



# PicoScope® 6000 Series

## PC Oscilloscopes

### Programmer's Guide



# Contents

1 Introduction .....	7
1 Welcome .....	7
2 Software license conditions .....	8
3 Trademarks .....	8
2 Programming overview .....	9
1 System requirements .....	9
2 Driver .....	10
3 Voltage ranges .....	10
4 Triggering .....	10
5 Sampling modes .....	11
1 Block mode .....	12
2 Rapid block mode .....	14
3 ETS (Equivalent Time Sampling) .....	19
4 Streaming mode .....	21
5 Retrieving stored data .....	22
6 Oversampling .....	24
7 Timebases .....	24
8 Combining several oscilloscopes .....	25
3 API functions .....	26
1 ps6000BlockReady .....	27
2 ps6000CloseUnit .....	28
3 ps6000DataReady .....	29
4 ps6000EnumerateUnits .....	30
5 ps6000FlashLed .....	31
6 ps6000GetAnalogueOffset .....	32
7 ps6000GetMaxDownSampleRatio .....	33
8 ps6000GetNoOfCaptures .....	34
9 ps6000GetNoOfProcessedCaptures .....	35
10 ps6000GetStreamingLatestValues .....	36
11 ps6000GetTimebase .....	37
12 ps6000GetTimebase2 .....	39
13 ps6000GetTriggerTimeOffset .....	40
14 ps6000GetTriggerTimeOffset64 .....	41
15 ps6000GetUnitInfo .....	42
16 ps6000GetValues .....	43
1 Downsampling modes .....	44
17 ps6000GetValuesAsync .....	45
18 ps6000GetValuesBulk .....	46
19 ps6000GetValuesBulkAsync .....	47

20 ps6000GetValuesOverlapped .....	48
1 Using the GetValuesOverlapped functions .....	48
21 ps6000GetValuesOverlappedBulk .....	50
22 ps6000GetValuesTriggerTimeOffsetBulk .....	51
23 ps6000GetValuesTriggerTimeOffsetBulk64 .....	52
24 ps6000IsReady .....	53
25 ps6000IsTriggerOrPulseWidthQualifierEnabled .....	54
26 ps6000MemorySegments .....	55
27 ps6000NoOfStreamingValues .....	56
28 ps6000OpenUnit .....	57
29 ps6000OpenUnitAsync .....	58
30 ps6000OpenUnitProgress .....	59
31 ps6000PingUnit .....	60
32 ps6000RunBlock .....	61
33 ps6000RunStreaming .....	63
34 ps6000SetChannel .....	65
35 ps6000SetDataBuffer .....	68
36 ps6000SetDataBufferBulk .....	69
37 ps6000SetDataBuffers .....	70
38 ps6000SetDataBuffersBulk .....	71
39 ps6000SetEts .....	72
40 ps6000SetEtsTimeBuffer .....	73
41 ps6000SetEtsTimeBuffers .....	74
42 ps6000SetExternalClock .....	75
43 ps6000SetNoOfCaptures .....	76
44 ps6000SetPulseWidthQualifier .....	77
1 PS6000_PWQ_CONDITIONS structure .....	79
45 ps6000SetSigGenArbitrary .....	80
1 Calculating deltaPhase .....	82
2 Index modes .....	82
46 ps6000SetSigGenBuiltIn .....	84
47 ps6000SetSigGenBuiltInV2 .....	87
48 ps6000SetSimpleTrigger .....	88
49 ps6000SetTriggerChannelConditions .....	89
1 PS6000_TRIGGER_CONDITIONS structure .....	90
50 ps6000SetTriggerChannelDirections .....	91
51 ps6000SetTriggerChannelProperties .....	92
1 TRIGGER_CHANNEL_PROPERTIES structure .....	93
52 ps6000SetTriggerDelay .....	94
53 ps6000SigGenArbitraryMinMaxValues .....	95
54 ps6000SigGenFrequencyToPhase .....	96
55 ps6000SigGenSoftwareControl .....	97
56 ps6000Stop .....	98

---

57 ps6000StreamingReady .....	99
58 Wrapper functions .....	100
4 Programming support and examples .....	102
5 Numeric data types .....	103
6 Enumerated types and constants .....	104
7 Driver status codes .....	105
8 Glossary .....	106
Index .....	107

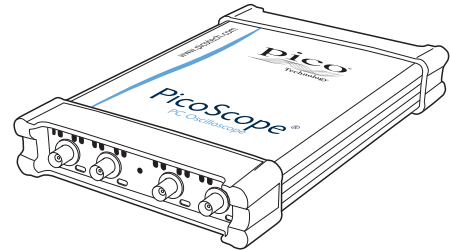


# 1 Introduction

## 1.1 Welcome

The **PicoScope 6000 Series** of oscilloscopes from Pico Technology is a range of compact high-performance units designed to replace traditional benchtop oscilloscopes and digitizers.

This manual explains how to use the Application Programming Interface (API) for the PicoScope 6000 Series scopes. For more information on the hardware, see the *PicoScope 6000 Series User's Guide* and *PicoScope 6000 A/B/C/D Series User's Guide* available separately.



## 1.2 Software license conditions

The material contained in this release is licensed, not sold. Pico Technology Limited 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 these conditions and agree to abide by them.

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

**Copyright.** Pico Technology Ltd. claims the copyright of, and retains the rights to, all material (software, documents, etc.) contained in this software development kit (SDK) except the example programs. 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 Technology and its agents shall not be liable for any loss, damage or injury, howsoever caused, related to the use of Pico Technology equipment or software, unless excluded by statute.

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

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

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

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

**Upgrades.** We provide upgrades, free of charge, from our web site at [www.picotech.com](http://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**, **Excel** and **Visual Basic for Applications** are registered trademarks or trademarks of Microsoft Corporation in the USA and other countries. **LabVIEW** is a registered trademark of National Instruments Corporation. **MATLAB** is a registered trademark of The MathWorks, Inc.



## 2 Programming overview

The `ps6000.dll` dynamic link library in the `lib` subdirectory of your Pico Technology SDK installation directory allows you to program a [PicoScope 6000 Series 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 [sample 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 System requirements

#### Using with PicoScope for Windows

To ensure that your [PicoScope 6000 Series](#) PC Oscilloscope 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 multi-core processor.

Item	Specification
<b>Operating system</b>	Windows 7, Windows 8 or Windows 10 32-bit and 64-bit versions supported
<b>Processor</b>	As required by the operating system
<b>Memory</b>	
<b>Free disk space</b>	
<b>Ports</b>	<a href="#">USB 1.1</a> compliant port* <a href="#">USB 2.0</a> compliant port (recommended for 6000 and 6000A/B Series) <a href="#">USB 3.0</a> compliant port (recommended for 6000C/D Series)

\* The oscilloscope will run slowly on a USB 1.1 port. This configuration is not recommended.

#### Using with custom applications

32-bit and 64-bit drivers are available for Windows. The 32-bit drivers will also run in 32-bit mode on 64-bit operating systems.

#### USB

The PicoScope 6000 Series driver offers [three different methods](#) of recording data, all of which support USB 1.1, USB 2.0, and USB 3.0. Currently only the C and D models are able to make use of the fastest transfer rates via USB 3.0. For other models, either USB 2.0 or USB 3.0 can be used for optimal speed.

## 2.2 Driver

Your application will communicate with a PicoScope 6000 API driver called `ps6000.dll`, which is supplied in 32-bit and 64-bit versions. The driver exports the PicoScope 6000 [function definitions](#) in standard C format, but this does not limit you to programming in C. You can use the API with any programming language that supports standard C calls.

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

## 2.3 Voltage ranges

Using the [ps6000SetChannel](#) function, you can set the oscilloscope input channels to the following voltage ranges:

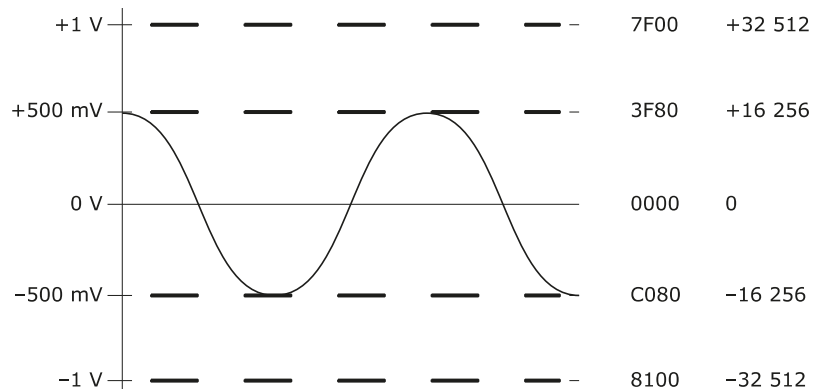
<b>PicoScope 6407</b>	$\pm 100$ mV
<b>All other PicoScope 6000 Series models</b>	$\pm 50$ mV to $\pm 20$ V (1 M $\Omega$ input) $\pm 50$ mV to $\pm 5$ V (50 $\Omega$ input)

Each sample is scaled to 16 bits so that the values returned to your application are as follows:

Constant	Voltage	Value returned	
		decimal	hex
PS6000_MAX_VALUE	maximum	32 512	7F00
	zero	0	0000
PS6000_MIN_VALUE	minimum	-32 512	8100

### Example

1. Call [ps6000SetChannel](#) with range set to PS6000\_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.



Trigger thresholds for the channel inputs are also scaled as above. The AUX trigger input has a fixed range of -1 V to +1 V.

## 2.4 Triggering

PicoScope 6000 Series PC 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 functions:

- [ps6000SetTriggerChannelConditions](#)
- [ps6000SetTriggerChannelDirections](#)
- [ps6000SetTriggerChannelProperties](#)
- [ps6000SetTriggerDelay](#) (optional)

These can be run collectively by calling [ps6000SetSimpleTrigger](#), or singly.

A trigger event can occur when one of the 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:

- [ps6000SetPulseWidthQualifierConditions](#)

## 2.5 Sampling modes

[PicoScope 6000 Series 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 run is started in the same [segment](#), the settings are changed, or the scope is powered down.
- **ETS mode.** In this mode, it is possible to increase the effective sampling rate of the scope when capturing repetitive signals. It is a modified form of [block mode](#).
- **Rapid block mode.** This is a variant of block mode that allows you to capture more than one waveform at a time with a minimum of delay between captures. You can use downsampling in this mode if you wish.
- **Streaming mode.** In this mode, data is passed directly to the PC without being stored in the scope's buffer memory. This enables long periods of slow data collection for chart recorder and data-logging applications. Streaming mode also provides fast streaming at up to 13.33 MS/s (75 ns per sample) with USB 2.0 or 156.25 MS/s with USB 3.0. Downsampling and triggering are supported in this mode.

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 block mode, you can also poll the driver instead of using a callback.

## 2.5.1 Block mode

In **block mode**, the computer prompts a [PicoScope 6000 series](#) oscilloscope to collect a block of data into its internal memory. When the oscilloscope has collected the whole block, it signals that it is ready and then transfers the whole block to the computer's memory through the USB port.

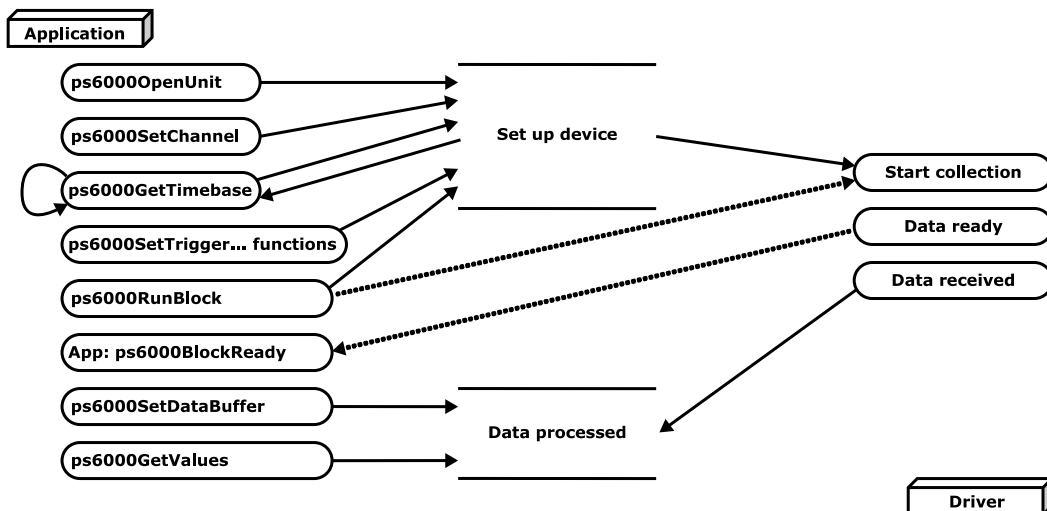
- **Block size.** The maximum number of values depends upon the size of the oscilloscope's memory. The memory buffer is shared between the enabled channels, so if two channels are enabled, each receives half the memory. These features are handled transparently by the driver. The block size also depends on the number of memory segments in use (see [ps6000MemorySegments](#)).
- **Sampling rate.** A PicoScope 6000 Series 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 6000 Series User's Guide](#) 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 [ps6000RunBlock](#), [ps6000Stop](#) and [ps6000GetValues](#).
- **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 [ps6000MemorySegments](#).
- **Data retention.** The data is lost when a new run is started in the same segment, the settings are changed, or the scope is powered down.

See [Using block mode](#) for programming details.

### 2.5.1.1 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 [ps6000OpenUnit](#).
2. Select channel ranges and AC/DC coupling using [ps6000SetChannel](#).
3. Using [ps6000GetTimebase](#), select timebases until the required nanoseconds per sample is located.
4. Use the trigger setup functions [ps6000SetTriggerChannelConditions](#), [ps6000SetTriggerChannelDirections](#) and [ps6000SetTriggerChannelProperties](#) to set up the trigger if required.
5. Start the oscilloscope running using [ps6000RunBlock](#).
6. Wait until the oscilloscope is ready using the [ps6000BlockReady](#) callback (or poll using [ps6000IsReady](#)).
7. Use [ps6000SetDataBuffer](#) to tell the driver where your memory buffer is. For greater efficiency with multiple captures, you can do this outside the loop after step 4.
8. Transfer the block of data from the oscilloscope using [ps6000GetValues](#).
9. Display the data.
10. Repeat steps 5 to 9.
11. Stop the oscilloscope using [ps6000Stop](#).
12. Request new views of stored data using different downsampling parameters: see [Retrieving stored data](#).
13. Close the device using [ps6000CloseUnit](#).



### 2.5.1.2 Asynchronous calls in block mode

The [ps6000GetValues](#) function may take a long time to complete if a large amount of data is being collected. For example, it can take about a minute to retrieve the full 2 billion samples from a PicoScope 6404D over a USB 2.0 connection or a few seconds over USB 3.0. To avoid hanging the calling thread, it is possible to call [ps6000GetValuesAsync](#) instead. This immediately returns control to the calling thread, which then has the option of waiting for the data or calling [ps6000Stop](#) to abort the operation.

## 2.5.2 Rapid block mode

In normal [block mode](#), the PicoScope 6000 Series scopes collect one waveform at a time. You start 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 1 microsecond.

See [Using rapid block mode](#) for details.

### 2.5.2.1 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 [ps6000OpenUnit](#).
2. Select channel ranges and AC/DC coupling using [ps6000SetChannel](#).
3. Set the number of memory segments equal to or greater than the number of captures required using [ps6000MemorySegments](#). Use [ps6000SetNoOfCaptures](#) before each run to specify the number of waveforms to capture.
4. Using [ps6000GetTimebase](#), select timebases until the required nanoseconds per sample is located.
5. Use the trigger setup functions [ps6000SetTriggerChannelConditions](#), [ps6000SetTriggerChannelDirections](#) and [ps6000SetTriggerChannelProperties](#) to set up the trigger if required.
6. Start the oscilloscope running using [ps6000RunBlock](#).
7. Wait until the oscilloscope is ready using the [ps6000BlockReady](#) callback.
8. Use [ps6000SetDataBufferBulk](#) 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 with multiple captures, you could do this outside the loop after step 5.
9. Transfer the blocks of data from the oscilloscope using [ps6000GetValuesBulk](#).
10. Retrieve the time offset for each data segment using [ps6000GetValuesTriggerTimeOffsetBulk64](#).
11. Display the data.
12. Repeat steps 6 to 11 if necessary.
13. Stop the oscilloscope using [ps6000Stop](#).
14. Close the device using [ps6000CloseUnit](#).

