSetTriggerChannelConditions
(
  int16_t                      handle,
  PS3000A_TRIGGER_CONDITIONS * conditions,
  int16_t                      nConditions
)
```

This function sets up trigger conditions on the scope's inputs. The trigger is defined by one or more PS3000A_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.

If complex triggering is not required, use ps3000aSetSimpleTrigger.

| | |
|---|---|
| **Applicability** | All modes |
| **Arguments** | handle, device identifier returned by ps3000aOpenUnit<br>* conditions, an array of PS3000A_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.<br><br>nConditions, the number of elements in the conditions array. If nConditions is zero then triggering is switched off. |
| **Returns** | PICO_OK<br>PICO_INVALID_HANDLE<br>PICO_USER_CALLBACK<br>PICO_CONDITIONS<br>PICO_MEMORY<br>PICO_DRIVER_FUNCTION |

*Note: using this function the driver will convert the PS3000A_TRIGGER_CONDITIONS into a PS3000A_TRIGGER_CONDITIONS_V2 and will set the condition for digital to PS3000A_DIGITAL_DONT_CARE.

## 4.60.1 PS3000A_TRIGGER_CONDITIONS structure

A structure of this type is passed to ps3000aSetTriggerChannelConditions in the conditions argument to specify the trigger conditions, and is defined as follows: -

```
typedef struct tPS3000ATriggerConditions
{
  PS3000A_TRIGGER_STATE channelA;
  PS3000A_TRIGGER_STATE channelB;
  PS3000A_TRIGGER_STATE channelC;
  PS3000A_TRIGGER_STATE channelD;
  PS3000A_TRIGGER_STATE external;
  PS3000A_TRIGGER_STATE aux;
  PS3000A_TRIGGER_STATE pulseWidthQualifier;
} PS3000A_TRIGGER_CONDITIONS
```

Each structure is the logical AND of the states of the scope's inputs. The ps3000aSetTriggerChannelConditions 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 structure is byte-aligned. In C++, for example, you should specify this using the `#pragma pack ()` instruction.

| | |
|---|---|
| **Elements** | `channelA, channelB, channelC, channelD, external, pulseWidthQualifier,` the type of condition that should be applied to each channel. Use these constants:<br>`PS3000A_CONDITION_DONT_CARE`<br>`PS3000A_CONDITION_TRUE`<br>`PS3000A_CONDITION_FALSE`<br><br>The channels that are set to `PS3000A_CONDITION_TRUE` or `PS3000A_CONDITION_FALSE` must all meet their conditions simultaneously to produce a trigger. Channels set to `PS3000A_CONDITION_DONT_CARE` are ignored.<br><br>`aux`, not used |

# 4.61 ps3000aSetTriggerChannelConditionsV2

```
PICO_STATUS ps3000aSetTriggerChannelConditionsV2
(
  int16_t                          handle,
  PS3000A_TRIGGER_CONDITIONS_V2 * conditions,
  int16_t                          nConditions
)
```

This function sets up trigger conditions on the scope's inputs. The trigger is defined by one or more PS3000A_TRIGGER_CONDITIONS_V2 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.

If complex triggering is not required, use ps3000aSetSimpleTrigger.

| | |
|---|---|
| **Applicability** | All modes |
| **Arguments** | handle, device identifier returned by ps3000aOpenUnit<br>* conditions, an array of PS3000A_TRIGGER_CONDITIONS_V2 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.<br><br>nConditions, the number of elements in the conditions array. If nConditions is zero then triggering is switched off. |
| **Returns** | PICO_OK<br>PICO_INVALID_HANDLE<br>PICO_USER_CALLBACK<br>PICO_CONDITIONS<br>PICO_MEMORY<br>PICO_DRIVER_FUNCTION |

## 4.61.1 PS3000A_TRIGGER_CONDITIONS_V2 structure

A structure of this type is passed to ps3000aSetTriggerChannelConditionsV2 in the conditions argument to specify the trigger conditions, and is defined as follows: -

```
typedef struct tPS3000ATriggerConditionsV2
{
  PS3000A_TRIGGER_STATE channelA;
  PS3000A_TRIGGER_STATE channelB;
  PS3000A_TRIGGER_STATE channelC;
  PS3000A_TRIGGER_STATE channelD;
  PS3000A_TRIGGER_STATE external;
  PS3000A_TRIGGER_STATE aux;
  PS3000A_TRIGGER_STATE pulseWidthQualifier;
  PS3000A_TRIGGER_STATE digital;
} PS3000A_TRIGGER_CONDITIONS_V2;
```

Each structure is the logical AND of the states of the scope's inputs. ps3000aSetTriggerChannelConditionsV2 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 structure is byte-aligned. In C++, for example, you should specify this using the #pragma pack() instruction.

| | |
|---|---|
| **Elements** | channelA, channelB, channelC, channelD, external, pulseWidthQualifier, the type of condition that should be applied to each channel. Use these constants:<br>PS3000A_CONDITION_DONT_CARE<br>PS3000A_CONDITION_TRUE<br>PS3000A_CONDITION_FALSE<br><br>The channels that are set to PS3000A_CONDITION_TRUE or PS3000A_CONDITION_FALSE must all meet their conditions simultaneously to produce a trigger. Channels set to PS3000A_CONDITION_DONT_CARE are ignored.<br><br>aux, not used |

# 4.62 ps3000aSetTriggerChannelDirections

```
PICO_STATUS ps3000aSetTriggerChannelDirections
(
  int16_t                     handle,
  PS3000A_THRESHOLD_DIRECTION channelA,
  PS3000A_THRESHOLD_DIRECTION channelB,
  PS3000A_THRESHOLD_DIRECTION channelC,
  PS3000A_THRESHOLD_DIRECTION channelD,
  PS3000A_THRESHOLD_DIRECTION ext,
  PS3000A_THRESHOLD_DIRECTION aux
)
```

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

| | |
|---|---|
| **Applicability** | All modes |
| **Arguments** | `handle`, device identifier returned by `ps3000aOpenUnit`<br>`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 `ps3000aSetPulseWidthQualifierV2` for more information.<br><br>`aux`, not used |
| **Returns** | `PICO_OK`<br>`PICO_INVALID_HANDLE`<br>`PICO_USER_CALLBACK`<br>`PICO_INVALID_PARAMETER` |

**`PS3000A_THRESHOLD_DIRECTION` constants**

| | |
|---|---|
| `PS3000A_ABOVE` | for gated triggers: above the upper threshold |
| `PS3000A_ABOVE_LOWER` | for gated triggers: above the lower threshold |
| `PS3000A_BELOW` | for gated triggers: below the upper threshold |
| `PS3000A_BELOW_LOWER` | for gated triggers: below the lower threshold |
| `PS3000A_RISING` | for threshold triggers: rising edge, using upper threshold |
| `PS3000A_RISING_LOWER` | for threshold triggers: rising edge, using lower threshold |
| `PS3000A_FALLING` | for threshold triggers: falling edge, using upper threshold |
| `PS3000A_FALLING_LOWER` | for threshold triggers: falling edge, using lower threshold |
| `PS3000A_RISING_OR_FALLING` | for threshold triggers: either edge |
| `PS3000A_INSIDE` | for window-qualified triggers: inside window |
| `PS3000A_OUTSIDE` | for window-qualified triggers: outside window |
| `PS3000A_ENTER` | for window triggers: entering the window |
| `PS3000A_EXIT` | for window triggers: leaving the window |
| `PS3000A_ENTER_OR_EXIT` | for window triggers: either entering or leaving the window |
| `PS3000A_POSITIVE_RUNT` | for window-qualified triggers |
| `PS3000A_NEGATIVE_RUNT` | for window-qualified triggers |
| `PS3000A_NONE` | no trigger |

# 4.63 ps3000aSetTriggerChannelProperties