#### With aggregation

To use rapid block mode with aggregation, follow steps 1 to 7 above and then proceed as follows:

- 8a. Call [ps6000SetDataBuffersBulk](#) to set up one pair of buffers for every waveform segment required.
- 9a. Call [ps6000GetValuesBulk](#) for each pair of buffers.
- 10a. Retrieve the time offset for each data segment using [ps6000GetValuesTriggerTimeOffsetBulk64](#).

Continue from step 11 above.

## 2.5.2.2 Rapid block mode example 1: no aggregation

```
#define MAX_WAVEFORMS 100
#define MAX_SAMPLES 1000
```

Set up the device up as usual.

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

```
// set the number of waveforms to MAX_WAVEFORMS
ps6000SetNoOfCaptures (handle, MAX_WAVEFORMS);

pParameter = false;
ps6000RunBlock
(
    handle,
    0, // noOfPreTriggerSamples
    10000, // noOfPostTriggerSamples
    1, // timebase to be used
    1, // oversample
    &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 = PS6000_CHANNEL_A; c <= PS6000_CHANNEL_D; c++)
    {
        ps6000SetDataBufferBulk
        (
            handle,
            c,
            buffer[c][i],
            MAX_SAMPLES,
            i
        );
    }
}
```

Comments: buffer has been created as a two-dimensional array of pointers to `uint16_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.

```
ps6000GetValuesBulk
(
    handle,
    &noOfSamples, // set to MAX_SAMPLES on entering the function
    10, // fromSegmentIndex
    19, // toSegmentIndex
    1, // downsampling ratio
    PS6000_RATIO_MODE_NONE, // downsampling ratio mode
    overflow // indices 10 to 19 will be populated
)
```

Comments: the number of samples could be up to `noOfPreTriggerSamples + noOfPostTriggerSamples`, the values set in [ps6000RunBlock](#). The samples are always returned from the first sample taken, unlike the [ps6000GetValues](#) function which allows the sample index to be set. This function does not support aggregation. The above segments start at 10 and finish at 19 inclusive. It is possible for the `fromSegmentIndex` to wrap around to the `toSegmentIndex`, by setting the `fromSegmentIndex` to 98 and the `toSegmentIndex` to 7.

```
ps6000GetValuesTriggerTimeOffsetBulk64
(
    handle,
    times,
    timeUnits,
    10,
    19
)
```

Comments: the above segments start at 10 and finish at 19 inclusive. It is possible for the `fromSegmentIndex` to wrap around to the `toSegmentIndex`, if the `fromSegmentIndex` is set to 98 and the `toSegmentIndex` to 7.



### 2.5.2.3 Rapid block mode example 2: using aggregation

```
#define MAX_WAVEFORMS 100
#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 MAX_WAVEFORMS
ps6000SetNoOfCaptures (handle, MAX_WAVEFORMS);

pParameter = false;
ps6000RunBlock
(
    handle,
    0, //noOfPreTriggerSamples,
    1000000, // noOfPostTriggerSamples,
    1, // timebase to be used,
    1, // oversample
    &timeIndisposedMs,
    0, // segmentIndex
    lpReady,
    &pParameter
);
```

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

```
for (int32_t c = PS6000_CHANNEL_A; c <= PS6000_CHANNEL_D; c++)
{
    ps6000SetDataBuffers
    (
        handle,
        c,
        bufferMax[c],
        bufferMin[c],
        MAX_SAMPLES,
        PS6000_RATIO_MODEAggregate
    );
}
```

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.

```
for (int32_t segment = 10; segment < 20; segment++)
{
    ps6000GetValues
    (
        handle,
        0,
        &noOfSamples, // set to MAX_SAMPLES on entering
        1000,
        &downSampleRatioMode, //set to RATIO_MODE_AGGREGATE
        index,
        overflow
    );

    ps6000GetTriggerTimeOffset64
    (
        handle,
        &time,
        &timeUnits,
        index
    )
}
```

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

## 2.5.3 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 [ps6000SetTrigger](#) and [ps6000SetEts](#) functions.

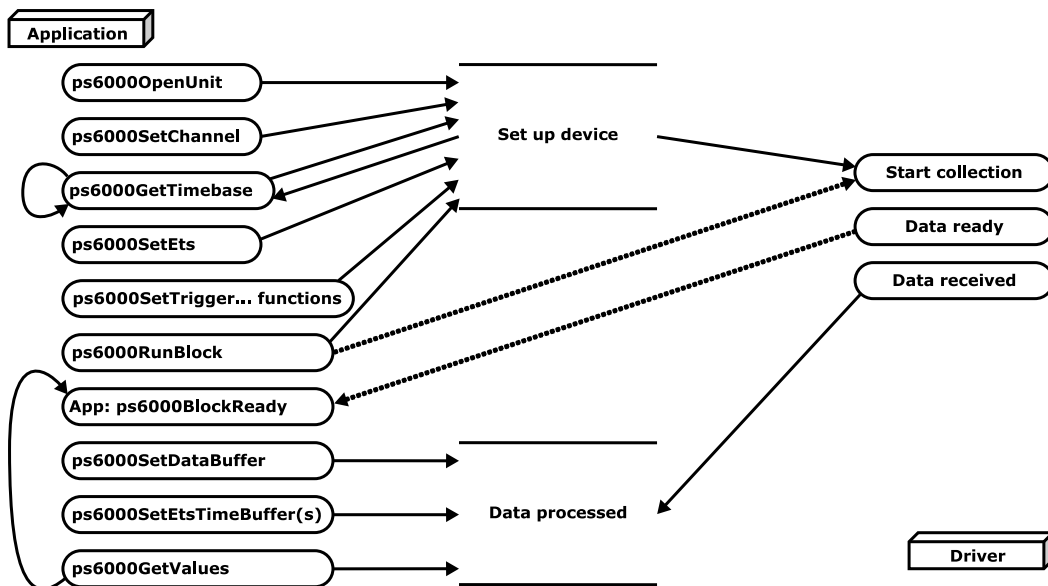
- **Overview.** ETS works by capturing several cycles of a repetitive waveform, then combining them to produce a composite waveform that has a higher effective sampling rate than the individual captures. The scope hardware accurately measures the delay, which is a small fraction of a single sampling interval, between each trigger event and the subsequent sample. The driver then shifts each capture slightly in time and overlays them so that the trigger points are exactly lined up. The result is a larger set of samples spaced by a small fraction of the original sampling interval. The maximum effective sampling rates that can be achieved with this method are listed in the data sheet 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 returns data to your application using the [ps6000BlockReady](#) callback function.

<b>Applicability</b>	Available in <a href="#">block mode</a> only.  Not suitable for one-shot (non-repetitive) signals.  <a href="#">Aggregation</a> and <a href="#">oversampling</a> are not supported.  <a href="#">Edge-triggering</a> only.  <a href="#">Auto trigger delay</a> ( <code>autoTriggerMilliseconds</code> ) is ignored.  Only supports <a href="#">timebases</a> 0, 1 and 2.
----------------------	--

### 2.5.3.1 Using ETS mode

This is the general procedure for reading and displaying data in [ETS mode](#) using a single [memory segment](#):

1. Open the oscilloscope using [ps6000OpenUnit](#).
2. Select channel ranges and AC/DC coupling using [ps6000SetChannel](#).
3. Use [ps6000GetTimebase](#) to verify the number of samples to be collected.
4. Set up ETS using [ps6000SetEts](#).
5. Use the trigger setup functions [ps6000SetTriggerChannelConditions](#), [ps6000SetTriggerChannelDirections](#) and [ps6000SetTriggerChannelProperties](#) to set up the trigger if required.
6. Start the oscilloscope running using [ps6000RunBlock](#).
7. Wait until the oscilloscope is ready using the [ps6000BlockReady](#) callback (or poll using [ps6000IsReady](#)).
8. Use [ps6000SetDataBuffer](#) to tell the driver where to store sampled data.
- 8a. Use [ps6000SetEtsTimeBuffer](#) or [ps6000SetEtsTimeBuffers](#) to tell the driver where to store sample times.
9. Transfer the block of data from the oscilloscope using [ps6000GetValues](#).
10. Display the data.
11. While you want to collect updated captures, repeat steps 7 to 10.
12. Stop the oscilloscope using [ps6000Stop](#).
13. Repeat steps 6 to 12.
14. Close the device using [ps6000CloseUnit](#).



## 2.5.4 Streaming mode

**Streaming mode** can capture data without the gaps that occur between blocks when using [block mode](#).

With USB 2.0 it can transfer data to the PC at speeds of at least 13.33 million samples per second (75 nanoseconds per sample), depending on the computer's performance. With USB 3.0 this speed increases to 156.25 MS/s. This makes it suitable for **high-speed data acquisition**, allowing you to capture long data sets limited only by the computer's memory.

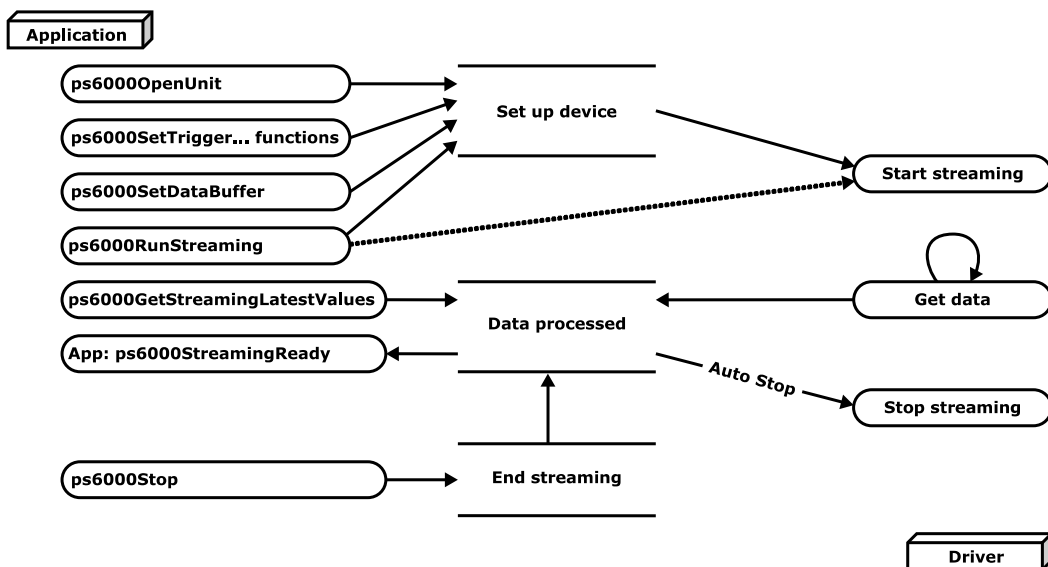
- **Aggregation.** The driver returns [aggregated readings](#) while the device is streaming. If aggregation is set to 1 then only one buffer is returned per channel. When aggregation is set above 1 then two buffers (maximum and minimum) per channel are returned.
- **Memory segmentation.** The memory 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.

### 2.5.4.1 Using streaming mode

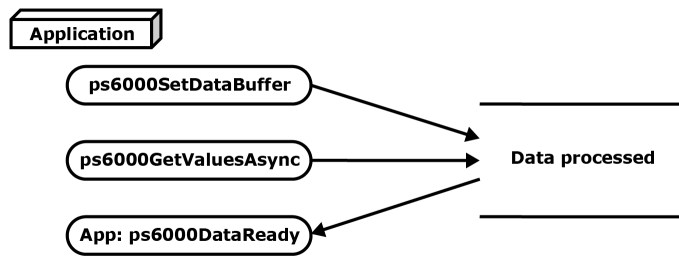
This is the general procedure for reading and displaying data in [streaming mode](#) using a single [memory segment](#):

1. Open the oscilloscope using [ps6000OpenUnit](#).
2. Select channels, ranges and AC/DC coupling using [ps6000SetChannel](#).
3. Use the trigger setup functions [ps6000SetTriggerChannelConditions](#), [ps6000SetTriggerChannelDirections](#) and [ps6000SetTriggerChannelProperties](#) to set up the trigger if required.
4. Call [ps6000SetDataBuffer](#) to tell the driver where your data buffer is.
5. Set up aggregation and start the oscilloscope running using [ps6000RunStreaming](#).
6. Call [ps6000GetStreamingLatestValues](#) to get data.
7. Process data returned to your application's function. This example is using `autoStop`, so after the driver has received all the data points requested by the application, it stops the device streaming.
8. Call [ps6000Stop](#), even if `autoStop` is enabled.
9. Request new views of stored data using different downsampling parameters: see [Retrieving stored data](#).
10. Close the device using [ps6000CloseUnit](#).



### 2.5.5 Retrieving stored data

You can collect data from the PicoScope 6000 driver with a different [downsampling](#) factor when [ps6000RunBlock](#) or [ps6000RunStreaming](#) has already been called and has successfully captured all the data. Use [ps6000GetValuesAsync](#).



## 2.6 Oversampling

*Note: This feature is provided for backward compatibility only. The same effect can be obtained more efficiently with the PicoScope 6000 Series using the hardware averaging feature (see [Downsampling modes](#)).*

When the oscilloscope is operating at sampling rates less than its maximum, it is possible to **oversample**. Oversampling is taking more than one measurement during a time interval and returning the average as one sample. The number of measurements per sample is called the oversampling factor. If the signal contains a small amount of wideband noise (strictly speaking, *Gaussian noise*), this technique can increase the effective [vertical resolution](#) of the oscilloscope by  $n$  bits, where  $n$  is given approximately by the equation below:

$$n = \log(\text{oversampling factor}) / \log 4$$

Conversely, for an improvement in resolution of  $n$  bits, the oversampling factor you need is given approximately by:

$$\text{oversampling factor} = 4^n$$

An oversample of 4, for example, would quadruple the time interval and quarter the maximum samples, and at the same time would increase the effective resolution by one bit.

<b>Applicability</b>	Available in <a href="#">block mode</a> only. Cannot be used at the same time as <a href="#">downsampling</a> .
----------------------	--

## 2.7 Timebases

The API allows you to select any of  $2^{32}$  different timebases based on a maximum sampling rate of 5 GHz. 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](#).

timebase	sample interval formula	sample interval examples
0 to 4	$2^{\text{timebase}} / 5\,000\,000\,000$	0 => 200 ps 1 => 400 ps 2 => 800 ps 3 => 1.6 ns 4 => 3.2 ns
5 to $2^{32}-1$	$(\text{timebase}-4) / 156\,250\,000$	5 => 6.4 ns ... $2^{32}-1$ => ~ 6.87 s

<b>Applicability</b>	Call either <a href="#">ps6000GetTimebase</a> or <a href="#">ps6000GetTimebase2</a> . Note that <a href="#">ps6000GetTimebase</a> should not be used for timebases 0, 1 or 2. <a href="#">ETS mode</a> only supports timebases 0, 1 and 2: see <a href="#">ps6000SetEts</a> for more information.
----------------------	---

### Notes

1. The maximum possible sampling rate may depend on the number of enabled channels and on the sampling mode: please refer to the data sheet for details.
2. In [streaming mode](#), the speed of the USB port may affect the rate of data transfer.



## 2.8 Combining several oscilloscopes

It is possible to collect data using up to 64 [PicoScope 6000 Series oscilloscopes](#) at the same time, depending on the capabilities of the PC. Each oscilloscope must be connected to a separate USB port. The [ps6000OpenUnit](#) function returns a handle to an oscilloscope. All the other functions require this handle for oscilloscope identification. For example, to collect data from two oscilloscopes at the same time:

```
CALLBACK ps6000BlockReady(...)
// define callback function specific to application

handle1 = ps6000OpenUnit
handle2 = ps6000OpenUnit

ps6000SetChannel(handle1)
// set up unit 1
ps6000RunBlock(handle1)

ps6000SetChannel(handle2)
// set up unit 2
ps6000RunBlock(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
```

Note: an [external clock](#) may be fed into the AUX input to provide some degree of synchronization between multiple oscilloscopes.

## 3 API functions

The PicoScope 6000 Series 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.

<a href="#"><u>ps6000BlockReady</u></a>	indicate when block-mode data ready
<a href="#"><u>ps6000CloseUnit</u></a>	close a scope device
<a href="#"><u>ps6000DataReady</u></a>	indicate when post-collection data ready
<a href="#"><u>ps6000EnumerateUnits</u></a>	find all connected oscilloscopes
<a href="#"><u>ps6000FlashLed</u></a>	flash the front-panel LED
<a href="#"><u>ps6000GetAnalogueOffset</u></a>	get min/max allowable analog offset
<a href="#"><u>ps6000GetMaxDownSampleRatio</u></a>	find out aggregation ratio for data
<a href="#"><u>ps6000GetStreamingLatestValues</u></a>	get streaming data while scope is running
<a href="#"><u>ps6000GetTimebase</u></a>	find out what timebases are available
<a href="#"><u>ps6000GetTimebase2</u></a>	find out what timebases are available
<a href="#"><u>ps6000GetTriggerTimeOffset</u></a>	find out when trigger occurred (32-bit)
<a href="#"><u>ps6000GetTriggerTimeOffset64</u></a>	find out when trigger occurred (64-bit)
<a href="#"><u>ps6000GetUnitInfo</u></a>	read information about scope device
<a href="#"><u>ps6000GetValues</u></a>	get block-mode data with callback
<a href="#"><u>ps6000GetValuesAsync</u></a>	get streaming data with callback
<a href="#"><u>ps6000GetValuesBulk</u></a>	get data in rapid block mode
<a href="#"><u>ps6000GetValuesBulkAsync</u></a>	get data in rapid block mode using callback
<a href="#"><u>ps6000GetValuesOverlapped</u></a>	set up data collection ahead of capture
<a href="#"><u>ps6000GetValuesOverlappedBulk</u></a>	set up data collection in rapid block mode
<a href="#"><u>ps6000GetValuesTriggerTimeOffsetBulk</u></a>	get rapid-block waveform timings (32-bit)
<a href="#"><u>ps6000GetValuesTriggerTimeOffsetBulk64</u></a>	get rapid-block waveform timings (64-bit)
<a href="#"><u>ps6000IsReady</u></a>	poll driver in block mode
<a href="#"><u>ps6000IsTriggerOrPulseWidthQualifierEnabled</u></a>	find out whether trigger is enabled
<a href="#"><u>ps6000MemorySegments</u></a>	divide scope memory into segments
<a href="#"><u>ps6000NoOfStreamingValues</u></a>	get number of samples in streaming mode
<a href="#"><u>ps6000OpenUnit</u></a>	open a scope device
<a href="#"><u>ps6000OpenUnitAsync</u></a>	open a scope device without waiting
<a href="#"><u>ps6000OpenUnitProgress</u></a>	check progress of OpenUnit call
<a href="#"><u>ps6000RunBlock</u></a>	start block mode
<a href="#"><u>ps6000RunStreaming</u></a>	start streaming mode
<a href="#"><u>ps6000SetChannel</u></a>	set up input channels
<a href="#"><u>ps6000SetDataBuffer</u></a>	register data buffer with driver
<a href="#"><u>ps6000SetDataBufferBulk</u></a>	set the buffers for each waveform
<a href="#"><u>ps6000SetDataBuffers</u></a>	register aggregated data buffers with driver
<a href="#"><u>ps6000SetDataBuffersBulk</u></a>	register data buffers for rapid block mode
<a href="#"><u>ps6000SetEts</u></a>	set up equivalent-time sampling
<a href="#"><u>ps6000SetEtsTimeBuffer</u></a>	set up buffer for ETS timings (64-bit)
<a href="#"><u>ps6000SetEtsTimeBuffers</u></a>	set up buffer for ETS timings (32-bit)
<a href="#"><u>ps6000SetExternalClock</u></a>	set AUX input to receive external clock
<a href="#"><u>ps6000SetNoOfCaptures</u></a>	set number of captures to collect in one run
<a href="#"><u>ps6000SetPulseWidthQualifier</u></a>	set up pulse width triggering
<a href="#"><u>ps6000SetSigGenArbitrary</u></a>	set up arbitrary waveform generator
<a href="#"><u>ps6000SetSigGenBuiltIn</u></a>	set up signal generator
<a href="#"><u>ps6000SetSigGenBuiltInV2</u></a>	set up signal generator (double precision)
<a href="#"><u>ps6000SetSimpleTrigger</u></a>	set up level triggers only
<a href="#"><u>ps6000SetTriggerChannelConditions</u></a>	specify which channels to trigger on
<a href="#"><u>ps6000SetTriggerChannelDirections</u></a>	set up signal polarities for triggering
<a href="#"><u>ps6000SetTriggerChannelProperties</u></a>	set up trigger thresholds
<a href="#"><u>ps6000SetTriggerDelay</u></a>	set up post-trigger delay
<a href="#"><u>ps6000SigGenArbitraryMinMaxValues</u></a>	get limits for AWG settings
<a href="#"><u>ps6000SigGenFrequencyToPhase</u></a>	calculate delta phase parameter for AWG setup
<a href="#"><u>ps6000SigGenSoftwareControl</u></a>	trigger the signal generator
<a href="#"><u>ps6000Stop</u></a>	stop data capture
<a href="#"><u>ps6000StreamingReady</u></a>	indicate when streaming-mode data ready

## 3.1 ps6000BlockReady

```
typedef void (CALLBACK *ps6000BlockReady)
(
    int16_t      handle,
    PICO\_STATUS   status,
    void         * pParameter
)
```

This [callback](#) function is part of your application. You register it with the PicoScope 6000 Series driver using [ps6000RunBlock](#), and the driver calls it back when block-mode data is ready. You can then download the data using the [ps6000GetValues](#) function.

<b>Applicability</b>	<a href="#">Block mode</a> only
<b>Arguments</b>	<p><code>handle</code>, identifies the device</p> <p><code>status</code>, indicates whether an error occurred during collection of the data.</p> <p><code>pParameter</code>, a void pointer passed from <a href="#">ps6000RunBlock</a>. Your callback function can write to this location to send any data, such as a status flag, back to your application.</p>
<b>Returns</b>	nothing

## 3.2 ps6000CloseUnit

```
PICO\_STATUS ps6000CloseUnit  
(  
    int16_t    handle  
)
```

This function shuts down a PicoScope 6000 Series oscilloscope.

<b>Applicability</b>	All modes
<b>Arguments</b>	<code>handle</code> , the identifier, returned by <a href="#">ps6000OpenUnit</a> , of the scope device to be closed.
<b>Returns</b>	PICO_OK PICO_HANDLE_INVALID PICO_USER_CALLBACK PICO_DRIVER_FUNCTION

## 3.3 ps6000DataReady

```
typedef void (CALLBACK *ps6000DataReady)
(
    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 [ps6000GetValuesAsync](#), and the driver calls your function back when the data is ready.

<b>Applicability</b>	All modes
<b>Arguments</b>	<p><code>handle</code>, identifies the device</p> <p><code>status</code>, a <code>PICO_STATUS</code> code returned by the driver.</p> <p><code>noOfSamples</code>, the number of samples collected.</p> <p><code>overflow</code>, 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.</p> <p><code>pParameter</code>, a void pointer passed from <a href="#">ps6000GetValuesAsync</a>. 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.</p>
<b>Returns</b>	nothing

## 3.4 ps6000EnumerateUnits

[PICO\\_STATUS](#) ps6000EnumerateUnits

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

This function counts the number of PicoScope 6000 units connected to the computer, and returns a list of serial numbers as a string. Note that this function will only detect devices that are not yet being controlled by an application.

<b>Applicability</b>	All modes
<b>Arguments</b>	<p>* <code>count</code>, on exit, the number of PicoScope 6000 units found</p> <p>* <code>serials</code>, 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.</p> <p>* <code>serialLth</code>, on entry, the length of the <code>int8_t</code> buffer pointed to by <code>serials</code>; on exit, the length of the string written to <code>serials</code></p>
<b>Returns</b>	PICO_OK PICO_BUSY PICO_NULL_PARAMETER PICO_FW_FAIL PICO_CONFIG_FAIL PICO_MEMORY_FAIL PICO_ANALOG_BOARD PICO_CONFIG_FAIL_AWG PICO_INITIALISE_FPGA

## 3.5 ps6000FlashLed

```
PICO\_STATUS ps6000FlashLed  
(  
    int16_t    handle,  
    int16_t    start  
)
```

This function flashes the LED on the front of the scope without blocking the calling thread. Calls to [ps6000RunStreaming](#) and [ps6000RunBlock](#) 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.

<b>Applicability</b>	All modes
<b>Arguments</b>	<code>handle</code> , identifies the device  <code>start</code> , the action required: < 0 : flash the LED indefinitely. 0 : stop the LED flashing. > 0 : flash the LED <code>start</code> times. If the LED is already flashing on entry to this function, the flash count will be reset to <code>start</code> .
<b>Returns</b>	PICO_OK PICO_HANDLE_INVALID PICO_BUSY PICO_DRIVER_FUNCTION PICO_NOT_RESPONDING

## 3.6 ps6000GetAnalogueOffset

```
PICO_STATUS ps6000GetAnalogueOffset
(
    int16_t          handle,
    PS6000_RANGE     range
    PS6000_COUPLING  coupling
    float            * maximumVoltage,
    float            * minimumVoltage
)
```

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

<b>Applicability</b>	Not PicoScope 6407
<b>Arguments</b>	<p><code>handle</code>, identifies the device</p> <p><code>range</code>, the voltage range for which minimum and maximum voltages are required</p> <p><code>coupling</code>, the type of AC/DC coupling used</p> <p>* <code>maximumVoltage</code>, on output, the maximum analog offset voltage allowed for the range. Set to <code>NULL</code> if not required.</p> <p>* <code>minimumVoltage</code>, on output, the minimum analog offset voltage allowed for the range. Set to <code>NULL</code> if not required.</p>
<b>Returns</b>	<p><code>PICO_OK</code></p> <p><code>PICO_INVALID_HANDLE</code></p> <p><code>PICO_DRIVER_FUNCTION</code></p> <p><code>PICO_INVALID_VOLTAGE_RANGE</code></p> <p><code>PICO_NULL_PARAMETER</code> (if both <code>maximumVoltage</code> and <code>minimumVoltage</code> are <code>NULL</code>)</p>



## 3.7 ps6000GetMaxDownSampleRatio

```
PICO_STATUS ps6000GetMaxDownSampleRatio
(
    int16_t          handle,
    uint32_t          noOfUnaggregatedSamples,
    uint32_t          * maxDownSampleRatio,
    PS6000_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.

<b>Applicability</b>	All modes
<b>Arguments</b>	<p><code>handle</code>, identifies the device</p> <p><code>noOfUnaggregatedSamples</code>, the number of unprocessed samples to be downsampled</p> <p><code>maxDownSampleRatio</code>, the maximum possible downsampling ratio</p> <p><code>downSampleRatioMode</code>, the downsampling mode. See <a href="#">ps6000GetValues</a>.</p> <p><code>segmentIndex</code>, the <a href="#">memory segment</a> where the data is stored</p>
<b>Returns</b>	PICO_OK PICO_INVALID_HANDLE PICO_NO_SAMPLES_AVAILABLE PICO_NULL_PARAMETER PICO_INVALID_PARAMETER PICO_SEGMENT_OUT_OF_RANGE PICO_TOO_MANY_SAMPLES

## 3.8 ps6000GetNoOfCaptures

```
PICO\_STATUS ps6000GetNoOfCaptures  
(  
    int16_t    handle,  
    uint32_t   * nCaptures  
)
```

This function returns the number of captures collected in one run of [rapid block mode](#). You can call this function during device capture, after collection has completed or after interrupting waveform collection by calling [ps6000Stop](#).

The returned value (`nCaptures`) can then be used to iterate through the number of segments using [ps6000GetValues](#), or in a single call to [ps6000GetValuesBulk](#) where it is used to calculate the `toSegmentIndex` parameter.

<b>Applicability</b>	All modes
<b>Arguments</b>	<code>handle</code> , identifies the device  <code>nCaptures</code> , on output, the number of available captures that has been collected from calling <a href="#">ps6000RunBlock</a>
<b>Returns</b>	PICO_OK PICO_INVALID_HANDLE PICO_NO_SAMPLES_AVAILABLE PICO_NULL_PARAMETER PICO_INVALID_PARAMETER PICO_SEGMENT_OUT_OF_RANGE PICO_TOO_MANY_SAMPLES

## 3.9 ps6000GetNoOfProcessedCaptures

```
PICO\_STATUS ps6000GetNoOfProcessedCaptures  
(  
    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 [ps6000RunBlock](#). It is for use in rapid block mode, alongside the [ps6000GetValuesOverlappedBulk](#) function, when the driver is set to transfer data from the device automatically as soon as the [ps6000RunBlock](#) function is called. You can call [ps6000GetNoOfProcessedCaptures](#) during device capture, after collection has completed or after interrupting waveform collection by calling [ps6000Stop](#).

The returned value (`nProcessedCaptures`) can then be used to iterate through the number of segments using [ps6000GetValues](#), or in a single call to [ps6000GetValuesBulk](#), where it is used to calculate the `toSegmentIndex` parameter.

### When capture is stopped

If `nProcessedCaptures` = 0, you will also need to call [ps6000GetNoOfCaptures](#), in order to determine how many waveform segments were captured, before calling [ps6000GetValues](#) or [ps6000GetValuesBulk](#).

<b>Applicability</b>	<a href="#">Rapid block mode</a>
<b>Arguments</b>	<code>handle</code> , the handle of the device.  * <code>nProcessedCaptures</code> , on exit, the number of waveforms captured and processed.
<b>Returns</b>	PICO_OK PICO_INVALID_HANDLE PICO_INVALID_PARAMETER

## 3.10 ps6000GetStreamingLatestValues

```
PICO\_STATUS ps6000GetStreamingLatestValues  
(  
    int16_t          handle,  
    ps6000StreamingReady lpPs6000Ready,  
    void             * pParameter  
)
```

This function instructs the driver to return the next block of values to your [ps6000StreamingReady](#) callback function. You must have previously called [ps6000RunStreaming](#) beforehand to set up [streaming](#).

<b>Applicability</b>	<a href="#">Streaming</a> mode only
<b>Arguments</b>	<p><code>handle</code>, identifies the device</p> <p><code>lpPs6000Ready</code>, a pointer to your <a href="#">ps6000StreamingReady</a> callback function</p> <p><code>pParameter</code>, a void pointer that will be passed to the <a href="#">ps6000StreamingReady</a> callback function. The callback function may optionally use this pointer to return information to the application.</p>
<b>Returns</b>	<p>PICO_OK PICO_INVALID_HANDLE PICO_NO_SAMPLES_AVAILABLE PICO_INVALID_CALL PICO_BUSY PICO_NOT_RESPONDING PICO_DRIVER_FUNCTION PICO_STARTINDEX_INVALID</p>

## 3.11 ps6000GetTimebase

```
PICO\_STATUS ps6000GetTimebase  
(  
    int16_t      handle,  
    uint32_t     timebase,  
    uint32_t     noSamples,  
    int32_t      * timeIntervalNanoseconds,  
    int16_t      oversample,  
    uint32_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 [ps6000SetChannel](#).

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, then we recommend that you use [ps6000GetTimebase2](#) instead.

To use `ps6000GetTimebase` or [ps6000GetTimebase2](#), first estimate the timebase number that you require using the information in the [timebase guide](#). Pass this timebase to the `GetTimebase` function and check the returned `timeIntervalNanoseconds` argument. If necessary, repeat until you obtain the time interval that you need.