```
PICO_STATUS ps3000aSetTriggerChannelProperties
(
  int16_t                             handle,
  PS3000A_TRIGGER_CHANNEL_PROPERTIES * channelProperties,
  int16_t                             nChannelProperties,
  int16_t                             auxOutputEnable,
  int32_t                             autoTriggerMilliseconds
)
```

This function is used to enable or disable triggering and set its parameters.

| | |
|---|---|
| **Applicability** | All modes |
| **Arguments** | `handle`, device identifier returned by `ps3000aOpenUnit`<br>`* channelProperties`, a pointer to an array of `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 is switched off.<br><br>`nChannelProperties`, the size of the `channelProperties` array. If zero, triggering is switched off.<br><br>`auxOutputEnable`, not used<br><br>`autoTriggerMilliseconds`, the time in milliseconds for which the scope device will wait before collecting data if no trigger event occurs. If this is set to zero, the scope device will wait indefinitely for a trigger. |
| **Returns** | `PICO_OK`<br>`PICO_INVALID_HANDLE`<br>`PICO_USER_CALLBACK`<br>`PICO_TRIGGER_ERROR`<br>`PICO_MEMORY`<br>`PICO_INVALID_TRIGGER_PROPERTY`<br>`PICO_DRIVER_FUNCTION`<br>`PICO_INVALID_PARAMETER` |

## 4.63.1 PS3000A_TRIGGER_CHANNEL_PROPERTIES structure

A structure of this type is passed to ps3000aSetTriggerChannelProperties in the channelProperties argument to specify the trigger mechanism, and is defined as follows: -

```
typedef struct tPS3000ATriggerChannelProperties
{
  int16_t                 thresholdUpper;
  uint16_t                thresholdUpperHysteresis;
  int16_t                 thresholdLower;
  uint16_t                thresholdLowerHysteresis;
  PS3000A_CHANNEL         channel;
  PS3000A_THRESHOLD_MODE thresholdMode;
} PS3000A_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**

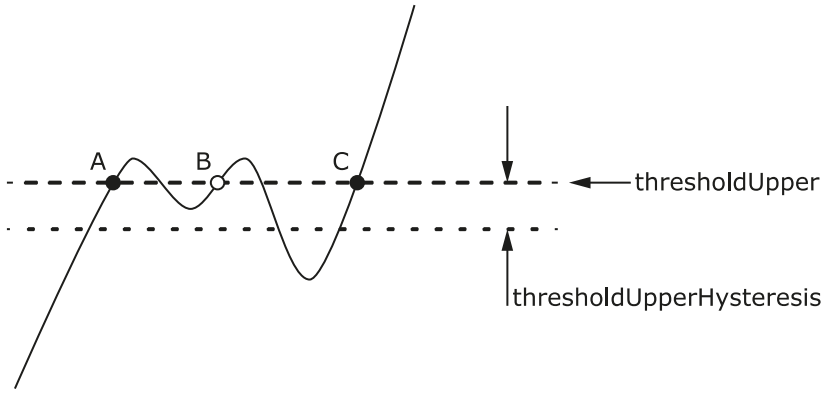
The digital triggering hardware in your PicoScope has two independent trigger thresholds called *upper* and *lower*. For some trigger types you can freely choose which threshold to use. The table in ps3000aSetTriggerChannelDirections shows which thresholds are available for use with which trigger types. 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 again. This double-threshold mechanism reduces unwanted trigger events caused by noisy or slowly changing signals.

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.<br><br>`thresholdUpperHysteresis`, the distance between the upper trigger threshold and the upper hysteresis threshold, scaled in 16-bit counts.<br><br>`thresholdLower`, `thresholdLowerHysteresis`, the settings for the lower threshold: see `thresholdUpper` and `thresholdUpperHysteresis`.<br><br>`channel`, the channel to which the properties apply. This can be one of the four input channels listed under `ps3000aSetChannel`, or `PS3000A_TRIGGER_EXT` for the **Ext** input fitted to some models.<br><br>`thresholdMode`, either a level or window trigger. Use one of these constants:<br>`PS3000A_LEVEL`<br>`PS3000A_WINDOW` |

# 4.64 ps3000aSetTriggerDelay

```
PICO_STATUS ps3000aSetTriggerDelay
(
  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 `ps3000aOpenUnit`<br>`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 500 MS/s, or 2 ns per sample, the total delay would be 100 x 2 ns = 200 ns.<br><br>Range: 0 to `MAX_DELAY_COUNT` |
| **Returns** | `PICO_OK`<br>`PICO_INVALID_HANDLE`<br>`PICO_USER_CALLBACK`<br>`PICO_DRIVER_FUNCTION` |

# 4.65 ps3000aSetTriggerDigitalPortProperties

```
PICO_STATUS ps3000aSetTriggerDigitalPortProperties
(
  int16_t                                handle,
  PS3000A_DIGITAL_CHANNEL_DIRECTIONS * directions
  int16_t                                nDirections
)
```

This function will set the individual digital channels' trigger directions. Each trigger direction consists of a channel name and a direction. If the channel is not included in the array of PS3000A_DIGITAL_CHANNEL_DIRECTIONS the driver assumes the digital channel's trigger direction is PS3000A_DIGITAL_DONT_CARE.

| | |
|---|---|
| **Applicability** | PicoScope 3000D MSO models only. |
| **Arguments** | handle, device identifier returned by ps3000aOpenUnit<br>* directions, a pointer to an array of PS3000A_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 triggering is switched off. A digital channel that is not included in the array will be set to PS3000A_DIGITAL_DONT_CARE. The outcomes of all the DIRECTIONS structures in the array are ORed together to produce the final trigger signal.<br><br>nDirections, the number of digital channel directions being passed to the driver |
| **Returns** | PICO_OK<br>PICO_INVALID_HANDLE<br>PICO_DRIVER_FUNCTION<br>PICO_INVALID_DIGITAL_CHANNEL<br>PICO_INVALID_DIGITAL_TRIGGER_DIRECTION |

### 4.65.1 PS3000A_DIGITAL_CHANNEL_DIRECTIONS structure

A structure of this type is passed to ps3000aSetTriggerDigitalPortProperties in the directions argument to specify the trigger mechanism, and is defined as follows: -

```
pragma pack(1)
typedef struct tPS3000ADigitalChannelDirections
{
  PS3000A_DIGITAL_CHANNEL   channel;
  PS3000A_DIGITAL_DIRECTION direction;
} PS3000A_DIGITAL_CHANNEL_DIRECTIONS;
#pragma pack ()

typedef enum enPS3000ADigitalChannel
{
  PS3000A_DIGITAL_CHANNEL_0,
  PS3000A_DIGITAL_CHANNEL_1,
  PS3000A_DIGITAL_CHANNEL_2,
  PS3000A_DIGITAL_CHANNEL_3,
  PS3000A_DIGITAL_CHANNEL_4,
  PS3000A_DIGITAL_CHANNEL_5,
  PS3000A_DIGITAL_CHANNEL_6,
  PS3000A_DIGITAL_CHANNEL_7,
  PS3000A_DIGITAL_CHANNEL_8,
  PS3000A_DIGITAL_CHANNEL_9,
  PS3000A_DIGITAL_CHANNEL_10,
  PS3000A_DIGITAL_CHANNEL_11,
  PS3000A_DIGITAL_CHANNEL_12,
  PS3000A_DIGITAL_CHANNEL_13,
  PS3000A_DIGITAL_CHANNEL_14,
  PS3000A_DIGITAL_CHANNEL_15,
  PS3000A_DIGITAL_CHANNEL_16,
  PS3000A_DIGITAL_CHANNEL_17,
  PS3000A_DIGITAL_CHANNEL_18,
  PS3000A_DIGITAL_CHANNEL_19,
  PS3000A_DIGITAL_CHANNEL_20,
  PS3000A_DIGITAL_CHANNEL_21,
  PS3000A_DIGITAL_CHANNEL_22,
  PS3000A_DIGITAL_CHANNEL_23,
  PS3000A_DIGITAL_CHANNEL_24,
  PS3000A_DIGITAL_CHANNEL_25,
  PS3000A_DIGITAL_CHANNEL_26,
  PS3000A_DIGITAL_CHANNEL_27,
  PS3000A_DIGITAL_CHANNEL_28,
  PS3000A_DIGITAL_CHANNEL_29,
  PS3000A_DIGITAL_CHANNEL_30,
  PS3000A_DIGITAL_CHANNEL_31,
  PS3000A_MAX_DIGITAL_CHANNELS
} PS3000A_DIGITAL_CHANNEL;

typedef enum enPS3000ADigitalDirection
{
  PS3000A_DIGITAL_DONT_CARE,
  PS3000A_DIGITAL_DIRECTION_LOW,
  PS3000A_DIGITAL_DIRECTION_HIGH,
  PS3000A_DIGITAL_DIRECTION_RISING,
  PS3000A_DIGITAL_DIRECTION_FALLING,
  PS3000A_DIGITAL_DIRECTION_RISING_OR_FALLING,
```

```
    PS3000A_DIGITAL_MAX_DIRECTION
  } PS3000A_DIGITAL_DIRECTION;
```

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

# 4.66 ps3000aSigGenArbitraryMinMaxValues

```
PICO_STATUS ps3000aSigGenArbitraryMinMaxValues
(
  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 ps3000aSetSignGenArbitrary for setting up the arbitrary waveform generator (AWG). These values vary between different models in the PicoScope 3000 Series.

| | |
|---|---|
| **Applicability** | All models with AWG |
| **Arguments** | handle, device identifier returned by ps3000aOpenUnit<br>minArbitraryWaveformValue, on exit, the lowest sample value allowed in the arbitraryWaveform buffer supplied to ps3000aSetSignGenArbitrary<br><br>maxArbitraryWaveformValue, on exit, the highest sample value allowed in the arbitraryWaveform buffer supplied to ps3000aSetSignGenArbitrary<br><br>minArbitraryWaveformSize, on exit, the minimum value allowed for the arbitraryWaveformSize argument supplied to ps3000aSetSignGenArbitrary<br><br>maxArbitraryWaveformSize, on exit, the maximum value allowed for the arbitraryWaveformSize argument supplied to ps3000aSetSignGenArbitrary |
| **Returns** | PICO_OK<br>PICO_NOT_SUPPORTED_BY_THIS_DEVICE, if the device does not have an arbitrary waveform generator.<br>PICO_NULL_PARAMETER, if all the parameter pointers are NULL.<br>PICO_INVALID_HANDLE<br>PICO_DRIVER_FUNCTION |

# 4.67 ps3000aSigGenFrequencyToPhase

```
PICO_STATUS ps3000aSigGenFrequencyToPhase
(
  int16_t              handle,
  double               frequency,
  PS3000A_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 (AWG). The value returned depends on the length of the buffer, the index mode passed and the device model. The phase count can then be sent to the driver through ps3000aSetSigGenArbitrary or ps3000aSetSigGenPropertiesArbitrary.

| | |
|---|---|
| **Applicability** | All models with AWG |
| **Arguments** | handle, device identifier returned by ps3000aOpenUnit<br>frequency, the required AWG output frequency<br><br>indexMode, see AWG index modes<br><br>bufferLength, the number of samples in the AWG buffer<br><br>phase, on exit, the deltaPhase argument to be sent to the AWG setup function |
| **Returns** | PICO_OK<br>PICO_NOT_SUPPORTED_BY_THIS_DEVICE, if the device does not have an AWG.<br>PICO_SIGGEN_FREQUENCY_OUT_OF_RANGE, if the frequency is out of range.<br>PICO_NULL_PARAMETER, if phase is a NULL pointer.<br>PICO_SIG_GEN_PARAM, if indexMode or bufferLength is out of range.<br>PICO_INVALID_HANDLE<br>PICO_DRIVER_FUNCTION |

# 4.68 ps3000aSigGenSoftwareControl

```
PICO_STATUS ps3000aSigGenSoftwareControl
(
  int16_t   handle,
  int16_t   state
)
```

This function causes a trigger event, or starts and stops gating. It is used when the signal generator is set to SIGGEN_SOFT_TRIG.

Gating occurs when the trigger type is set to either PS3000A_SIGGEN_GATE_HIGH or PS3000A_SIGGEN_GATE_LOW. With other trigger types, calling this function causes the signal generator to trigger immediately.

| | |
|---|---|
| **Applicability** | Use with ps3000aSetSigGenBuiltIn or ps3000aSetSigGenArbitrary. |
| **Arguments** | handle, device identifier returned by ps3000aOpenUnit<br>state, sets the trigger gate high or low:<br>0: gate low condition<br><> 0: gate high condition<br>Ignored if trigger type is not set to either PS3000A_SIGGEN_GATE_HIGH or PS3000A_SIGGEN_GATE_LOW. |
| **Returns** | PICO_OK<br>PICO_INVALID_HANDLE<br>PICO_NO_SIGNAL_GENERATOR<br>PICO_SIGGEN_TRIGGER_SOURCE<br>PICO_DRIVER_FUNCTION<br>PICO_NOT_RESPONDING |

# 4.69 ps3000aStop

```
PICO_STATUS ps3000aStop
(
  int16_t   handle
)
```

This function stops the scope device from sampling data. If this function is called before a trigger event occurs, the oscilloscope may not contain valid data.

The function is mainly used in streaming mode to stop a streaming capture. It can optionally be used in block mode to stop a capture early, either before or after triggering; and in rapid block mode to stop a sequence of captures. If a block mode capture is interrupted, ps3000aGetValues will indicate that no samples are available and the buffer will contain no data.

Always call this function after the end of a capture to ensure that the scope is ready for the next capture.

| | |
|---|---|
| **Applicability** | All modes |
| **Arguments** | `handle`, device identifier returned by `ps3000aOpenUnit` |
| **Returns** | `PICO_OK`<br>`PICO_INVALID_HANDLE`<br>`PICO_USER_CALLBACK`<br>`PICO_DRIVER_FUNCTION` |

# 4.70 ps3000aStreamingReady (callback)

```
typedef void (CALLBACK *ps3000aStreamingReady)
(
  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 ps3000aGetStreamingLatestValues, and the driver calls it back when streaming-mode data is ready. You can then download the data using the ps3000aGetValuesAsync function.

Your callback 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 ps3000aOpenUnit<br>`noOfSamples`, the number of samples to collect<br><br>`startIndex`, an index to the first valid sample in the buffer. This is the buffer that was previously passed to ps3000aSetDataBuffer.<br><br>`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 denoting Channel A.<br><br>`triggerAt`, an index to the buffer indicating the location of the trigger point relative to `startIndex`. This parameter is valid only when `triggered` is non-zero.<br><br>`triggered`, a flag indicating whether a trigger occurred. If non-zero, a trigger occurred at the location indicated by `triggerAt`.<br><br>`autoStop`, the flag that was set in the call to ps3000aRunStreaming<br><br>`* pParameter`, a void pointer passed from ps3000aGetStreamingLatestValues. The callback function can write to this location to send any data, such as a status flag, back to the application. |
| **Returns** | nothing |

# 5 Wrapper functions

The wrapper functions are for use with programming languages that do not support features of C such as callback functions.

To use the wrapper functions you must include the `ps3000aWrap.dll` library, which is supplied in the SDK, in your project.

For all other functions, see the list of API functions.

## 5.1 Using the wrapper functions for streaming data capture

1. Open the oscilloscope using `ps3000aOpenUnit`.
   1a. Register the handle with the wrapper and obtain a device index for use with some wrapper function calls by calling `initWrapUnitInfo`.
   1b. Inform the wrapper of the number of channels on the device by calling `setChannelCount`.
   1c. [MSOs only] Inform the wrapper of the number of digital ports on the device by calling `setDigitalPortCount`.

2. Select channels, ranges and AC/DC coupling using `ps3000aSetChannel`.
   2a. Inform the wrapper which channels have been enabled by calling `setEnabledChannels`.
   2b. [MSOs only] Inform the wrapper which digital ports have been enabled by calling `setEnabledDigitalPorts`.

3. [MSOs only] Set the digital port using `ps3000aSetDigitalPort`.

4. Use the trigger setup functions `ps3000aSetTriggerChannelConditionsV2`, `ps3000aSetTriggerChannelDirections` and `ps3000aSetTriggerChannelProperties` to set up the trigger if required. For programming languages that do not support structures, use the wrapper's `SetTriggerConditionsV2` in place of `ps3000aSetTriggerCHannelConditionsV2` and `SetTriggerProperties` in place of `ps3000aSetTriggerChannelProperties`.

5. [MSOs only] Use the trigger setup function `ps3000aSetTriggerDigitalPortProperties` to set up the digital trigger if required.

6. Call `ps3000aSetDataBuffer` to tell the driver where your data buffer is.
   6a. Register the data buffer(s) with the wrapper and set the application buffer into which the data will be copied.
   - For analog channels: Call `setAppAndDriverBuffers` or `setMaxMinAppAndDriverBuffers`.
   - [MSOs Only] For digital ports: Call `setAppAndDriverDigiBuffers` or `setMaxMinAppAndDriverDigiBuffers`.