Note that `ps6000GetTimebase` should not be called for timebases 0, 1 or 2, as they will return values smaller than 1 nanosecond.

<b>Applicability</b>	All modes.
<b>Arguments</b>	<p><code>handle</code>, identifies the device.</p> <p><code>timebase</code>, <a href="#">see timebase guide</a>. In ETS mode the driver selects its own timebase and this argument is ignored.</p> <p><code>noSamples</code>, the number of samples required. This value is used to calculate the most suitable time interval.</p> <p><code>timeIntervalNanoseconds</code>, on exit, the time interval between readings at the selected timebase. Use <code>NULL</code> if not required. In ETS mode this argument is not valid; use the sample time returned by <a href="#">ps6000SetEts</a> instead.</p> <p><code>oversample</code>, the amount of <a href="#">oversample</a> required. Range: 0 to <a href="#">PS6000_MAX_OVERSAMPLE_8BIT</a>.</p> <p><code>maxSamples</code>, 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 <code>NULL</code> if not required.</p> <p><code>segmentIndex</code>, the index of the memory segment to use.</p>
<b>Returns</b>	<p>PICO_OK</p> <p>PICO_INVALID_HANDLE</p> <p>PICO_TOO_MANY_SAMPLES</p> <p>PICO_INVALID_CHANNEL</p> <p>PICO_INVALID_TIMEBASE</p> <p>PICO_INVALID_PARAMETER</p> <p>PICO_SEGMENT_OUT_OF_RANGE</p> <p>PICO_DRIVER_FUNCTION</p>

## 3.12 ps6000GetTimebase2

```
PICO\_STATUS ps6000GetTimebase2  
(  
    int16_t      handle,  
    uint32_t      timebase,  
    uint32_t      noSamples,  
    float        * timeIntervalNanoseconds,  
    int16_t      oversample,  
    uint32_t      * maxSamples  
    uint32_t      segmentIndex  
)
```

This function is an upgraded version of [ps6000GetTimebase](#), and returns the time interval as a `float` rather than a `uint32_t`. This allows it to return sub-nanosecond time intervals. See [ps6000GetTimebase](#) for a full description.

Note that [ps6000GetTimebase](#) should not be called for timebases 0, 1 or 2, as they will return values smaller than 1 nanosecond.

<b>Applicability</b>	All modes
<b>Arguments</b>	<code>timeIntervalNanoseconds</code> , a pointer to the time interval between readings at the selected timebase. If a null pointer is passed, nothing will be written here.  All other arguments: see <a href="#">ps6000GetTimebase</a>
<b>Returns</b>	See <a href="#">ps6000GetTimebase</a>

### 3.13 ps6000GetTriggerTimeOffset

```
PICO_STATUS ps6000GetTriggerTimeOffset
(
    int16_t          handle
    uint32_t          * timeUpper
    uint32_t          * timeLower
    PS6000_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, [ps6000GetTriggerTimeOffset64](#), is available that returns the time as a single 64-bit value.

<b>Applicability</b>	<a href="#">Block mode</a> , <a href="#">rapid block mode</a>
<b>Arguments</b>	<p><code>handle</code>, identifies the device</p> <p><code>timeUpper</code>, on exit, the upper 32 bits of the time at which the trigger point occurred</p> <p><code>timeLower</code>, on exit, the lower 32 bits of the time at which the trigger point occurred</p> <p><code>timeUnits</code>, returns the time units in which <code>timeUpper</code> and <code>timeLower</code> are measured. The allowable values are:</p> <p><a href="#">PS6000_FS</a>  <a href="#">PS6000_PS</a>  <a href="#">PS6000_NS</a>  <a href="#">PS6000_US</a>  <a href="#">PS6000_MS</a>  <a href="#">PS6000_S</a></p> <p><code>segmentIndex</code>, the number of the <a href="#">memory segment</a> for which the information is required.</p>
<b>Returns</b>	PICO_OK PICO_INVALID_HANDLE PICO_DEVICE_SAMPLING PICO_SEGMENT_OUT_OF_RANGE PICO_NULL_PARAMETER PICO_NO_SAMPLES_AVAILABLE PICO_DRIVER_FUNCTION



## 3.14 ps6000GetTriggerTimeOffset64

```
PICO_STATUS ps6000GetTriggerTimeOffset64
(
    int16_t          handle,
    int64_t          * time,
    PS6000_TIME_UNITS * timeUnits,
    uint32_t          segmentIndex
)
```

This function gets the trigger time offset for a waveform. It is equivalent to [ps6000GetTriggerTimeOffset](#) except that the time offset is returned as a single 64-bit value instead of two 32-bit values.

<b>Applicability</b>	<a href="#">Block mode</a> , <a href="#">rapid block mode</a>
<b>Arguments</b>	<p><code>handle</code>, identifies the device</p> <p><code>time</code>, on exit, the time at which the trigger point occurred</p> <p><code>timeUnits</code>, on exit, the time units in which time is measured. The possible values are:</p> <p> <a href="#">PS6000_FS</a>  <a href="#">PS6000_PS</a>  <a href="#">PS6000_NS</a>  <a href="#">PS6000_US</a>  <a href="#">PS6000_MS</a>  <a href="#">PS6000_S</a> </p> <p><code>segmentIndex</code>, the number of the <a href="#">memory segment</a> for which the information is required</p>
<b>Returns</b>	PICO_OK PICO_INVALID_HANDLE PICO_DEVICE_SAMPLING PICO_SEGMENT_OUT_OF_RANGE PICO_NULL_PARAMETER PICO_NO_SAMPLES_AVAILABLE PICO_DRIVER_FUNCTION

## 3.15 ps6000GetUnitInfo

```
PICO_STATUS ps6000GetUnitInfo
(
    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, only the driver version and error code are available to explain why the last open unit call failed.

<b>Applicability</b>	All modes
<b>Arguments</b>	<p><code>handle</code>, identifies the device from which information is required. If an invalid <code>handle</code> is passed, the error code from the last unit that failed to open is returned.</p> <p><code>string</code>, on exit, the unit information string selected specified by the <code>info</code> argument. If <code>string</code> is NULL, only <code>requiredSize</code> is returned.</p> <p><code>stringLength</code>, the maximum number of <code>int8_t</code> values that may be written to <code>string</code>.</p> <p><code>requiredSize</code>, on exit, the required length of the <code>string</code> array.</p> <p><code>info</code>, a number specifying what information is required. The possible values are listed in the table below.</p>
<b>Returns</b>	PICO_OK PICO_INVALID_HANDLE PICO_NULL_PARAMETER PICO_INVALID_INFO PICO_INFO_UNAVAILABLE PICO_DRIVER_FUNCTION

info		Example
0	PICO_DRIVER_VERSION - Version number of PicoScope 6000 DLL	1,0,0,1
1	PICO_USB_VERSION - Type of USB connection to device: 1.1, 2.0 or 3.0	3.0
2	PICO_HARDWARE_VERSION - Hardware version of device	1
3	PICO_VARIANT_INFO - Model number of device	6403
4	PICO_BATCH_AND_SERIAL - Batch and serial number of device	KJL87/6
5	PICO_CAL_DATE - Calibration date of device	30Sep09
6	PICO_KERNEL_VERSION - Version of kernel driver	1,1,2,4
7	PICO_DIGITAL_HARDWARE_VERSION - Hardware version of the digital section	1
8	PICO_ANALOGUE_HARDWARE_VERSION - Hardware version of the analog section	1
9	PICO_FIRMWARE_VERSION_1 - Version information of Firmware 1	1,0,0,1
A	PICO_FIRMWARE_VERSION_2 - Version information of Firmware 2	1,0,0,1

## 3.16 ps6000GetValues

```
PICO_STATUS ps6000GetValues
(
    int16_t          handle,
    uint32_t         startIndex,
    uint32_t         * noOfSamples,
    uint32_t         downSampleRatio,
    PS6000_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 [ps6000SetDataBuffer\(\)](#) or [ps6000SetDataBuffers\(\)](#). It blocks the calling function while retrieving data.

<b>Applicability</b>	<a href="#">Block mode</a> , <a href="#">rapid block mode</a>
<b>Arguments</b>	<p><code>handle</code>, identifies the device.</p> <p><code>startIndex</code>, a zero-based index that indicates the start point for data collection. It is measured in sample intervals from the start of the buffer.</p> <p><code>noOfSamples</code>, 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 always starts with the first sample captured.</p> <p><code>downSampleRatio</code>, the <a href="#">downsampling</a> factor that will be applied to the raw data. Must be greater than zero.</p> <p><code>downSampleRatioMode</code>, which <a href="#">downsampling</a> mode to use. The available values are:  <a href="#">PS6000_RATIO_MODE_NONE</a> (<code>downSampleRatio</code> is ignored)  <a href="#">PS6000_RATIO_MODE_AGGREGATE</a>  <a href="#">PS6000_RATIO_MODE_AVERAGE</a>  <a href="#">PS6000_RATIO_MODE_DECIMATE</a></p> <p><code>PS6000_RATIO_MODE_AGGREGATE</code>, <code>PS6000_RATIO_MODE_AVERAGE</code>, and <code>PS6000_RATIO_MODE_DECIMATE</code> are single-bit constants that can be ORed to apply multiple downsampling modes to the same data.</p> <p><code>segmentIndex</code>, the zero-based number of the <a href="#">memory segment</a> where the data is stored.</p> <p><code>overflow</code>, 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.</p>

<b>Returns</b>	PICO_OK PICO_INVALID_HANDLE PICO_NO_SAMPLES_AVAILABLE PICO_DEVICE_SAMPLING PICO_NULL_PARAMETER PICO_SEGMENT_OUT_OF_RANGE PICO_INVALID_PARAMETER PICO_TOO_MANY_SAMPLES PICO_DATA_NOT_AVAILABLE PICO_STARTINDEX_INVALID PICO_INVALID_SAMPLERATIO PICO_INVALID_CALL PICO_NOT_RESPONDING PICO_MEMORY PICO_RATIO_MODE_NOT_SUPPORTED PICO_DRIVER_FUNCTION
----------------	--

### 3.16.1 Downsampling modes

Various methods of data reduction, or **downsampling**, are possible with the PicoScope 6000 Series 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 [ps6000GetValues](#). The following modes are available:

PS6000_RATIO_MODE_NONE	No downsampling. Returns the raw data values.
PS6000_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.
PS6000_RATIO_MODE_AVERAGE	Reduces every block of $n$ values to a single value representing the average (arithmetic mean) of all the values.
PS6000_RATIO_MODE_DECIMATE	Reduces every block of $n$ values to just the first value in the block, discarding all the other values.
PS6000_RATIO_MODE_DISTRIBUTION	Not implemented.

## 3.17 ps6000GetValuesAsync

```
PICO_STATUS ps6000GetValuesAsync
(
    int16_t          handle,
    uint32_t          startIndex,
    uint32_t          noOfSamples,
    uint32_t          downSampleRatio,
    PS6000_RATIO_MODE downSampleRatioMode,
    uint32_t          segmentIndex,
    void              * lpDataReady,
    void              * pParameter
)
```

This function returns data, with [downsampling](#) if requested, starting at the specified sample number. In streaming mode it retrieves stored data from the driver after data collection has stopped. In block mode it retrieves data from the oscilloscope. It returns the data using a [callback](#).

<b>Applicability</b>	<a href="#">Streaming mode</a> and <a href="#">block mode</a>
<b>Arguments</b>	<p>handle, startIndex, noOfSamples, downSampleRatio, downSampleRatioMode, segmentIndex: see <a href="#">ps6000GetValues</a></p> <p>lpDataReady, a pointer to the user-supplied function that will be called when the data is ready. This will be a <a href="#">ps6000DataReady</a> function for block-mode data or a <a href="#">ps6000StreamingReady</a> function for streaming-mode data.</p> <p>pParameter, a void pointer that will be passed to the callback function. The data type is determined by the application.</p>
<b>Returns</b>	PICO_OK PICO_INVALID_HANDLE PICO_NO_SAMPLES_AVAILABLE PICO_DEVICE_SAMPLING PICO_NULL_PARAMETER PICO_STARTINDEX_INVALID PICO_SEGMENT_OUT_OF_RANGE PICO_INVALID_PARAMETER PICO_DATA_NOT_AVAILABLE PICO_INVALID_SAMPLERATIO PICO_INVALID_CALL PICO_DRIVER_FUNCTION

## 3.18 ps6000GetValuesBulk

```
PICO_STATUS ps6000GetValuesBulk
(
    int16_t          handle,
    uint32_t         * noOfSamples,
    uint32_t         fromSegmentIndex,
    uint32_t         toSegmentIndex,
    uint32_t         downSampleRatio,
    PS6000_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.

<b>Applicability</b>	<a href="#">Rapid block mode</a>
<b>Arguments</b>	<p><code>handle</code>, identifies the device</p> <p>* <code>noOfSamples</code>, 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.</p> <p><code>fromSegmentIndex</code>, the first segment from which the waveform should be retrieved</p> <p><code>toSegmentIndex</code>, the last segment from which the waveform should be retrieved</p> <p><code>downSampleRatio</code>,  <code>downSampleRatioMode</code>: see <a href="#">ps6000GetValues</a></p> <p>* <code>overflow</code>, 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 <code>overflow</code> array, with <code>overflow[0]</code> containing the flags for the segment numbered <code>fromSegmentIndex</code> and the last element in the array containing the flags for the segment numbered <code>toSegmentIndex</code>. Each element in the array is a bit field as described under <a href="#">ps6000GetValues</a>.</p>
<b>Returns</b>	PICO_OK PICO_INVALID_HANDLE PICO_INVALID_PARAMETER PICO_SEGMENT_OUT_OF_RANGE PICO_NO_SAMPLES_AVAILABLE PICO_STARTINDEX_INVALID PICO_NOT_RESPONDING PICO_DRIVER_FUNCTION PICO_INVALID_SAMPLERATIO

## 3.19 ps6000GetValuesBulkAsync

```
PICO_STATUS ps6000GetValuesBulkAsync
(
    int16_t          handle,
    uint32_t          startIndex,
    uint32_t          * noOfSamples,
    uint32_t          downSampleRatio,
    PS6000_RATIO_MODE downSampleRatioMode,
    uint32_t          fromSegmentIndex,
    uint32_t          toSegmentIndex,
    int16_t          * overflow
)
```

This function retrieves more than one waveform at a time from the driver in [rapid block mode](#) after data collection has stopped. The waveforms must have been collected sequentially and in the same run. The data is returned using a [callback](#).