7. Set up aggregation and start the oscilloscope running using `ps3000aRunStreaming`.

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's function.

10. Call ps3000aStop, even if Auto Stop is enabled.

11. To disconnect a device, call ps3000aCloseUnit followed by the wrapper's decrementDeviceCount function.

12. Call the resetNextDeviceIndex wrapper function.

## 5.2 AutoStopped

```
int16_t AutoStopped
(
  uint16_t deviceIndex
)
```

This function indicates if the device has stopped after collecting of the number of samples specified in the call to ps3000aRunStreaming. This occurs only if the ps3000aRunStreaming function's autoStop flag is set.

| | |
|---|---|
| **Applicability** | Streaming mode |
| **Arguments** | deviceIndex, identifies the required device |
| **Returns** | 0 – if streaming has not stopped or deviceIndex is out of range<br><> 0 – if streaming has stopped automatically |

# 5.3 AvailableData

```
uint32_t AvailableData
(
  uint16_t   deviceIndex,
  uint32_t * startIndex
)
```

This function indicates the number of samples returned from the driver and shows the start index of the data in the buffer when collecting data in streaming mode.

| | |
|---|---|
| **Applicability** | Streaming mode |
| **Arguments** | `deviceIndex`, identifies the required device<br><br>`startIndex`, on exit, an index to the first valid sample in the buffer (when data is available) |
| **Returns** | `0` – data is not yet available or the device index is invalid<br>`<>0` – the number of samples returned from the driver |

## 5.4 BlockCallback

```
void BlockCallback
(
  int16_t       handle,
  PICO_STATUS   status,
  void        * pParameter
)
```

This is a wrapper for the ps3000aBlockReady callback. The driver calls it back when block-mode data is ready.

| | |
|---|---|
| **Applicability** | Block mode |
| **Arguments** | See ps3000aBlockReady |
| **Returns** | Nothing |

## 5.5 ClearTriggerReady

```
PICO_STATUS ClearTriggerReady
(
  uint16_t deviceIndex
)
```

This function clears the `triggered` and `triggeredAt` flags for use with streaming-mode capture.

| | |
|---|---|
| **Applicability** | Streaming mode |
| **Arguments** | `deviceIndex`, identifies the device to use |
| **Returns** | `PICO_OK`, if successful<br>`PICO_INVALID_PARAMETER`, if `deviceIndex` is out of bounds |

## 5.6 decrementDeviceCount

```
PICO_STATUS decrementDeviceCount
(
  uint16_t deviceIndex
)
```

Reduces the count of the number of PicoScope devices being controlled by the application.

Note: This function does not close the connection to the device being controlled. Use the ps3000aCloseUnit function for this.

| | |
|---|---|
| **Applicability** | All modes |
| **Arguments** | deviceIndex, identifies the device to use |
| **Returns** | PICO_OK, if successful<br>PICO_INVALID_PARAMETER, if deviceIndex is out of bounds |

# 5.7 getDeviceCount

```
uint16_t getDeviceCount
(
  void
)
```

This function returns the number of PicoScope 3000 Series devices being controlled by the application.

| | |
|---|---|
| **Applicability** | All modes |
| **Arguments** | None |
| **Returns** | The number of PicoScope 3000 Series devices being controlled |

## 5.8 GetStreamingLatestValues

```
PICO_STATUS GetStreamingLatestValues
(
  uint16_t deviceIndex
)
```

This function returns the next block of values to your application when capturing data in streaming mode. Use with programming languages that do not support callback functions.

| | |
|---|---|
| **Applicability** | Streaming mode |
| **Arguments** | `deviceIndex`, identifies the required device |
| **Returns** | `PICO_INVALID_PARAMETER`, if `deviceIndex` is invalid<br>See also `ps3000aGetStreamingLatestValues` return values |

# 5.9 initWrapUnitInfo

```
PICO_STATUS initWrapUnitInfo
(
  int16_t    handle,
  uint16_t * deviceIndex
)
```

This function initializes a `WRAP_UNIT_INFO` structure for a PicoScope 3000 Series device and places it in the `g_deviceInfo` array at the next available index.

The wrapper supports a maximum of 4 devices.

Your main application should map the handle to the index starting with the first handle corresponding to index 0.

| | |
|---|---|
| **Applicability** | All modes |
| **Arguments** | `deviceIndex`, on exit, the index at which the `WRAP_UNIT_INFO` structure will be stored in the `g_deviceInfo` array |
| **Returns** | `PICO_OK`, if successful<br>`PICO_INVALID_HANDLE`, if the handle is less than or equal to 0<br>`PICO_MAX_UNITS_OPENED`, if the wrapper already has records for the maximum number of devices that it will support |

# 5.10 IsReady

```
int16_t IsReady
(
  uint16_t deviceIndex
)
```

This function polls the driver to verify that streaming data is ready to be received. You must call the RunBlock or GetStreamingLatestValues before calling this function.

| | |
|---|---|
| **Applicability** | Streaming mode. (In block mode, we recommend using ps3000aIsReady instead.) |
| **Arguments** | deviceIndex, the index assigned by the wrapper corresponding to the required device |
| **Returns** | 0 – data is not yet available or deviceIndex is out of range<br><>0 – data is ready to be collected |

# 5.11 IsTriggerReady

```
int16_t IsTriggerReady
(
  uint16_t   deviceIndex
  uint32_t * triggeredAt
)
```

This function indicates whether a trigger has occurred when collecting data in streaming mode, and provides the location of the trigger point in the buffer.

| | |
|---|---|
| **Applicability** | Streaming mode |
| **Arguments** | `deviceIndex`, the index assigned by the wrapper corresponding to the required device<br><br>`triggeredAt`, on exit, the index of the sample in the buffer where the trigger occurred, relative to the first valid sample index. This value is set to 0 when the function returns 0. |
| **Returns** | `0` – the device has not triggered, or `deviceIndex` is invalid<br>`<>0` – the device has been triggered |

# 5.12 resetNextDeviceIndex

```
PICO_STATUS resetNextDeviceIndex
(
  void
)
```

This function is used to reset the index used to determine the next point at which to store a `WRAP_UNIT_INFO` structure.

Call this function only after the devices have been disconnected.

| | |
|---|---|
| **Applicability** | All modes |
| **Arguments** | None |
| **Returns** | `PICO_OK` |

# 5.13 RunBlock

```
PICO_STATUS RunBlock
(
  uint16_t   deviceIndex,
  int32_t    preTriggerSamples,
  int32_t    postTriggerSamples,
  uint32_t   timebase,
  uint32_t    segmentIndex
)
```

This function starts collecting data in block mode without the requirement for specifying callback functions. Use the `IsReady` function to poll the driver once this function has been called.

| | |
|---|---|
| **Applicability** | Block mode |
| **Arguments** | `deviceIndex`, the index assigned by the wrapper corresponding to the required device<br><br>`preTriggerSamples`,<br>`postTriggerSamples`, see `noOfPreTriggerSamples` in `ps3000aRunBlock`<br><br>`timebase`,<br>`segmentIndex`, see `ps3000aRunBlock` |
| **Returns** | See `ps3000aRunBlock` return values |

# 5.14 setAppAndDriverBuffers

```
PICO_STATUS setAppAndDriverBuffers
(
  uint16_t   deviceIndex,
  int16_t    channel,
  int16_t  * appBuffer,
  int16_t  * driverBuffer,
  uint32_t   bufferLength
)
```

This function sets the application buffer and corresponding driver buffer in order for the streaming callback to copy the data for the analog channel from the driver buffer to the application buffer.

| | |
|---|---|
| **Applicability** | Streaming mode |
| **Arguments** | `deviceIndex`, the index assigned by the wrapper corresponding to the required device<br><br>`channel`, the channel number (should be a numerical value corresponding to a `PS3000A_CHANNEL` enumeration value)<br><br>`appBuffer`, the application buffer<br><br>`driverBuffer`, the buffer set by the driver<br><br>`bufferLength`, the length of the buffers (the lengths of the buffers must be equal) |
| **Returns** | `PICO_OK`, if successful<br>`PICO_INVALID_PARAMETER`, if `deviceIndex` is out of bounds<br>`PICO_INVALID_CHANNEL`, if channel is not valid |