<b>Applicability</b>	<a href="#">Rapid block mode</a>
<b>Arguments</b>	handle, startIndex, * noOfSamples, downSampleRatio, downSampleRatioMode: see <a href="#">ps6000GetValues</a>  fromSegmentIndex, toSegmentIndex, * overflow: see <a href="#">ps6000GetValuesBulk</a>
<b>Returns</b>	PICO_OK PICO_INVALID_HANDLE PICO_INVALID_PARAMETER PICO_SEGMENT_OUT_OF_RANGE PICO_NO_SAMPLES_AVAILABLE PICO_STARTINDEX_INVALID PICO_NOT_RESPONDING PICO_DRIVER_FUNCTION

## 3.20 ps6000GetValuesOverlapped

```
PICO_STATUS ps6000GetValuesOverlapped
(
    int16_t          handle,
    uint32_t          startIndex,
    uint32_t          * noOfSamples,
    uint32_t          downSampleRatio,
    PS6000_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 [ps6000RunBlock](#). The advantage of this function is that the driver makes contact with the scope only once, when you call [ps6000RunBlock](#), compared with the two contacts that occur when you use the conventional [ps6000RunBlock](#), [ps6000GetValues](#) calling sequence. This slightly reduces the dead time between successive captures in block mode.

After calling [ps6000RunBlock](#), you can optionally use [ps6000GetValues](#) to request further copies of the data. This might be required if you wish to display the data with different data reduction settings.

For more information, see [Using the GetValuesOverlapped functions](#).

<b>Applicability</b>	<a href="#">Block mode</a>
<b>Arguments</b>	handle, startIndex, * noOfSamples, downSampleRatio, downSampleRatioMode, segmentIndex: see <a href="#">ps6000GetValues</a>  * overflow: see <a href="#">ps6000GetValuesBulk</a>
<b>Returns</b>	PICO_OK PICO_INVALID_HANDLE PICO_INVALID_PARAMETER PICO_DRIVER_FUNCTION

### 3.20.1 Using the GetValuesOverlapped functions

1. Open the oscilloscope using [ps6000OpenUnit](#).
2. Select channel ranges and AC/DC coupling using [ps6000SetChannel](#).
3. Using [ps6000GetTimebase](#), select timebases until the required nanoseconds per sample is located.
4. Use the trigger setup functions [ps6000SetTriggerChannelConditions](#), [ps6000SetTriggerChannelDirections](#) and [ps6000SetTriggerChannelProperties](#) to set up the trigger if required.
5. Use [ps6000SetDataBuffer](#) to tell the driver where your memory buffer is.
6. Set up the transfer of the block of data from the oscilloscope using [ps6000GetValuesOverlapped](#).
7. Start the oscilloscope running using [ps6000RunBlock](#).



8. Wait until the oscilloscope is ready using the [ps6000BlockReady](#) callback (or poll using [ps6000IsReady](#)).
9. Display the data.
10. Repeat steps 7 to 9 if needed.
11. Stop the oscilloscope by calling [ps6000Stop](#).

A similar procedure can be used with [rapid block mode](#) using the [ps6000GetValuesOverlappedBulk](#) function.

## 3.21 ps6000GetValuesOverlappedBulk

[PICO\\_STATUS](#) ps6000GetValuesOverlappedBulk

```
(
    int16_t          handle,
    uint32_t          startIndex,
    uint32_t          * noOfSamples,
    uint32_t          downSampleRatio,
    PS6000_RATIO_MODE downSampleRatioMode,
    uint32_t          fromSegmentIndex,
    uint32_t          toSegmentIndex,
    int16_t          * overflow
)
```

This function allows you to make a deferred data-collection request in rapid block mode. The request will be executed, and the arguments validated, when you call [ps6000RunBlock](#). The advantage of this method is that the driver makes contact with the scope only once, when you call [ps6000RunBlock](#), compared with the two contacts that occur when you use the conventional [ps6000RunBlock](#), [ps6000GetValues](#) calling sequence. This slightly reduces the dead time between successive captures in rapid block mode.

After calling [ps6000RunBlock](#), you can optionally use [ps6000GetValues](#) to request further copies of the data. This might be required if you wish to display the data with different data reduction settings.

For more information, see [Using the GetValuesOverlapped functions](#).

<b>Applicability</b>	<a href="#">Rapid block mode</a>
<b>Arguments</b>	handle, startIndex, * noOfSamples, downSampleRatio, downSampleRatioMode: see <a href="#">ps6000GetValues</a>  fromSegmentIndex, toSegmentIndex, * overflow, see <a href="#">ps6000GetValuesBulk</a>
<b>Returns</b>	PICO_OK PICO_INVALID_HANDLE PICO_INVALID_PARAMETER PICO_DRIVER_FUNCTION

## 3.22 ps6000GetValuesTriggerTimeOffsetBulk

```
PICO_STATUS ps6000GetValuesTriggerTimeOffsetBulk
(
    int16_t          handle,
    uint32_t         * timesUpper,
    uint32_t         * timesLower,
    PS6000_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 [ps6000GetTriggerTimeOffset](#) once for each waveform required. See [ps6000GetTriggerTimeOffset](#) for an explanation of trigger time offsets.

There is another version of this function, [ps6000GetValuesTriggerTimeOffsetBulk64](#), that returns trigger time offsets as 64-bit values instead of pairs of 32-bit values.

<b>Applicability</b>	<a href="#">Rapid block mode</a>
<b>Arguments</b>	<p><code>handle</code>, identifies the device</p> <p>* <code>timesUpper</code>, an array of integers. On exit, the most significant 32 bits of the time offset for each requested segment index. <code>times[0]</code> will hold the <code>fromSegmentIndex</code> time offset and the last <code>times</code> index will hold the <code>toSegmentIndex</code> time offset. The array must be long enough to hold the number of requested times.</p> <p>* <code>timesLower</code>, an array of integers. On exit, the least-significant 32 bits of the time offset for each requested segment index. <code>times[0]</code> will hold the <code>fromSegmentIndex</code> time offset and the last <code>times</code> index will hold the <code>toSegmentIndex</code> time offset. The array size must be long enough to hold the number of requested times.</p> <p>* <code>timeUnits</code>, an array of integers. The array must be long enough to hold the number of requested times. On exit, <code>timeUnits[0]</code> will contain the time unit for <code>fromSegmentIndex</code> and the last element will contain the time unit for <code>toSegmentIndex</code>. PS6000_TIME_UNITS values are listed under <a href="#">ps6000GetTriggerTimeOffset</a>.</p> <p><code>fromSegmentIndex</code>, the first segment for which the time offset is required</p> <p><code>toSegmentIndex</code>, the last segment for which the time offset is required. If <code>toSegmentIndex</code> is less than <code>fromSegmentIndex</code> then the driver will wrap around from the last segment to the first.</p>
<b>Returns</b>	PICO_OK PICO_INVALID_HANDLE PICO_NULL_PARAMETER PICO_DEVICE_SAMPLING PICO_SEGMENT_OUT_OF_RANGE PICO_NO_SAMPLES_AVAILABLE PICO_DRIVER_FUNCTION

## 3.23 ps6000GetValuesTriggerTimeOffsetBulk64

[PICO\\_STATUS](#) ps6000GetValuesTriggerTimeOffsetBulk64

```
(
    int16_t          handle,
    int64_t          * times,
    PS6000_TIME_UNITS * timeUnits,
    uint32_t          fromSegmentIndex,
    uint32_t          toSegmentIndex
)
```

This function retrieves the 64-bit time offsets for waveforms captured in [rapid block mode](#).

A 32-bit version of this function, [ps6000GetValuesTriggerTimeOffsetBulk](#), is available for use with programming languages that do not support 64-bit integers. See that function for an explanation of waveform time offsets.

<b>Applicability</b>	<a href="#">Rapid block mode</a>
<b>Arguments</b>	<p><code>handle</code>, identifies the device</p> <p>* <code>times</code>, an array of integers. On exit, this will hold the time offset for each requested segment index. <code>times[0]</code> will hold the time offset for <code>fromSegmentIndex</code>, and the last <code>times</code> index will hold the time offset for <code>toSegmentIndex</code>. The array must be long enough to hold the number of times requested.</p> <p>* <code>timeUnits</code>, see <a href="#">ps6000GetValuesTriggerTimeOffsetBulk</a>.</p> <p><code>fromSegmentIndex</code>, the first segment for which the time offset is required. The results for this segment will be placed in <code>times[0]</code> and <code>timeUnits[0]</code>.</p> <p><code>toSegmentIndex</code>, the last segment for which the time offset is required. The results for this segment will be placed in the last elements of the <code>times</code> and <code>timeUnits</code> arrays. If <code>toSegmentIndex</code> is less than <code>fromSegmentIndex</code> then the driver will wrap around from the last segment to the first.</p>
<b>Returns</b>	PICO_OK PICO_INVALID_HANDLE PICO_NULL_PARAMETER PICO_DEVICE_SAMPLING PICO_SEGMENT_OUT_OF_RANGE PICO_NO_SAMPLES_AVAILABLE PICO_DRIVER_FUNCTION

## 3.24 ps6000IsReady

```
PICO\_STATUS ps6000IsReady  
(  
    int16_t    handle,  
    int16_t    * ready  
)
```

This function may be used instead of a callback function to receive data from [ps6000RunBlock](#). To use this method, pass a NULL pointer as the `lpReady` argument to [ps6000RunBlock](#). You must then poll the driver to see if it has finished collecting the requested samples.

<b>Applicability</b>	<a href="#">Block mode</a>
<b>Arguments</b>	<code>handle</code> , identifies the device  <code>ready</code> , output: indicates the state of the collection. If zero, the device is still collecting. If non-zero, the device has finished collecting and <a href="#">ps6000GetValues</a> can be used to retrieve the data.
<b>Returns</b>	

## 3.25 ps6000IsTriggerOrPulseWidthQualifierEnabled

```
PICO_STATUS ps6000IsTriggerOrPulseWidthQualifierEnabled
(
    int16_t    handle,
    int16_t    * triggerEnabled,
    int16_t    * pulseWidthQualifierEnabled
)
```

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

<b>Applicability</b>	Call after setting up the trigger, just before calling either <a href="#">ps6000RunBlock</a> or <a href="#">ps6000RunStreaming</a>
<b>Arguments</b>	<p>handle, identifies the device</p> <p>triggerEnabled, on exit, indicates whether the trigger will successfully be set when <a href="#">ps6000RunBlock</a> or <a href="#">ps6000RunStreaming</a> is called. A non-zero value indicates that the trigger is set, zero that the trigger is not set.</p> <p>pulseWidthQualifierEnabled, on exit, indicates whether the pulse width qualifier will successfully be set when <a href="#">ps6000RunBlock</a> or <a href="#">ps6000RunStreaming</a> is called. A non-zero value indicates that the pulse width qualifier is set, zero that the pulse width qualifier is not set.</p>
<b>Returns</b>	PICO_OK PICO_INVALID_HANDLE PICO_NULL_PARAMETER PICO_DRIVER_FUNCTION

## 3.26 ps6000MemorySegments

[PICO\\_STATUS](#) ps6000MemorySegments

```
(
    int16_t      handle
    uint32_t      nSegments,
    uint32_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	<p>handle, identifies the device</p> <p>nSegments, the number of segments required:</p> <table><tr><th>Model</th><th>Min</th><th>Max</th></tr><tr><td>PicoScope 6402</td><td>1</td><td>32 768</td></tr><tr><td>PicoScope 6402A</td><td>1</td><td>125 000</td></tr><tr><td>PicoScope 6402B</td><td>1</td><td>250 000</td></tr><tr><td>PicoScope 6402C</td><td>1</td><td>250 000</td></tr><tr><td>PicoScope 6402D</td><td>1</td><td>500 000</td></tr><tr><td>PicoScope 6403</td><td>1</td><td>1 000 000</td></tr><tr><td>PicoScope 6403A</td><td>1</td><td>250 000</td></tr><tr><td>PicoScope 6403B</td><td>1</td><td>500 000</td></tr><tr><td>PicoScope 6403C</td><td>1</td><td>500 000</td></tr><tr><td>PicoScope 6403D</td><td>1</td><td>1 000 000</td></tr><tr><td>PicoScope 6404</td><td>1</td><td>1 000 000</td></tr><tr><td>PicoScope 6404A</td><td>1</td><td>500 000</td></tr><tr><td>PicoScope 6404B</td><td>1</td><td>1 000 000</td></tr><tr><td>PicoScope 6404C</td><td>1</td><td>1 000 000</td></tr><tr><td>PicoScope 6404D</td><td>1</td><td>2 000 000</td></tr><tr><td>PicoScope 6407</td><td>1</td><td>1 000 000</td></tr></table> <p>* 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 then the number of samples available to each channel is nMaxSamples divided by the number of channels.</p>	Model	Min	Max	PicoScope 6402	1	32 768	PicoScope 6402A	1	125 000	PicoScope 6402B	1	250 000	PicoScope 6402C	1	250 000	PicoScope 6402D	1	500 000	PicoScope 6403	1	1 000 000	PicoScope 6403A	1	250 000	PicoScope 6403B	1	500 000	PicoScope 6403C	1	500 000	PicoScope 6403D	1	1 000 000	PicoScope 6404	1	1 000 000	PicoScope 6404A	1	500 000	PicoScope 6404B	1	1 000 000	PicoScope 6404C	1	1 000 000	PicoScope 6404D	1	2 000 000	PicoScope 6407	1	1 000 000
Model	Min	Max																																																		
PicoScope 6402	1	32 768																																																		
PicoScope 6402A	1	125 000																																																		
PicoScope 6402B	1	250 000																																																		
PicoScope 6402C	1	250 000																																																		
PicoScope 6402D	1	500 000																																																		
PicoScope 6403	1	1 000 000																																																		
PicoScope 6403A	1	250 000																																																		
PicoScope 6403B	1	500 000																																																		
PicoScope 6403C	1	500 000																																																		
PicoScope 6403D	1	1 000 000																																																		
PicoScope 6404	1	1 000 000																																																		
PicoScope 6404A	1	500 000																																																		
PicoScope 6404B	1	1 000 000																																																		
PicoScope 6404C	1	1 000 000																																																		
PicoScope 6404D	1	2 000 000																																																		
PicoScope 6407	1	1 000 000																																																		
Returns	<p>PICO_OK</p> <p>PICO_USER_CALLBACK</p> <p>PICO_INVALID_HANDLE</p> <p>PICO_TOO_MANY_SEGMENTS</p> <p>PICO_MEMORY</p> <p>PICO_DRIVER_FUNCTION</p>																																																			

## 3.27 ps6000NoOfStreamingValues

```
PICO\_STATUS ps6000NoOfStreamingValues  
(  
    int16_t    handle,  
    uint32_t   * noOfValues  
)
```

This function returns the number of samples available after data collection in [streaming mode](#). Call it after calling [ps6000Stop](#).

<b>Applicability</b>	<a href="#">Streaming mode</a>
<b>Arguments</b>	<code>handle</code> , identifies the device  <code>* noOfValues</code> , on exit, the number of samples
<b>Returns</b>	PICO_OK PICO_INVALID_HANDLE PICO_NULL_PARAMETER PICO_NO_SAMPLES_AVAILABLE PICO_NOT_USED PICO_BUSY PICO_DRIVER_FUNCTION



## 3.28 ps6000OpenUnit

```
PICO_STATUS ps6000OpenUnit
(
    int16_t * handle,
    int8_t * serial
)
```

This function opens a PicoScope 6000 Series scope attached to the computer. The maximum number of units that can be opened depends on the operating system, the kernel driver and the computer.

<b>Applicability</b>	All modes
<b>Arguments</b>	<p>* <code>handle</code>, on exit, the result of the attempt to open a scope:</p> <ul style="list-style-type: none"> <li>-1 : if the scope fails to open</li> <li>0 : if no scope is found</li> <li>&gt; 0 : a number that uniquely identifies the scope</li> </ul> <p>If a valid handle is returned, it must be used in all subsequent calls to API functions to identify this scope.</p> <p><code>serial</code>, on entry, a null-terminated string containing the serial number of the scope to be opened. If <code>serial</code> is NULL then the function opens the first scope found; otherwise, it tries to open the scope that matches the string.</p>
<b>Returns</b>	<p>PICO_OK  PICO_OS_NOT_SUPPORTED  PICO_OPEN_OPERATION_IN_PROGRESS  PICO_EEPROM_CORRUPT  PICO_KERNEL_DRIVER_TOO_OLD  PICO_FW_FAIL  PICO_MAX_UNITS_OPENED  PICO_NOT_FOUND (if the specified unit was not found)  PICO_NOT_RESPONDING  PICO_MEMORY_FAIL  PICO_ANALOG_BOARD  PICO_CONFIG_FAIL_AWG  PICO_INITIALISE_FPGA</p>

## 3.29 ps6000OpenUnitAsync

```
PICO\_STATUS ps6000OpenUnitAsync  
(  
    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 [ps6000OpenUnitProgress](#) until that function returns a non-zero value.

<b>Applicability</b>	All modes
<b>Arguments</b>	<ul style="list-style-type: none"><li>* <code>status</code>, a status code:<ul style="list-style-type: none"><li>0 if the open operation was disallowed because another open operation is in progress</li><li>1 if the open operation was successfully started</li></ul></li><li>* <code>serial</code>: see <a href="#">ps6000OpenUnit</a></li></ul>
<b>Returns</b>	<code>PICO_OK</code> <code>PICO_OPEN_OPERATION_IN_PROGRESS</code> <code>PICO_OPERATION_FAILED</code>

## 3.30 ps6000OpenUnitProgress

```
PICO\_STATUS ps6000OpenUnitProgress  
(  
    int16_t * handle,  
    int16_t * progressPercent,  
    int16_t * complete  
)
```

This function checks on the progress of a request made to [ps6000OpenUnitAsync](#) to open a scope.