# 5.15 setMaxMinAppAndDriverBuffers

```
PICO_STATUS setMaxMinAppAndDriverBuffers
(
  uint16_t   deviceIndex,
  int16_t    channel,
  int16_t  * appMaxBuffer,
  int16_t  * appMinBuffer,
  int16_t  * driverMaxBuffer,
  int16_t  * driverMinBuffer,
  uint32_t   bufferLength
)
```

Set the application buffer and corresponding driver buffer in order for the streaming callback to copy the data for the analog channel from the driver maximum and minimum buffers to the respective application buffers for aggregated data collection.

| | |
|---|---|
| **Applicability** | Streaming mode |
| **Arguments** | `deviceIndex`, the index assigned by the wrapper corresponding to the required device<br><br>`channel`, the channel number (should be a numerical value corresponding to a `PS3000A_CHANNEL` enumeration value)<br><br>`appMaxBuffer`, the application buffer for maximum values (the 'max buffer')<br><br>`appMinBuffer`, the application buffer for minimum values (the 'min buffer')<br><br>`driverMaxBuffer`, the max buffer set by the driver<br><br>`driverMinBuffer`, the min buffer set by the driver<br><br>`bufferLength`, the length of the buffers (the lengths of the buffers must be equal) |
| **Returns** | `PICO_OK`, if successful<br>`PICO_INVALID_PARAMETER`, if `deviceIndex` is out of bounds<br>`PICO_INVALID_CHANNEL`, if channel is not valid |

# 5.16 setAppAndDriverDigiBuffers

```
PICO_STATUS setAppAndDriverDigiBuffers
(
  uint16_t   deviceIndex,
  int16_t    digiPort,
  int16_t  * appDigiBuffer,
  int16_t  * driverDigiBuffer,
  uint32_t   bufferLength
)
```

This function sets the application buffer and corresponding driver buffer in order for the streaming callback to copy the data for the digital port from the driver buffer to the application buffer.

| | |
|---|---|
| **Applicability** | Streaming mode. PicoScope 3000 MSO and 3000D MSO models only. |
| **Arguments** | `deviceIndex`, the index assigned by the wrapper corresponding to the required device<br><br>`digiPort`, the digital port number (0 or 1)<br><br>`appDigiBuffer`, the application buffer for the digital port<br><br>`driverDigitalBuffer`, the buffer for the digital port set by the driver<br><br>`bufferLength`, the length of the buffers (the lengths of the buffers must be equal) |
| **Returns** | `PICO_OK`, if successful<br>`PICO_INVALID_PARAMETER`, if `deviceIndex` is out of bounds<br>`PICO_INVALID_DIGITAL_PORT`, if `digiPort` is not 0 (Port 0) or 1 (Port 1) |

# 5.17 setMaxMinAppAndDriverDigiBuffers

```
PICO_STATUS setMaxMinAppAndDriverDigiBuffers
(
  uint16_t   deviceIndex,
  int16_t    digiPort,
  int16_t  * appMaxDigiBuffer,
  int16_t  * appMinDigiBuffer,
  int16_t  * driverMaxDigiBuffer,
  int16_t  * driverMinDigiBuffer,
  uint32_t   bufferLength
)
```

This functions sets the application buffers and corresponding driver buffers in order for the streaming callback to copy the data for the digital port from the driver 'max' and 'min' buffers to the respective application buffers for aggregated data collection.

| | |
|---|---|
| **Applicability** | Streaming mode. PicoScope 3000 MSO and 3000D models only. |
| **Arguments** | `deviceIndex`, the index assigned by the wrapper corresponding to the required device<br><br>`digiPort`, the digital port number (0 or 1)<br><br>`appMaxDigiBuffer`, the application max. buffer for the digital port<br><br>`appMinDigiBuffer`, the application min. buffer for the digital port<br><br>`driverMaxDigiBuffer`, the max. buffer set by the driver for the digital port<br><br>`driverMinDigiBuffer`, the min. buffer set by the driver for the digital port<br><br>`bufferLength`, the length of the buffers (the lengths of the buffers must be equal) |
| **Returns** | `PICO_OK`, if successful<br>`PICO_INVALID_PARAMETER`, if `deviceIndex` is out of bounds<br>`PICO_INVALID_DIGITAL_PORT`, if `digiPort` is not 0 (Port 0) or 1 (Port 1) |

# 5.18 setChannelCount

```
PICO_STATUS setChannelCount
(
  uint16_t deviceIndex,
  int16_t  channelCount
)
```

This function sets the number of analog channels on the device. This is used to assist with copying data in the streaming callback.

You must call `initWrapUnitInfo` before calling this function.

| | |
|---|---|
| **Applicability** | Streaming mode |
| **Arguments** | `deviceIndex`, the index assigned by the wrapper corresponding to the required device<br><br>`channelCount`, the number of channels on the device |
| **Returns** | `PICO_OK`, if successful<br>`PICO_INVALID_PARAMETER`, if `deviceIndex` is out of bounds or `channelCount` is not 2 or 4 |

# 5.19 setDigitalPortCount

```
PICO_STATUS setDigitalPortCount
(
  uint16_t deviceIndex,
  int16_t  digitalPortCount
)
```

Set the number of digital ports on the device. This is used to assist with copying data in the streaming callback.

You must call `initWrapUnitInfo` before calling this function.

| | |
|---|---|
| **Applicability** | Streaming mode |
| **Arguments** | `deviceIndex`, the index assigned by the wrapper corresponding to the required device<br><br>`digitalPortCount`, the number of digital ports on the device. Set to 2 for the PicoScope 3000 MSO and 3000D MSO devices and 0 for other models. |
| **Returns** | `PICO_OK`, if successful<br>`PICO_INVALID_PARAMETER`, `deviceIndex` is out of bounds or `digitalPortCount` is invalid |

# 5.20 setEnabledChannels

```
PICO_STATUS setEnabledChannels
(
  uint16_t   deviceIndex,
  int16_t  * enabledChannels
)
```

Set the number of enabled analog channels on the device. This is used to assist with copying data in the streaming callback.

You must call `setChannelCount` before calling this function.

| | |
|---|---|
| **Applicability** | Streaming mode |
| **Arguments** | `deviceIndex`, the index assigned by the wrapper corresponding to the required device<br><br>`enabledChannels`, an array of 4 elements representing the channel states |
| **Returns** | `PICO_OK`, if successful<br>`PICO_INVALID_PARAMETER`, if `deviceIndex` is out of bounds or `channelCount` is not 2 or 4 |

# 5.21 setEnabledDigitalPorts

```
PICO_STATUS setEnabledDigitalPorts
(
  uint16_t   deviceIndex,
  int16_t  * enabledDigitalPorts
)
```

This function sets the number of enabled digital ports on the device. This is used to assist with copying data in the streaming callback.

For PicoScope 3000 MSO and 3000D MSO models, you must call `setDigitalPortCount` first.

| | |
|---|---|
| **Applicability** | Streaming mode |
| **Arguments** | `deviceIndex`, the index assigned by the wrapper corresponding to the required device<br><br>`enabledDigitalPorts`, an array of 4 elements representing the digital port states |
| **Returns** | `PICO_OK`, if successful<br>`PICO_INVALID_PARAMETER`, if `deviceIndex` is out of bounds, or `digitalPortCount` is invalid |

# 5.22 SetPulseWidthQualifier

```
PICO_STATUS SetPulseWidthQualifier
(
  int16_t    handle,
  uint32_t * pwqConditionsArray,
  int16_t    nConditions,
  uint32_t   direction,
  uint32_t   lower,
  uint32_t   upper,
  uint32_t   type
)
```

This function sets up pulse-width qualification, which can be used on its own for pulse-width triggering or combined with level triggering or window triggering to produce more complex triggers.

The pulse-width qualifier is defined by one or more sets of integers corresponding to `PS3000A_PWQ_CONDITIONS` structures which are then converted and passed to `ps3000aSetPulseWidthQualifier`.

Use this function with programming languages that do not support structs.

| | |
|---|---|
| **Applicability** | Analog-input models only (for MSOs, use `SetPulseWidthQualifierV2`) |
| **Arguments** | `handle`, the handle of the required device<br><br>`pwqConditionsArray`, an array of integer values specifying the conditions for each channel<br><br>`nConditions`, the number that will be passed after the wrapper code has created its structures (i.e. the number of `pwqConditionsArray` elements / 6)<br><br>`direction`, the direction of the signal required for the pulse width trigger to fire (see `PS3000A_THRESHOLD_DIRECTION` enumerations)<br><br>`lower`, the lower limit of the pulse-width counter, measured in samples<br><br>`upper`, the upper limit of the pulse-width counter, measured in samples<br><br>`type`, the pulse-width type (see `PS3000A_PULSE_WIDTH_TYPE` enumerations) |
| **Returns** | See `ps3000aSetPulseWidthQualifier` return values |

# 5.23 SetPulseWidthQualifierV2

```
PICO_STATUS SetPulseWidthQualifierV2
(
  int16_t    handle,
  uint32_t * pwqConditionsArrayV2,
  int16_t    nConditions,
  uint32_t   direction,
  uint32_t   lower,
  uint32_t   upper,
  uint32_t   type
)
```

This function sets up pulse-width qualification, which can be used on its own for pulse-width triggering or combined with level triggering or window triggering to produce more complex triggers.

The pulse-width qualifier is defined by one or more sets of integers corresponding to `PS3000A_PWQ_CONDITIONS_V2` structures which are then converted and passed to `ps3000aSetPulseWidthQualifierV2`.

Use this function with programming languages that do not support structs.

| | |
|---|---|
| **Applicability** | All models |
| **Arguments** | `handle`, the handle of the required device<br><br>`pwqConditionsArray`, an array of integer values specifying the conditions for each channel<br><br>`nConditions`, the number that will be passed after the wrapper code has created its structures (i.e. the number of `pwqConditionsArrayV2` elements / 6)<br><br>`direction`, the direction of the signal required for the pulse width trigger to fire (see `PS3000A_THRESHOLD_DIRECTION` enumerations)<br><br>`lower`, the lower limit of the pulse-width counter, measured in samples<br><br>`upper`, the upper limit of the pulse-width counter, measured in samples<br><br>`type`, the pulse-width type (see `PS3000A_PULSE_WIDTH_TYPE` enumerations) |
| **Returns** | See `ps3000aSetPulseWidthQualifier` return values |

# 5.24 SetTriggerConditions

```
PICO_STATUS SetTriggerConditions
(
  int16_t   handle,
  int32_t * conditionsArray,
  int16_t   nConditions
)
```

This function sets up trigger conditions on the scope's inputs. The trigger is defined by one or more sets of integers corresponding to `PS3000A_TRIGGER_CONDITIONS` structures which are then converted and passed to `ps3000aSetTriggerChannelConditions`.

Use this function with programming languages that do not support structs.

| | |
|---|---|
| **Applicability** | Analog-input models only ( for MSOs use `SetTriggerConditionsV2`) |
| **Arguments** | `handle`, the handle of the required device<br><br>`conditionsArray`, an array of integer values specifying the conditions for each channel<br><br>`nConditions`, the number that will be passed after the wrapper code has created its structures (i.e. the number of `conditionsArray` elements divided by 7) |
| **Returns** | See `ps3000aSetTriggerChannelConditions` return values |

### Examples

Below are examples for using the function in Visual Basic.

### To trigger off channels A OR B

```
Dim conditionsArray(13) As Integer
conditionsArray(0) = 1          ' channel A
conditionsArray(1) = 0          ' channel B
conditionsArray(2) = 0          ' channel C
conditionsArray(3) = 0          ' channel D
conditionsArray(4) = 0          ' external
conditionsArray(5) = 0          ' aux
conditionsArray(6) = 0          ' pulse width qualifier

' *** OR'ed with

conditionsArray(7) = 0          ' channel A
conditionsArray(8) = 1          ' channel B
conditionsArray(9) = 0          ' channel C
conditionsArray(10) = 0         ' channel D
conditionsArray(11) = 0         ' external
conditionsArray(12) = 0         ' aux
conditionsArray(13) = 0         ' pulse width qualifier
status = SetTriggerConditions(handle, conditionsArray(0), 2)
```

### To trigger off channels A AND B

```
Dim conditionsArray(6) As Integer
conditionsArray(0) = 1          ' channel A
conditionsArray(1) = 1          ' channel B
conditionsArray(2) = 0          ' channel C
conditionsArray(3) = 0          ' channel D
```

```
conditionsArray(4) = 0        ' external
conditionsArray(5) = 0        ' aux
conditionsArray(6) = 0        ' pulse width qualifier

status = SetTriggerConditions(handle, conditionsArray(0), 1)
```

# 5.25 SetTriggerConditionsV2

```
PICO_STATUS SetTriggerConditionsV2
(
  int16_t   handle,
  int32_t * conditionsArrayV2,
  int16_t   nConditions
)
```

This function sets up trigger conditions on the scope's inputs. The trigger is defined by one or more sets of integers corresponding to `PS3000A_TRIGGER_CONDITIONS_V2` structures which are then converted and passed to `ps3000aSetTriggerChannelConditionsV2`.

Use this function with programming languages that do not support structs.

| | |
|---|---|
| **Applicability** | All models |
| **Arguments** | `handle`, the handle of the required device<br><br>`conditionsArrayV2`, an array of integer values specifying the conditions for each channel<br><br>`nConditions`, the number that will be passed after the wrapper code has created its structures (i.e. the number of `conditionsArray` elements divided by 8) |
| **Returns** | See `ps3000aSetTriggerChannelConditionsV2` return values |

# 5.26 SetTriggerProperties

```
PICO_STATUS SetTriggerProperties
(
  int16_t   handle,
  int32_t * propertiesArray,
  int16_t   nProperties,
  int32_t   autoTrig
)
```

This function is used to enable or disable triggering and set its parameters. This is done by assigning the values from the `propertiesArray` to an array of `PS3000A_TRIGGER_CHANNEL_PROPERTIES` structures which are then passed to the `ps3000aSetTriggerChannelProperties` function with the other parameters.

Use this function with programming languages that do not support structs.

| | |
|---|---|
| **Applicability** | All modes |
| **Arguments** | `handle`, the handle of the required device<br><br>`propertiesArray`, an array of sets of integers corresponding to `PS3000A_TRIGGER_CHANNEL_PROPERTIES` structures describing the required properties to be set. See also `channelProperties` in `ps3000aSetTriggerChannelProperties`.<br><br>`nProperties`, the number that will be passed after the wrapper code has created its structures (i.e. the number of `propertiesArray` elements divided by 6)<br><br>`autoTrig`, see `autoTriggerMilliseconds` in `ps3000aSetTriggerChannelProperties` |
| **Returns** | See `ps3000aSetTriggerChannelProperties` return values |

**Example**

Here is an example for using the function in Visual Basic:

```
Dim propertiesArray(11) As Integer

'channel A
propertiesArray(0) = 1500 ' Upper
propertiesArray(1) = 300  ' UpperHysteresis
propertiesArray(2) = 0    ' Lower
propertiesArray(3) = 0    ' LowerHysteresis
propertiesArray(4) = 0    ' channel (0=ChA, 1=ChB, 2=ChC, 3=ChD)
propertiesArray(5) = 0    ' thresholdMode (Level=0, Window=1)

'channel B
propertiesArray(6) = 1500 ' Upper
propertiesArray(7) = 300  ' UpperHysteresis
propertiesArray(8) = 0    ' Lower
propertiesArray(9) = 0    ' LowerHysteresis
propertiesArray(10) = 1   ' channel (0=ChA, 1=ChB, 2=ChC, 3=ChD)
propertiesArray(11) = 0   ' thresholdMode (Level=0, Window=1)

status = SetTriggerProperties(handle, propertiesArray(0), 2, 0, 1000)
```

# 5.27 StreamingCallback

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

This is a wrapper for the ps3000aStreamingReady callback. The driver calls it back when streaming-mode data is ready.

| | |
|---|---|
| **Applicability** | Streaming mode |
| **Arguments** | See ps3000aStreamingReady |
| **Returns** | Nothing |

# 6 Programming examples

Your PicoScope SDK installation includes example code in a number of programming languages and development environments. Please refer to the SDK for details.