<b>Applicability</b>	Use after <a href="#">ps6000OpenUnitAsync</a>
<b>Arguments</b>	<ul style="list-style-type: none"><li>* <code>handle</code>: see <a href="#">ps6000OpenUnit</a>. This handle is valid only if the function returns <code>PICO_OK</code>.</li><li>* <code>progressPercent</code>, on exit, the percentage progress towards opening the scope. 100% implies that the open operation is complete.</li><li>* <code>complete</code>, set to 1 when the open operation has finished</li></ul>
<b>Returns</b>	<code>PICO_OK</code> <code>PICO_NULL_PARAMETER</code> <code>PICO_OPERATION_FAILED</code>

## 3.31 ps6000PingUnit

```
PICO\_STATUS ps6000PingUnit  
(  
    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.

<b>Applicability</b>	All modes
<b>Arguments</b>	<code>handle</code> , the handle of the required device
<b>Returns</b>	<code>PICO_OK</code> <code>PICO_INVALID_HANDLE</code> <code>PICO_DRIVER_FUNCTION</code> <code>PICO_POWER_SUPPLY_CONNECTED</code> <code>PICO_POWER_SUPPLY_NOT_CONNECTED</code> <code>PICO_BUSY</code> <code>PICO_NOT_RESPONDING</code>

## 3.32 ps6000RunBlock

```
PICO_STATUS ps6000RunBlock
(
    int16_t          handle,
    uint32_t         noOfPreTriggerSamples,
    uint32_t         noOfPostTriggerSamples,
    uint32_t         timebase,
    int16_t          oversample,
    int32_t          * timeIndisposedMs,
    uint32_t         segmentIndex,
    ps6000BlockReady 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`.

Note that [ETS mode](#) only supports timebases 0, 1 and 2.

<b>Applicability</b>	<a href="#">Block mode</a> , <a href="#">rapid block mode</a>
<b>Arguments</b>	<p><code>handle</code>, identifies the device</p> <p><code>noOfPreTriggerSamples</code>, the number of samples to return before the trigger event. If no trigger has been set, then this argument is added to <code>noOfPostTriggerSamples</code> to give the maximum number of data points (samples) to collect.</p> <p><code>noOfPostTriggerSamples</code>, the number of samples to return after the trigger event. If no trigger event has been set, then this argument is added to <code>noOfPreTriggerSamples</code> 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:</p> $\text{noOfPreTriggerSamples} + \text{noOfPostTriggerSamples}$ <p><code>timebase</code>, a number in the range 0 to <math>2^{32}-1</math>. See the <a href="#">guide to calculating timebase values</a>.</p> <p><code>oversample</code>, the <a href="#">oversampling</a> factor, a number in the range 1 to 256.</p> <p>* <code>timeIndisposedMs</code>, 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.</p> <p><code>segmentIndex</code>, zero-based, specifies which <a href="#">memory segment</a> to use.</p>

	<p><code>lpReady</code>, a pointer to the <a href="#">ps6000BlockReady</a> callback function that the driver will call when the data has been collected. To use the <a href="#">ps6000IsReady</a> polling method instead of a callback function, set this pointer to <code>NULL</code>.</p> <p>* <code>pParameter</code>, a void pointer that is passed to the <a href="#">ps6000BlockReady</a> callback function. The callback can use this pointer to return arbitrary data to the application.</p>
<b>Returns</b>	<p> <code>PICO_OK</code>  <code>PICO_INVALID_HANDLE</code>  <code>PICO_USER_CALLBACK</code>  <code>PICO_SEGMENT_OUT_OF_RANGE</code>  <code>PICO_INVALID_CHANNEL</code>  <code>PICO_INVALID_TRIGGER_CHANNEL</code>  <code>PICO_INVALID_CONDITION_CHANNEL</code>  <code>PICO_TOO_MANY_SAMPLES</code>  <code>PICO_INVALID_TIMEBASE</code>  <code>PICO_NOT_RESPONDING</code>  <code>PICO_CONFIG_FAIL</code>  <code>PICO_INVALID_PARAMETER</code>  <code>PICO_NOT_RESPONDING</code>  <code>PICO_TRIGGER_ERROR</code>  <code>PICO_DRIVER_FUNCTION</code>  <code>PICO_EXTERNAL_FREQUENCY_INVALID</code>  <code>PICO_FW_FAIL</code>  <code>PICO_NOT_ENOUGH_SEGMENTS</code> (in Bulk mode)  <code>PICO_TRIGGER_AND_EXTERNAL_CLOCK_CLASH</code>  <code>PICO_PWQ_AND_EXTERNAL_CLOCK_CLASH</code>  <code>PICO_PULSE_WIDTH_QUALIFIER</code>  <code>PICO_SEGMENT_OUT_OF_RANGE</code> (in Overlapped mode)  <code>PICO_STARTINDEX_INVALID</code> (in Overlapped mode)  <code>PICO_INVALID_SAMPLERATIO</code> (in Overlapped mode)  <code>PICO_CONFIG_FAIL</code>  <code>PICO_SIGGEN_GATING_AUXIO_ENABLED</code> (signal generator is set to trigger on AUX input with incompatible trigger type) </p>

### 3.33 ps6000RunStreaming

```
PICO_STATUS ps6000RunStreaming
(
    int16_t          handle,
    uint32_t          * sampleInterval,
    PS6000_TIME_UNITS sampleIntervalTimeUnits
    uint32_t          maxPreTriggerSamples,
    uint32_t          maxPostTriggerSamples,
    int16_t          autoStop,
    uint32_t          downSampleRatio,
    PS6000_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 [ps6000GetStreamingLatestValues](#) to retrieve the data. See [Using streaming mode](#) for a step-by-step guide to this process.

When a trigger is set, the total number of samples stored in the driver is the sum of `maxPreTriggerSamples` and `maxPostTriggerSamples`. If `autoStop` is false then this will become the maximum number of samples without downsampling.

<b>Applicability</b>	<a href="#">Streaming mode</a>
<b>Arguments</b>	<p><code>handle</code>, identifies the device</p> <p><code>* sampleInterval</code>, on entry, the requested time interval between samples; on exit, the actual time interval used</p> <p><code>sampleIntervalTimeUnits</code>, the unit of time used for <code>sampleInterval</code>. Use one of these values:  <a href="#">PS6000_FS</a>  <a href="#">PS6000_PS</a>  <a href="#">PS6000_NS</a>  <a href="#">PS6000_US</a>  <a href="#">PS6000_MS</a>  <a href="#">PS6000_S</a></p> <p><code>maxPreTriggerSamples</code>, the maximum number of raw samples before a trigger event for each enabled channel. If no trigger condition is set this argument is ignored.</p> <p><code>maxPostTriggerSamples</code>, the maximum number of raw samples after a trigger event for each enabled channel. If no trigger condition is set, this argument states the maximum number of samples to be stored.</p> <p><code>autoStop</code>, a flag that specifies if the streaming should stop when all of <code>maxSamples</code> have been captured.</p> <p><code>downSampleRatio</code>,  <code>downSampleRatioMode</code>: see <a href="#">ps6000GetValues</a></p>

	<code>overviewBufferSize</code> , 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 <code>bufferLth</code> value passed to <a href="#">ps6000SetDataBuffer</a> .
<b>Returns</b>	PICO_OK PICO_INVALID_HANDLE PICO_USER_CALLBACK PICO_NULL_PARAMETER PICO_INVALID_PARAMETER PICO_STREAMING_FAILED PICO_NOT_RESPONDING PICO_TRIGGER_ERROR PICO_INVALID_SAMPLE_INTERVAL PICO_INVALID_BUFFER PICO_DRIVER_FUNCTION PICO_EXTERNAL_FREQUENCY_INVALID PICO_FW_FAIL PICO_TRIGGER_AND_EXTERNAL_CLOCK_CLASH PICO_PWQ_AND_EXTERNAL_CLOCK_CLASH PICO_MEMORY PICO_SIGGEN_GATING_AUXIO_ENABLED (signal generator is set to trigger on AUX input with incompatible trigger type)



## 3.34 ps6000SetChannel

```
PICO_STATUS ps6000SetChannel
(
    int16_t          handle,
    PS6000_CHANNEL  channel,
    int16_t          enabled,
    PS6000_COUPLING type,
    PS6000_RANGE     range,
    float            analogueOffset,
    PS6000_BANDWIDTH_LIMITER bandwidth
)
```

This function specifies whether an input channel is to be enabled, its input coupling type, voltage range, analog offset and bandwidth limit. Some of the arguments within this function have model-specific values. Please consult the relevant section below according to the model you have.

<b>Applicability</b>	All modes
<b>Arguments</b>	
<p><code>handle</code>, identifies the device</p> <p><code>channel</code>, the channel to be configured. The values are:</p> <p><a href="#"><u>PS6000_CHANNEL_A</u></a>: Channel input  <a href="#"><u>PS6000_CHANNEL_B</u></a>: Channel input  <a href="#"><u>PS6000_CHANNEL_C</u></a>: Channel input  <a href="#"><u>PS6000_CHANNEL_D</u></a>: Channel input</p> <p><code>enabled</code>, whether or not to enable the channel. The values are:</p> <p>TRUE: enable  FALSE: do not enable</p> <p><code>type</code>, the impedance and coupling type. The values supported are:</p> <p><b>PicoScope 6402/6403/6404 (all model variants)</b></p> <p><code>PS6000_AC</code>, 1 M<math>\Omega</math> impedance, AC coupling. The channel accepts input frequencies from about 1 hertz up to its maximum -3 dB analog bandwidth.</p> <p><code>PS6000_DC_1M</code>, 1 M<math>\Omega</math> impedance, DC coupling. The scope accepts all input frequencies from zero (DC) up to its maximum -3 dB analog bandwidth.</p> <p><code>PS6000_DC_50R</code>, DC coupling, 50 <math>\Omega</math> impedance. In this mode the <math>\pm 10</math> volt and <math>\pm 20</math> volt input ranges are not available.</p> <p><b>PicoScope 6407</b></p> <p><code>PS6000_DC_50R</code>, DC coupling, 50 <math>\Omega</math> impedance.</p>	

`range`, the input voltage range:

**PicoScope 6402/6403/6404 (all model variants)**

[`PS6000\_50MV`](#):  $\pm 50$  mV  
[`PS6000\_100MV`](#):  $\pm 100$  mV  
[`PS6000\_200MV`](#):  $\pm 200$  mV  
[`PS6000\_500MV`](#):  $\pm 500$  mV  
[`PS6000\_1V`](#):  $\pm 1$  V  
[`PS6000\_2V`](#):  $\pm 2$  V  
[`PS6000\_5V`](#):  $\pm 5$  V  
[`PS6000\_10V`](#):  $\pm 10$  V \*  
[`PS6000\_20V`](#):  $\pm 20$  V \*

\* not available when `type = PS6000_DC_50R`

**PicoScope 6407**

[`PS6000\_100MV`](#):  $\pm 100$  mV

`analogueOffset`, a voltage to add to the input channel before digitization.

**PicoScope 6402/6403 (all model variants)**

The allowable range of offsets depends on the input range selected for the channel, as follows:

50 mV to 200 mV: [`MIN\_ANALOGUE\_OFFSET\_50MV\_200MV`](#) to [`MAX\_ANALOGUE\_OFFSET\_50MV\_200MV`](#)

500 mV to 2 V: [`MIN\_ANALOGUE\_OFFSET\_500MV\_2V`](#) to [`MAX\_ANALOGUE\_OFFSET\_500MV\_2V`](#)

5 V to 20 V: [`MIN\_ANALOGUE\_OFFSET\_5V\_20V`](#) to [`MAX\_ANALOGUE\_OFFSET\_5V\_20V`](#).  
 (When `type = PS6000_DC_50R`, the allowable range is reduced to that of the 50 mV to 200 mV input range, i.e. [`MIN\_ANALOGUE\_OFFSET\_50MV\_200MV`](#) to [`MAX\_ANALOGUE\_OFFSET\_50MV\_200MV`](#)).

Allowable range of offsets can also be returned by [`ps6000GetAnalogueOffset`](#) for the device currently connected.

**PicoScope 6404 (all model variants)**

Allowable range of offsets is returned by [`ps6000GetAnalogueOffset`](#) for the device currently connected.

**PicoScope 6407**

`analogueOffset`, Not used. Set to 0.

`bandwidth`, the bandwidth limiter setting:

**PicoScope 6402/6403 (all model variants)**

[`PS6000\_BW\_FULLL`](#): the connected scope's full specified bandwidth  
[`PS6000\_BW\_20MHZ`](#): -3 dB bandwidth limited to 20 MHz

**PicoScope 6404 (all model variants)**

[`PS6000\_BW\_FULLL`](#): the scope's full specified bandwidth  
[`PS6000\_BW\_25MHZ`](#): -3 dB bandwidth limited to 25 MHz

**PicoScope 6407**

[`PS6000\_BW\_FULLL`](#): the scope's full specified bandwidth

<b>Returns</b>	<code>PICO_OK</code> <code>PICO_USER_CALLBACK</code> <code>PICO_INVALID_HANDLE</code> <code>PICO_INVALID_CHANNEL</code> <code>PICO_INVALID_VOLTAGE_RANGE</code> <code>PICO_INVALID_COUPLING</code> <code>PICO_COUPLING_NOT_SUPPORTED</code> <code>PICO_INVALID_ANALOGUE_OFFSET</code> <code>PICO_INVALID_BANDWIDTH</code> <code>PICO_BANDWIDTH_NOT_SUPPORTED</code> <code>PICO_DRIVER_FUNCTION</code>
----------------	---

## 3.35 ps6000SetDataBuffer

```
PICO_STATUS ps6000SetDataBuffer
(
    int16_t          handle,
    PS6000_CHANNEL   channel,
    int16_t          * buffer,
    uint32_t          bufferLth,
    PS6000_RATIO_MODE downSampleRatioMode
)
```

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 must call [ps6000SetDataBuffers](#) instead.

The buffer remains persistent between captures until it is replaced with another buffer or the buffer is set to NULL. The buffer can be replaced at any time between calls to [ps6000GetValues](#).

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

<b>Applicability</b>	<a href="#">Block</a> , <a href="#">rapid block</a> and <a href="#">streaming</a> modes. All <a href="#">downsampling</a> modes except <a href="#">aggregation</a> .
<b>Arguments</b>	<p><code>handle</code>, identifies the device</p> <p><code>channel</code>, the channel you want to use with the buffer. Use one of these values:</p> <p><a href="#">PS6000_CHANNEL_A</a>  <a href="#">PS6000_CHANNEL_B</a>  <a href="#">PS6000_CHANNEL_C</a>  <a href="#">PS6000_CHANNEL_D</a></p> <p><code>buffer</code>, the location of the buffer</p> <p><code>bufferLth</code>, the size of the <code>buffer</code> array</p> <p><code>downSampleRatioMode</code>, the <a href="#">downsampling</a> mode. See <a href="#">ps6000GetValues</a> for the available modes, but note that a single call to <a href="#">ps6000SetDataBuffer</a> can only associate one buffer with one downsampling mode. If you intend to call <a href="#">ps6000GetValues</a> with more than one downsampling mode activated, then you must call <a href="#">ps6000SetDataBuffer</a> several times to associate a separate buffer with each downsampling mode.</p>
<b>Returns</b>	PICO_OK PICO_INVALID_HANDLE PICO_INVALID_CHANNEL PICO_RATIO_MODE_NOT_SUPPORTED PICO_DRIVER_FUNCTION PICO_INVALID_PARAMETER

## 3.36 ps6000SetDataBufferBulk