# 7 Reference

## 7.1 Numeric data types

Here is a list of the sizes and ranges of the numeric data types used in the *ps3000a* 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) |
| double | 64 | signed (IEEE 754) |
| int64_t | 64 | signed |
| uint64_t | 64 | unsigned |

## 7.2 Enumerated types, constants and structures

The enumerated types, constants and structures used in the *ps3000a* API are defined in the file ps3000aApi.h. We recommend that you refer to these constants by name unless your programming language allows only numerical values.

## 7.3 Driver status codes

Every function in the *ps3000a* driver returns a driver status code from the following list of PICO_STATUS values. These definitions can also be found in the file PicoStatus.h, which is included in the inc subdirectory of the *ps3000a* SDK. Not all codes apply to the *ps3000a* API.

| Code (hex) | Symbol and meaning |
|---|---|
| 00 | PICO_OK<br>The PicoScope is functioning correctly |
| 01 | PICO_MAX_UNITS_OPENED<br>An attempt has been made to open more than PS3000A_MAX_UNITS. |
| 02 | PICO_MEMORY_FAIL<br>Not enough memory could be allocated on the host machine |
| 03 | PICO_NOT_FOUND<br>No PicoScope could be found |
| 04 | PICO_FW_FAIL<br>Unable to download firmware |
| 05 | PICO_OPEN_OPERATION_IN_PROGRESS |
| 06 | PICO_OPERATION_FAILED |
| 07 | PICO_NOT_RESPONDING<br>The PicoScope is not responding to commands from the PC |
| 08 | PICO_CONFIG_FAIL<br>The configuration information in the PicoScope has become corrupt or is missing |
| 09 | PICO_KERNEL_DRIVER_TOO_OLD<br>The picopp.sys file is too old to be used with the device driver |

| | |
|---|---|
| 0A | PICO_EEPROM_CORRUPT<br>The EEPROM has become corrupt, so the device will use a default setting |
| 0B | PICO_OS_NOT_SUPPORTED<br>The operating system on the PC is not supported by this driver |
| 0C | PICO_INVALID_HANDLE<br>There is no device with the handle value passed |
| 0D | PICO_INVALID_PARAMETER<br>A parameter value is not valid |
| 0E | PICO_INVALID_TIMEBASE<br>The timebase is not supported or is invalid |
| 0F | PICO_INVALID_VOLTAGE_RANGE<br>The voltage range is not supported or is invalid |
| 10 | PICO_INVALID_CHANNEL<br>The channel number is not valid on this device or no channels have been set |
| 11 | PICO_INVALID_TRIGGER_CHANNEL<br>The channel set for a trigger is not available on this device |
| 12 | PICO_INVALID_CONDITION_CHANNEL<br>The channel set for a condition is not available on this device |
| 13 | PICO_NO_SIGNAL_GENERATOR<br>The device does not have a signal generator |
| 14 | PICO_STREAMING_FAILED<br>Streaming has failed to start or has stopped without user request |
| 15 | PICO_BLOCK_MODE_FAILED<br>Block failed to start - a parameter may have been set wrongly |
| 16 | PICO_NULL_PARAMETER<br>A parameter that was required is NULL |
| 18 | PICO_DATA_NOT_AVAILABLE<br>No data is available from a run block call |
| 19 | PICO_STRING_BUFFER_TOO_SMALL<br>The buffer passed for the information was too small |
| 1A | PICO_ETS_NOT_SUPPORTED<br>ETS is not supported on this device |
| 1B | PICO_AUTO_TRIGGER_TIME_TOO_SHORT<br>The auto trigger time is less than the time it will take to collect the pre-trigger data |
| 1C | PICO_BUFFER_STALL<br>The collection of data has stalled as unread data would be overwritten |
| 1D | PICO_TOO_MANY_SAMPLES<br>Number of samples requested is more than available in the current memory segment |
| 1E | PICO_TOO_MANY_SEGMENTS<br>Not possible to create number of segments requested |
| 1F | PICO_PULSE_WIDTH_QUALIFIER<br>A null pointer has been passed in the trigger function or one of the parameters is out of range |
| 20 | PICO_DELAY<br>One or more of the hold-off parameters are out of range |
| 21 | PICO_SOURCE_DETAILS<br>One or more of the source details are incorrect |
| 22 | PICO_CONDITIONS<br>One or more of the conditions are incorrect |
| 23 | PICO_USER_CALLBACK |

| | |
|---|---|
| | The driver's thread is currently in the ps3000a...Ready callback function and therefore the action cannot be carried out |
| 24 | PICO_DEVICE_SAMPLING<br>An attempt is being made to get stored data while streaming. Either stop streaming by calling ps3000aStop, or use ps3000aGetStreamingLatestValues |
| 25 | PICO_NO_SAMPLES_AVAILABLE<br>...because a run has not been completed |
| 26 | PICO_SEGMENT_OUT_OF_RANGE<br>The memory index is out of range |
| 27 | PICO_BUSY<br>Data cannot be returned yet |
| 28 | PICO_STARTINDEX_INVALID<br>The start time to get stored data is out of range |
| 29 | PICO_INVALID_INFO<br>The information number requested is not a valid number |
| 2A | PICO_INFO_UNAVAILABLE<br>The handle is invalid so no information is available about the device. Only PICO_DRIVER_VERSION is available. |
| 2B | PICO_INVALID_SAMPLE_INTERVAL<br>The sample interval selected for streaming is out of range |
| 2C | PICO_TRIGGER_ERROR |
| 2D | PICO_MEMORY<br>Driver cannot allocate memory |
| 2E | PICO_SIG_GEN_PARAM<br>Incorrect parameter passed to the signal generator |
| 2F | PICO_SHOTS_SWEEPS_WARNING<br>Conflict between the shots and sweeps parameters sent to the signal generator |
| 33 | PICO_WARNING_EXT_THRESHOLD_CONFLICT<br>Attempt to set different EXT input thresholds for signal generator and oscilloscope trigger |
| 35 | PICO_SIGGEN_OUTPUT_OVER_VOLTAGE<br>The combined peak to peak voltage and the analog offset voltage exceed the allowable voltage the signal generator can produce |
| 36 | PICO_DELAY_NULL<br>NULL pointer passed as delay parameter |
| 37 | PICO_INVALID_BUFFER<br>The buffers for overview data have not been set while streaming |
| 38 | PICO_SIGGEN_OFFSET_VOLTAGE<br>The analog offset voltage is out of range |
| 39 | PICO_SIGGEN_PK_TO_PK<br>The analog peak to peak voltage is out of range |
| 3A | PICO_CANCELLED<br>A block collection has been cancelled |
| 3B | PICO_SEGMENT_NOT_USED<br>The segment index is not currently being used |
| 3C | PICO_INVALID_CALL<br>The wrong GetValues function has been called for the collection mode in use |
| 3F | PICO_NOT_USED<br>The function is not available |
| 40 | PICO_INVALID_SAMPLERATIO<br>The aggregation ratio requested is out of range |
| 41 | PICO_INVALID_STATE |