```
PICO_STATUS ps6000SetDataBufferBulk
(
    int16_t          handle,
    PS6000_CHANNEL   channel,
    int16_t          * buffer,
    uint32_t          bufferLth,
    uint32_t          waveform,
    PS6000_RATIO_MODE downSampleRatioMode
)
```

This function allows you to associate a buffer with a specified waveform number and input channel in [rapid block mode](#). The number of waveforms captured is determined by the `nCaptures` argument sent to [ps6000SetNoOfCaptures](#). There is only one buffer for each waveform because the only downsampling mode that requires two buffers, [aggregation](#) mode, is not available in rapid block mode. Call one of the [GetValues](#) functions to retrieve the data after capturing.

<b>Applicability</b>	<a href="#">Rapid block mode</a> without <a href="#">aggregation</a> .
<b>Arguments</b>	<p><code>handle</code>, identifies the device</p> <p><code>channel</code>, the input channel to use with this buffer</p> <p><code>buffer</code>, an array in which the captured data is stored</p> <p><code>bufferLth</code>, the size of the buffer</p> <p><code>waveform</code>, an index to the waveform number. Range: 0 to <code>nCaptures</code> - 1</p> <p><code>downSampleRatioMode</code>: see <a href="#">ps6000GetValues</a></p>
<b>Returns</b>	PICO_OK PICO_INVALID_HANDLE PICO_INVALID_CHANNEL PICO_INVALID_PARAMETER PICO_RATIO_MODE_NOT_SUPPORTED PICO_DRIVER_FUNCTION

## 3.37 ps6000SetDataBuffers

[PICO\\_STATUS](#) ps6000SetDataBuffers

```
(
    int16_t          handle,
    PS6000_CHANNEL   channel,
    int16_t          * bufferMax,
    int16_t          * bufferMin,
    uint32_t          bufferLth,
    PS6000_RATIO_MODE downSampleRatioMode
)
```

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 [ps6000SetDataBuffer](#) instead.

<b>Applicability</b>	<a href="#">Block</a> and <a href="#">streaming</a> modes with <a href="#">aggregation</a> .
<b>Arguments</b>	<p><code>handle</code>, identifies the device</p> <p><code>channel</code>, the channel for which you want to set the buffers. Use one of these constants:</p> <p><a href="#">PS6000_CHANNEL_A</a>  <a href="#">PS6000_CHANNEL_B</a>  <a href="#">PS6000_CHANNEL_C</a>  <a href="#">PS6000_CHANNEL_D</a></p> <p>* <code>bufferMax</code>, a buffer to receive the maximum data values in aggregation mode, or the non-aggregated values otherwise.</p> <p>* <code>bufferMin</code>, a buffer to receive the minimum aggregated data values. Not used in other downsampling modes.</p> <p><code>bufferLth</code>, the size of the <code>bufferMax</code> and <code>bufferMin</code> arrays.</p> <p><code>downSampleRatioMode</code>: see <a href="#">ps6000GetValues</a></p>
<b>Returns</b>	PICO_OK PICO_INVALID_HANDLE PICO_INVALID_CHANNEL PICO_RATIO_MODE_NOT_SUPPORTED PICO_DRIVER_FUNCTION PICO_INVALID_PARAMETER

### 3.38 ps6000SetDataBuffersBulk

```
PICO_STATUS ps6000SetDataBuffersBulk
(
    int16_t          handle,
    PS6000_CHANNEL   channel,
    int16_t          * bufferMax,
    int16_t          * bufferMin,
    uint32_t         bufferLth,
    uint32_t         waveform,
    PS6000_RATIO_MODE downSampleRatioMode
)
```

This function tells the driver where to find the buffers for [aggregated](#) data for each waveform in [rapid block mode](#). The number of waveforms captured is determined by the `nCaptures` argument sent to [ps6000SetNoOfCaptures](#). Call one of the [GetValues](#) functions to retrieve the data after capture. If you do not need two buffers, because you are not using [aggregate](#) mode, then you can optionally use [ps6000SetDataBufferBulk](#) instead.

<b>Applicability</b>	<a href="#">Rapid block mode</a> with <a href="#">aggregation</a>
<b>Arguments</b>	<p><code>handle</code>, identifies the device</p> <p><code>channel</code>, the input channel to use with the buffer</p> <p>* <code>bufferMax</code>, a buffer to receive the maximum data values in aggregation mode, or the non-aggregated values otherwise</p> <p>* <code>bufferMin</code>, a buffer to receive the minimum data values in <a href="#">aggregate</a> mode. Not used in other <a href="#">downsampling</a> modes.</p> <p><code>bufferLth</code>, the size of the buffer</p> <p><code>waveform</code>, an index to the waveform number between 0 and <code>nCaptures-1</code></p> <p><code>downSampleRatioMode</code>: see <a href="#">ps6000GetValues</a></p>
<b>Returns</b>	PICO_OK PICO_INVALID_HANDLE PICO_INVALID_CHANNEL PICO_INVALID_PARAMETER PICO_RATIO_MODE_NOT_SUPPORTED PICO_DRIVER_FUNCTION

## 3.39 ps6000SetEts

[PICO\\_STATUS](#) ps6000SetEts

```
(
    int16_t          handle,
    PS6000_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.

<b>Applicability</b>	<a href="#">Block mode</a>
<b>Arguments</b>	<p><code>handle</code>, identifies the device</p> <p><code>mode</code>, the ETS mode. Use one of these values:</p> <ul style="list-style-type: none"> <li><code>PS6000_ETS_OFF</code> - disables ETS</li> <li><code>PS6000_ETS_FAST</code> - enables ETS and provides <code>etsCycles</code> of data, which may contain data from previously returned cycles</li> <li><code>PS6000_ETS_SLOW</code> - enables ETS and provides fresh data every <code>etsCycles</code>. This mode takes longer to provide each data set, but the data sets are more stable and are guaranteed to contain only new data.</li> </ul> <p><code>etscycles</code>, the number of cycles to store: the computer can then select <code>etsInterleave</code> cycles to give the most uniform spread of samples  Range: between two and five times the value of <code>etsInterleave</code>, and not more than <a href="#">PS6000_MAX_ETS_CYCLES</a></p> <p><code>etsInterleave</code>, the number of waveforms to combine into a single ETS capture  Maximum value: <a href="#">PS6000_MAX_INTERLEAVE</a></p> <p>* <code>sampleTimePicoseconds</code>, on exit, the minimum possible effective sampling interval of the ETS data. The actual sampling interval depends on the <code>timebase</code> argument passed to <a href="#">ps6000RunBlock</a>. For example, if the captured sample time is 200 ps and <code>etsInterleave</code> is 4, then the effective sample time in ETS mode is 50 ps.</p>
<b>Returns</b>	<p>PICO_OK</p> <p>PICO_USER_CALLBACK</p> <p>PICO_INVALID_HANDLE</p> <p>PICO_INVALID_PARAMETER</p> <p>PICO_DRIVER_FUNCTION</p>



## 3.40 ps6000SetEtsTimeBuffer

```
PICO\_STATUS ps6000SetEtsTimeBuffer  
(  
    int16_t      handle,  
    int64_t      * buffer,  
    uint32_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.

<b>Applicability</b>	<a href="#">ETS mode</a> only. If your programming language does not support 64-bit data, use the 32-bit version <a href="#">ps6000SetEtsTimeBuffers</a> instead.
<b>Arguments</b>	<code>handle</code> , identifies the device  * <code>buffer</code> , an array of 64-bit words, each representing the time in femtoseconds ( $10^{-15}$ seconds) at which the sample was captured  <code>bufferLth</code> , the size of the buffer array
<b>Returns</b>	PICO_OK PICO_INVALID_HANDLE PICO_NULL_PARAMETER PICO_DRIVER_FUNCTION

## 3.41 ps6000SetEtsTimeBuffers

[PICO\\_STATUS](#) ps6000SetEtsTimeBuffers

```
(
    int16_t      handle,
    uint32_t *   timeUpper,
    uint32_t *   timeLower,
    uint32_t     bufferLth
)
```

This function is a 32-bit equivalent of [ps6000SetEtsTimeBuffer](#) for programming environments that do not support 64-bit data. It defines two buffers containing the upper and lower 32-bit parts of the timing information.

<b>Applicability</b>	<a href="#">ETS mode</a> only
<b>Arguments</b>	<p><code>handle</code>, identifies the device</p> <p>* <code>timeUpper</code>, an array of 32-bit words, each representing the upper 32 bits of the time in femtoseconds (<math>10^{-15}</math> seconds) at which the sample was captured</p> <p>* <code>timeLower</code>, an array of 32-bit words, each representing the lower 32 bits of the time in femtoseconds (<math>10^{-15}</math> seconds) at which the sample was captured</p> <p><code>bufferLth</code>, the size of the <code>timeUpper</code> and <code>timeLower</code> arrays</p>
<b>Returns</b>	<p>PICO_OK</p> <p>PICO_INVALID_HANDLE</p> <p>PICO_NULL_PARAMETER</p> <p>PICO_DRIVER_FUNCTION</p>

## 3.42 ps6000SetExternalClock

```
PICO_STATUS ps6000SetExternalClock
(
    int16_t          handle,
    PS6000_EXTERNAL_FREQUENCY frequency,
    int16_t          threshold
)
```

This function tells the scope whether or not to use an external clock signal fed into the AUX input. The external clock can be used to synchronize one or more PicoScope 6000 units to an external source.

When the external clock input is enabled, the oscilloscope relies on the clock signal for all of its timing. The driver checks that the clock is running before starting a capture, but if the clock signal stops after the initial check, the oscilloscope will not respond to any further commands until it is powered off and on again.

Note: if the AUX input is set as an external clock input, it cannot also be used as an external trigger input.

<b>Applicability</b>	All modes						
<b>Arguments</b>	<p><code>handle</code>, identifies the device</p> <p><code>frequency</code>, the external clock frequency. The possible values are:</p> <p><a href="#"><code>PS6000_FREQUENCY_OFF</code></a>: the scope generates its own clock</p> <p><a href="#"><code>PS6000_FREQUENCY_5MHZ</code></a>: 5 MHz external clock</p> <p><a href="#"><code>PS6000_FREQUENCY_10MHZ</code></a>: 10 MHz external clock</p> <p><a href="#"><code>PS6000_FREQUENCY_20MHZ</code></a>: 20 MHz external clock</p> <p><a href="#"><code>PS6000_FREQUENCY_25MHZ</code></a>: 25 MHz external clock</p> <p>The external clock signal must be within <math>\pm 5\%</math> of the selected frequency, otherwise this function will report an error.</p> <p><code>threshold</code>, the logic threshold voltage:</p> <table border="1"> <tr> <td>-32,512</td><td>-1 volt</td></tr> <tr> <td>0</td><td>0 volts</td></tr> <tr> <td>32,512</td><td>+1 volt</td></tr> </table>	-32,512	-1 volt	0	0 volts	32,512	+1 volt
-32,512	-1 volt						
0	0 volts						
32,512	+1 volt						
<b>Returns</b>	<p>PICO_OK</p> <p>PICO_USER_CALLBACK</p> <p>PICO_INVALID_HANDLE</p> <p>PICO_INVALID_PARAMETER</p> <p>PICO_DRIVER_FUNCTION</p> <p>PICO_EXTERNAL_FREQUENCY_INVALID</p> <p>PICO_FW_FAIL</p> <p>PICO_NOT_RESPONDING</p> <p>PICO_CLOCK_CHANGE_ERROR</p> <p>PICO_WARNING_SIGGEN_AUXIO_TRIGGER_DISABLED (signal generator was</p>						

## 3.43 ps6000SetNoOfCaptures

```
PICO\_STATUS ps6000SetNoOfCaptures  
(  
    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.

<b>Applicability</b>	<a href="#">Rapid block mode</a>
<b>Arguments</b>	<code>handle</code> , identifies the device  <code>nCaptures</code> , the number of waveforms to capture in one run
<b>Returns</b>	PICO_OK PICO_INVALID_HANDLE PICO_INVALID_PARAMETER PICO_DRIVER_FUNCTION

## 3.44 ps6000SetPulseWidthQualifier

```
PICO_STATUS ps6000SetPulseWidthQualifier
(
    int16_t                handle,
    PS6000_PWQ_CONDITIONS * conditions,
    int16_t                nConditions,
    PS6000_THRESHOLD_DIRECTION direction,
    uint32_t               lower,
    uint32_t               upper,
    PS6000_PULSE_WIDTH_TYPE type
)
```

This function sets up the conditions for pulse width qualification, which is used with either threshold triggering, level triggering or window triggering to produce time-qualified triggers. Each call to this function creates a pulse width qualifier equal to the logical AND of the elements of the conditions array. Calling this function multiple times creates the logical OR of multiple AND operations. This AND-OR logic allows you to create any possible Boolean function of the scope's inputs.

<b>Applicability</b>	All modes
<b>Arguments</b>	<p><code>handle</code>, identifies the device</p> <p><code>* conditions</code>, an array of <a href="#">PS6000_PWQ_CONDITIONS</a> 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 <code>conditions</code> is <code>NULL</code>, the pulse-width qualifier is not used.</p> <p><code>nConditions</code>, the number of elements in the <code>conditions</code> array. If <code>nConditions</code> is zero then the pulse-width qualifier is not used. Range: 0 to <a href="#">PS6000_MAX_PULSE_WIDTH_QUALIFIER_COUNT</a>.</p> <p><code>direction</code>, the direction of the signal required for the trigger to fire. See <a href="#">ps6000SetTriggerChannelDirections</a> for the list of possible values. Each channel of the oscilloscope (except the AUX input) has two thresholds for each direction—for example, <a href="#">PS6000_RISING</a> and <a href="#">PS6000_RISING_LOWER</a>—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 <a href="#">PS6000_RISING</a> as the <code>direction</code> argument for both <a href="#">ps6000SetTriggerConditions</a> and <a href="#">ps6000SetPulseWidthQualifier</a> at the same time. There is no such restriction when using window triggers.</p> <p><code>lower</code>, the lower limit of the pulse-width counter, in samples.</p> <p><code>upper</code>, the upper limit of the pulse-width counter, in samples. This parameter is used only when the type is set to <a href="#">PS6000_PW_TYPE_IN_RANGE</a> or <a href="#">PS6000_PW_TYPE_OUT_OF_RANGE</a>.</p>

	<p>type, the pulse-width type, one of these constants:</p> <p><a href="#">PS6000_PW_TYPE_NONE</a>: do not use the pulse width qualifier</p> <p><a href="#">PS6000_PW_TYPE_LESS_THAN</a>: pulse width less than lower</p> <p><a href="#">PS6000_PW_TYPE_GREATER_THAN</a>: pulse width greater than lower</p> <p><a href="#">PS6000_PW_TYPE_IN_RANGE</a>: pulse width between lower and upper</p> <p><a href="#">PS6000_PW_TYPE_OUT_OF_RANGE</a>: pulse width not between lower and upper</p>
<b>Returns</b>	<p>PICO_OK</p> <p>PICO_INVALID_HANDLE</p> <p>PICO_USER_CALLBACK</p> <p>PICO_CONDITIONS</p> <p>PICO_PULSE_WIDTH_QUALIFIER</p> <p>PICO_DRIVER_FUNCTION</p>

### 3.44.1 PS6000\_PWQ\_CONDITIONS structure

A structure of this type is passed to [ps6000SetPulseWidthQualifier](#) in the `conditions` argument to specify the trigger conditions. It is defined as follows:

```
typedef struct tPwqConditions
{
    PS6000_TRIGGER_STATE    channelA;
    PS6000_TRIGGER_STATE    channelB;
    PS6000_TRIGGER_STATE    channelC;
    PS6000_TRIGGER_STATE    channelD;
    PS6000_TRIGGER_STATE    external;
    PS6000_TRIGGER_STATE    aux;
} PS6000_PWQ_CONDITIONS
```

Each structure is the logical AND of the states of the scope's inputs. The [ps6000SetPulseWidthQualifier](#) 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.