| | |
|---|---|
| | Device is in an invalid state |
| 42 | PICO_NOT_ENOUGH_SEGMENTS<br>The number of segments allocated is fewer than the number of captures requested |
| 43 | PICO_DRIVER_FUNCTION<br>You called a driver function while another driver function was still being processed |
| 44 | PICO_RESERVED |
| 45 | PICO_INVALID_COUPLING<br>An invalid coupling type was specified in ps3000aSetChannel |
| 46 | PICO_BUFFERS_NOT_SET<br>An attempt was made to get data before a data buffer was defined |
| 47 | PICO_RATIO_MODE_NOT_SUPPORTED<br>The selected downsampling mode (used for data reduction) is not allowed |
| 49 | PICO_INVALID_TRIGGER_PROPERTY<br>An invalid parameter was passed to ps3000aSetTriggerChannelProperties |
| 4A | PICO_INTERFACE_NOT_CONNECTED<br>The driver was unable to contact the oscilloscope |
| 4D | PICO_SIGGEN_WAVEFORM_SETUP_FAILED<br>A problem occurred in ps3000aSetSigGenBuiltIn or ps3000aSetSigGenArbitrary |
| 4E | PICO_FPGA_FAIL |
| 4F | PICO_POWER_MANAGER |
| 50 | PICO_INVALID_ANALOGUE_OFFSET<br>An impossible analog offset value was specified in ps3000aSetChannel |
| 51 | PICO_PLL_LOCK_FAILED<br>Unable to configure the PicoScope |
| 52 | PICO_ANALOG_BOARD<br>The oscilloscope's analog board is not detected, or is not connected to the digital board |
| 53 | PICO_CONFIG_FAIL_AWG<br>Unable to configure the signal generator |
| 54 | PICO_INITIALISE_FPGA<br>The FPGA cannot be initialized, so unit cannot be opened |
| 56 | PICO_EXTERNAL_FREQUENCY_INVALID<br>The frequency for the external clock is not within ±5% of the stated value |
| 57 | PICO_CLOCK_CHANGE_ERROR<br>The FPGA could not lock the clock signal |
| 58 | PICO_TRIGGER_AND_EXTERNAL_CLOCK_CLASH<br>You are trying to configure the AUX input as both a trigger and a reference clock |
| 59 | PICO_PWQ_AND_EXTERNAL_CLOCK_CLASH<br>You are trying to congfigure the AUX input as both a pulse width qualifier and a reference clock |
| 5A | PICO_UNABLE_TO_OPEN_SCALING_FILE<br>The scaling file set can not be opened. |
| 5B | PICO_MEMORY_CLOCK_FREQUENCY<br>The frequency of the memory is reporting incorrectly. |
| 5C | PICO_I2C_NOT_RESPONDING<br>The I2C that is being actioned is not responding to requests. |
| 5D | PICO_NO_CAPTURES_AVAILABLE<br>There are no captures available and therefore no data can be returned. |
| 5E | PICO_NOT_USED_IN_THIS_CAPTURE_MODE |

| | |
|---|---|
| | The capture mode the device is currently running in does not support the current request. |
| 103 | PICO_GET_DATA_ACTIVE<br>Reserved |
| 104 | PICO_IP_NETWORKED<br>The device is currently connected via the IP Network socket and thus the call made is not supported. |
| 105 | PICO_INVALID_IP_ADDRESS<br>An IP address that is not correct has been passed to the driver. |
| 106 | PICO_IPSOCKET_FAILED<br>The IP socket has failed. |
| 107 | PICO_IPSOCKET_TIMEDOUT<br>The IP socket has timed out. |
| 108 | PICO_SETTINGS_FAILED<br>The settings requested have failed to be set. |
| 109 | PICO_NETWORK_FAILED<br>The network connection has failed. |
| 10A | PICO_WS2_32_DLL_NOT_LOADED<br>Unable to load the WS2 dll. |
| 10B | PICO_INVALID_IP_PORT<br>The IP port is invalid |
| 10C | PICO_COUPLING_NOT_SUPPORTED<br>The type of coupling requested is not supported on the opened device. |
| 10D | PICO_BANDWIDTH_NOT_SUPPORTED<br>Bandwidth limit is not supported on the opened device. |
| 10E | PICO_INVALID_BANDWIDTH<br>The value requested for the bandwidth limit is out of range. |
| 10F | PICO_AWG_NOT_SUPPORTED<br>The arbitrary waveform generator is not supported by the opened device. |
| 110 | PICO_ETS_NOT_RUNNING<br>Data has been requested with ETS mode set but run block has not been called, or stop has been called. |
| 111 | PICO_SIG_GEN_WHITENOISE_NOT_SUPPORTED<br>White noise is not supported on the opened device. |
| 112 | PICO_SIG_GEN_WAVETYPE_NOT_SUPPORTED<br>The wave type requested is not supported by the opened device. |
| 113 | PICO_INVALID_DIGITAL_PORT<br>A port number that does not evaluate to either PS3000A_DIGITAL_PORT0 or PS3000A_DIGITAL_PORT1, the ports that are supported. |
| 114 | PICO_INVALID_DIGITAL_CHANNEL<br>The digital channel is not in the range PS3000A_DIGITAL_CHANNEL0 to PS3000_DIGITAL_CHANNEL15, the digital channels that are supported. |
| 115 | PICO_INVALID_DIGITAL_TRIGGER_DIRECTION<br>The digital trigger direction is not a valid trigger direction and should be equal in value to one of the PS3000A_DIGITAL_DIRECTION enumerations. |
| 116 | PICO_SIG_GEN_PRBS_NOT_SUPPORTED<br>Siggen does not generate pseudo-random bit stream. |
| 117 | PICO_ETS_NOT_AVAILABLE_WITH_LOGIC_CHANNELS<br>When a digital port is enabled, ETS sample mode is not available for use. |
| 118 | PICO_WARNING_REPEAT_VALUE<br>Not applicable to this device. |
| 119 | PICO_POWER_SUPPLY_CONNECTED |

| | |
|---|---|
| | 4-Channel only - The DC power supply is connected. |
| 11A | PICO_POWER_SUPPLY_NOT_CONNECTED<br>4-Channel only - The DC power supply isn't connected. |
| 11B | PICO_POWER_SUPPLY_REQUEST_INVALID<br>Incorrect power mode passed for current power source. |
| 11C | PICO_POWER_SUPPLY_UNDERVOLTAGE<br>The supply voltage from the USB source is too low. |
| 11D | PICO_CAPTURING_DATA<br>The oscilloscope is in the process of capturing data. |
| 11E | PICO_USB3_0_DEVICE_NON_USB3_0_PORT<br>A USB 3.0 device is connected to a non-USB 3.0 port. |

## 7.4 Glossary

**Aggregation.** The *ps3000a* driver can use a method called aggregation to reduce the amount of data your application needs to process. This means that for every block of consecutive samples, it stores only the minimum and maximum values. You can set the number of samples in each block, called the aggregation parameter, when you call ps3000aRunStreaming for real-time capture, and when you call ps3000aGetStreamingLatestValues to obtain post-processed data.

**Aliasing.** An effect that can cause digital oscilloscopes to display fast-moving waveforms incorrectly, by showing spurious low-frequency signals ("aliases") that do not exist in the input. To avoid this problem, choose a sampling rate that is at least twice the frequency of the fastest-changing input signal.

**Analog bandwidth.** All oscilloscopes have an upper limit to the range of frequencies at which they can measure accurately. The analog bandwidth of an oscilloscope is defined as the frequency at which a measured sine wave has half the power (or about 71% of the amplitude) of the input sine wave.

**AWG.** Arbitrary waveform generator. On selected models, the signal generator output marked **Gen** or **AWG** can produce an arbitrary waveform defined by the user. Define this waveform by calling ps3000aSetSigGenArbitrary and related functions.

**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 is high frequency. 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.

**ETS.** Equivalent Time Sampling. ETS constructs a representation of a repetitive signal by accumulating information over many similar cycles. This allows the oscilloscope to 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**. It can be used as a signal to start data capture, but not as an analog input.

**Flexible power.** The 4-channel 3000 Series oscilloscopes can be powered by either the USB port or the power supply supplied. A two-headed USB cable, available separately, can be used to obtain power from two USB ports.

**Maximum sampling rate.** The maximum number of samples the oscilloscope is capable of acquiring per second. Maximum sample rates are given in MS/s (megasamples 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.

**Overvoltage.** Any input voltage to the oscilloscope must not exceed the overvoltage limit, measured with respect to ground, otherwise the oscilloscope may be permanently damaged.

**Signal generator.** This is a feature of some oscilloscopes which allows a signal to be generated without an external input device being present. The signal generator output is the BNC socket marked **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 or triangle wave that can be swept back and forth.

**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.

**USB 1.1.** USB (Universal Serial Bus) is a standard port that enables you to connect external devices to PCs. A USB 1.1 port supports data transfer rates up to 12 megabits per second, much faster than an RS-232 port.

**USB 2.0.** A USB 2.0 port supports data transfer rates up to 480 Mbps and is backward-compatible with USB 1.1.

**USB 3.0.** A USB 3.0 port supports data transfer rates up to 5 Gbps and is backward-compatible with USB 2.0 and USB 1.1.

**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 measured by the oscilloscope.


pico
Technology

# Index

## A



## B



## C



## D



## E



## F



# G



# H



# I



# L



# M



# N



# O



# P



# R



# S



# T



# U



# V



# W




pico
Technology

**UK 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

**US headquarters**

Pico Technology
320 N Glenwood Blvd
Tyler
TX 75702
United States of America

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

www.picotech.com

ps3000apg.en-15