<b>Elements</b>	<p><code>channelA</code>, <code>channelB</code>, <code>channelC</code>, <code>channelD</code>, <code>aux</code>: the type of condition that should be applied to each channel. Use these constants:</p> <p><a href="#">PS6000_CONDITION_DONT_CARE</a>  <a href="#">PS6000_CONDITION_TRUE</a>  <a href="#">PS6000_CONDITION_FALSE</a></p> <p>The channels that are set to <a href="#">PS6000_CONDITION_TRUE</a> or <a href="#">PS6000_CONDITION_FALSE</a> must all meet their conditions simultaneously to produce a trigger. Channels set to <a href="#">PS6000_CONDITION_DONT_CARE</a> are ignored.</p> <p><code>external</code>: not used</p>
-----------------	---

## 3.45 ps6000SetSigGenArbitrary

```
PICO_STATUS ps6000SetSigGenArbitrary
(
    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,
    PS6000_SWEEP_TYPE      sweepType,
    PS6000_EXTRA_OPERATIONS operation,
    PS6000_INDEX_MODE      indexMode,
    uint32_t               shots,
    uint32_t               sweeps,
    PS6000_SIGGEN_TRIG_TYPE triggerType,
    PS6000_SIGGEN_TRIG_SOURCE triggerSource,
    int16_t                extInThreshold
)
```

This function programs the arbitrary waveform generator (AWG).

The AWG 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 output frequency is controlled by the *startDeltaPhase* and *stopDeltaPhase* arguments. Only *startDeltaPhase* is required to generate a fixed frequency, *stopDeltaPhase* being additionally required when generating a frequency sweep. Each *deltaPhase* argument can be calculated by calling [ps6000SigGenFrequencyToPhase](#). For information on how this works, see [Calculating deltaPhase](#).

For more information about using this function, read the article [Triggering a PicoScope signal generator using the PicoScope API functions](#).

<b>Applicability</b>	PicoScope 6402/3/4, 6402B/3B/4B, 6402D/3D/4D
<b>Arguments</b>	
<i>handle</i> , identifies the device	
<i>offsetVoltage</i> , the voltage offset, in microvolts, to be applied to the waveform	
<i>pkToPk</i> , the peak-to-peak voltage, in microvolts, of the waveform signal	
<i>startDeltaPhase</i> , the initial value of <i>deltaPhase</i> added to the phase counter as the generator begins to step through the waveform buffer. This argument defines the output frequency when a fixed frequency is desired, or the initial output frequency when a frequency sweep is desired. Call <a href="#">ps6000SigGenFrequencyToPhase</a> to calculate a suitable value.	



`stopDeltaPhase`, the final value of `deltaPhase` added to the phase counter before the generator restarts or reverses the sweep. This argument defines the final output frequency when a frequency sweep is desired. Call [ps6000SigGenFrequencyToPhase](#) to calculate a suitable value. This argument is ignored if `deltaPhaseIncrement` is zero.

`deltaPhaseIncrement`, the amount added to the delta phase value after every `dwellCount` period. This determines the amount by which the generator increments or decrements the output frequency in each `dwellCount` period. If no frequency sweep is required, `deltaPhaseIncrement` must be zero.

`dwellCount`, the time, in units of [dacPeriod](#), between successive additions of `deltaPhaseIncrement` to the delta phase counter. This determines the rate at which the generator sweeps the output frequency. If `deltaPhaseIncrement` is zero, this argument is ignored.

Minimum value: [PS6000\\_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:

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 [ps6000SigGenArbitraryMinMaxValues](#).

`arbitraryWaveformSize`, the size of the arbitrary waveform buffer, in samples. The minimum and maximum allowable values are returned by [ps6000SigGenArbitraryMinMaxValues](#).

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

[PS6000\\_UP](#)

[PS6000\\_DOWN](#)

[PS6000\\_UPDOWN](#)

[PS6000\\_DOWNUP](#)

operation, see [ps6000SigGenBuiltIn](#)

`indexMode`, specifies how the signal will be formed from the arbitrary waveform data. [Single, dual and quad index modes](#) are possible. Use one of these constants:

[PS6000\\_SINGLE](#)

[PS6000\\_DUAL](#)

[PS6000\\_QUAD](#)

`shots`,  
`sweeps`,  
`triggerType`,  
`triggerSource`,  
`extInThreshold`, see [ps6000SigGenBuiltIn](#)

<b>Returns</b>	PICO_OK PICO_INVALID_HANDLE PICO_SIG_GEN_PARAM PICO_SHOTS_SWEEPS_WARNING PICO_NOT_RESPONDING PICO_WARNING_AUX_OUTPUT_CONFLICT PICO_WARNING_EXT_THRESHOLD_CONFLICT
----------------	---

```

PICO_NO_SIGNAL_GENERATOR
PICO_SIGGEN_OFFSET_VOLTAGE
PICO_SIGGEN_PK_TO_PK
PICO_SIGGEN_OUTPUT_OVER_VOLTAGE
PICO_DRIVER_FUNCTION
PICO_SIGGEN_WAVEFORM_SETUP_FAILED
PICO_AWG_NOT_SUPPORTED (e.g. if device is a 6402/3/4 A/C)

```

### 3.45.1 Calculating deltaPhase

The AWG steps through the waveform by adding a *deltaPhase* value between 1 and *phaseAccumulatorSize*–1 to the phase accumulator every *dacPeriod* ( $= 1/\text{dacFrequency}$ ). If *deltaPhase* is constant, the generator produces a waveform at a constant frequency that can be calculated as follows:

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

where:

*outputFrequency* = repetition rate of the complete arbitrary waveform  
*dacFrequency* = update rate of AWG DAC (see table below)  
*deltaPhase* = calculated from *startDeltaPhase* and *deltaPhaseIncrement*  
*phaseAccumulatorSize* = maximum count of phase accumulator (see table below)  
*awgBufferSize* = maximum AWG buffer size (see table below)  
*arbitraryWaveformSize* = length in samples of the user-defined waveform

Parameter	Original/A/B models	C/D models
<i>dacFrequency</i>	200 MHz	
<i>dacPeriod</i> ( $= 1/\text{dacFrequency}$ )	5 ns	
<i>phaseAccumulatorSize</i>	4 294 967 296 ( $2^{32}$ )	
<i>awgBufferSize</i>	16 384	65 536

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 specified intervals.

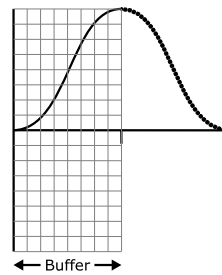
### 3.45.2 Index modes

The [arbitrary waveform generator](#) supports **single**, **dual** and **quad** index modes to help you make the best use of the waveform buffer.

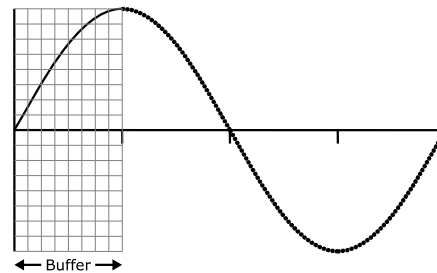
**Single mode.** The generator outputs the raw contents of the buffer repeatedly. This mode is the only one that can generate asymmetrical waveforms. You can also use this mode for symmetrical waveforms, but the dual and quad modes make more efficient use of the buffer memory.



**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.



**Quad mode.** The generator outputs the contents of the buffer, then on its second pass through the buffer outputs the same data in reverse order. On the third and fourth passes it does the same but with a negative version of the data. This allows you to specify only the first quarter of a waveform with fourfold symmetry, such as a sine wave, and let the generator fill in the other three quarters.



## 3.46 ps6000SetSigGenBuiltIn

```
PICO_STATUS ps6000SetSigGenBuiltIn
(
    int16_t          handle,
    int32_t          offsetVoltage,
    uint32_t         pkToPk
    int16_t          waveType
    float            startFrequency,
    float            stopFrequency,
    float            increment,
    float            dwellTime,
    PS6000_SWEEP_TYPE sweepType,
    PS6000_EXTRA_OPERATIONS operation,
    uint32_t         shots,
    uint32_t         sweeps,
    PS6000_SIGGEN_TRIG_TYPE triggerType,
    PS6000_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.

<b>Applicability</b>	All modes
<b>Arguments</b>	
<code>handle</code> , identifies the device	
<code>offsetVoltage</code> , the voltage offset, in microvolts, to be applied to the waveform	
<code>pkToPk</code> , the peak-to-peak voltage, in microvolts, of the waveform signal	
<code>waveType</code> , the type of waveform to be generated:	
PS6000_SINE	sine wave
PS6000_SQUARE	square wave
PS6000_TRIANGLE	triangle wave
PS6000_RAMP_UP	rising sawtooth
PS6000_RAMP_DOWN	falling sawtooth
PS6000_SINC	sin (x)/x
PS6000_GAUSSIAN	Gaussian
PS6000_HALF_SINE	half (full-wave rectified) sine
PS6000_DC_VOLTAGE	DC voltage
PS6000_WHITE_NOISE	white noise
<code>startFrequency</code> , the frequency that the signal generator will initially produce. For allowable values see <a href="#">PS6000_SINE_MAX_FREQUENCY</a> and related values.	
<code>stopFrequency</code> , the frequency at which the sweep reverses direction or returns to the initial frequency	
<code>increment</code> , 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:

[`PS6000\_UP`](#)  
[`PS6000\_DOWN`](#)  
[`PS6000\_UPDOWN`](#)  
[`PS6000\_DOWNUP`](#)

`operation`, selects periodic signal, white noise or PRBS:

<code>PS6000_ES_OFF (0)</code>	produces the waveform specified by <code>waveType</code>
<code>PS6000_WHITENOISE (1)</code>	produces white noise and ignores all settings except <code>offsetVoltage</code> and <code>pkTopk</code>
<code>PS6000_PRBS (2)</code>	produces a <a href="#">pseudo-random binary sequence (PRBS)</a> and ignores all settings except <code>offsetVoltage</code> and <code>pkTopk</code>

`shots`, the number of cycles of the waveform to be produced after a trigger event. If non-zero (from 1 to [`MAX\_SWEEPS\_SHOTS`](#)), `sweeps` must be zero.

`sweeps`, the number of times to sweep the frequency after a trigger event, according to `sweepType`. If non-zero (from 1 to [`MAX\_SWEEPS\_SHOTS`](#)), `shots` must be zero.

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

<code>PS6000_SIGGEN_RISING</code>	trigger on rising edge
<code>PS6000_SIGGEN_FALLING</code>	trigger on falling edge
<code>PS6000_SIGGEN_GATE_HIGH</code>	run while trigger is high (not available if <code>triggerSource</code> is AUX)
<code>PS6000_SIGGEN_GATE_LOW</code>	run while trigger is low (not available if <code>triggerSource</code> is AUX)

`triggerSource`, the source that will trigger the signal generator:

<code>PS6000_SIGGEN_NONE</code>	run without waiting for trigger
<code>PS6000_SIGGEN_SCOPE_TRIG</code>	use scope trigger
<code>PS6000_SIGGEN_AUX_IN</code>	use AUX input
<code>PS6000_SIGGEN_SOFT_TRIG</code>	wait for software trigger provided by <a href="#"><code>ps6000SigGenSoftwareControl</code></a>
<code>PS6000_SIGGEN_TRIGGER_RAW</code>	reserved

If a trigger source other than [`PS6000\_SIGGEN\_NONE`](#) is specified, either `shots` or `sweeps`, but not both, must be non-zero.

`extInThreshold`, the threshold voltage on the AUX input when used as a trigger source. If a different AUX threshold has previously been set up by [`ps6000SetTriggerChannelProperties`](#), [`ps6000SetPulseWidthQualifier`](#) or [`ps6000SetSimpleTrigger`](#), this function will override it and return `PICO_WARNING_AUX_OUTPUT_CONFLICT`.

<b>Returns</b>	<p>PICO_OK PICO_INVALID_HANDLE PICO_SIG_GEN_PARAM PICO_SHOTS_SWEEPS_WARNING PICO_NOT_RESPONDING PICO_WARNING_AUX_OUTPUT_CONFLICT (see <code>extInThreshold</code> above) PICO_WARNING_EXT_THRESHOLD_CONFLICT PICO_NO_SIGNAL_GENERATOR PICO_SIGGEN_OFFSET_VOLTAGE PICO_SIGGEN_PK_TO_PK PICO_SIGGEN_OUTPUT_OVER_VOLTAGE PICO_DRIVER_FUNCTION PICO_SIGGEN_WAVEFORM_SETUP_FAILED PICO_NOT_RESPONDING PICO_SIGGEN_GATING_AUXIO_NOT_AVAILABLE (AUX input cannot be used with requested <code>triggerType</code>) PICO_SIGGEN_TRIGGER_AND_EXTERNAL_CLOCK_CLASH (cannot use AUX as trigger input because it is being used as a <a href="#">clock input</a>)</p>
----------------	---

## 3.47 ps6000SetSigGenBuiltInV2

```
PICO_STATUS ps6000SetSigGenBuiltInV2
(
    int16_t                handle,
    int32_t                offsetVoltage,
    uint32_t               pkToPk
    int16_t                waveType
    double                 startFrequency,
    double                 stopFrequency,
    double                 increment,
    double                 dwellTime,
    PS6000_SWEEP_TYPE      sweepType,
    PS6000_EXTRA_OPERATIONS operation,
    uint32_t               shots,
    uint32_t               sweeps,
    PS6000_SIGGEN_TRIG_TYPE triggerType,
    PS6000_SIGGEN_TRIG_SOURCE triggerSource,
    int16_t                extInThreshold
)
```

This function sets up the signal generator. It differs from [ps6000SetSigGenBuiltIn](#) in having double-precision arguments instead of floats, giving greater resolution when setting the output frequency.

For more information about using this function, read the article [Triggering a PicoScope signal generator using the PicoScope API functions](#).

<b>Applicability</b>	All modes
<b>Arguments</b>	See <a href="#">ps6000SetSigGenBuiltIn</a>
<b>Returns</b>	See <a href="#">ps6000SetSigGenBuiltIn</a>

## 3.48 ps6000SetSimpleTrigger

```
PICO_STATUS ps6000SetSimpleTrigger
(
    int16_t                handle,
    int16_t                enable,
    PS6000_CHANNEL        source,
    int16_t                threshold,
    PS6000_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 canceled. The trigger threshold includes a small, fixed amount of [hysteresis](#).

<b>Applicability</b>	All modes
<b>Arguments</b>	<p><code>handle</code>, identifies the device</p> <p><code>enabled</code>: zero to disable the trigger, any non-zero value to set the trigger.</p> <p><code>source</code>: the channel on which to trigger. This can be one of the four input channels listed under <a href="#">ps6000SetChannel</a>, or <a href="#">PS6000_TRIGGER_AUX</a> for the AUX input.</p> <p><code>threshold</code>: the <a href="#">ADC count</a> at which the trigger will fire.</p> <p><code>direction</code>: 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.</p> <p><code>delay</code>: the time between the trigger occurring and the first sample being taken.</p> <p><code>autoTrigger_ms</code>: the number of milliseconds the device will wait if no trigger occurs.</p>
<b>Returns</b>	PICO_OK PICO_INVALID_HANDLE PICO_USER_CALLBACK PICO_DRIVER_FUNCTION



## 3.49 ps6000SetTriggerChannelConditions

```
PICO_STATUS ps6000SetTriggerChannelConditions
(
    int16_t                handle,
    PS6000_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 [PS6000\\_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 [ps6000SetSimpleTrigger](#).

<b>Applicability</b>	All modes
<b>Arguments</b>	<p><code>handle</code>, identifies the device</p> <p><code>conditions</code>, an array of <a href="#">PS6000_TRIGGER_CONDITIONS</a> 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.</p> <p><code>nConditions</code>, the number of elements in the <code>conditions</code> array. If <code>nConditions</code> is zero then triggering is switched off.</p>
<b>Returns</b>	PICO_OK PICO_INVALID_HANDLE PICO_USER_CALLBACK PICO_CONDITIONS PICO_MEMORY_FAIL PICO_DRIVER_FUNCTION

### 3.49.1 PS6000\_TRIGGER\_CONDITIONS structure

A structure of this type is passed to [ps6000SetTriggerChannelConditions](#) in the `conditions` argument to specify the trigger conditions, and is defined as follows:

```
typedef struct tTriggerConditions
{
    PS6000_TRIGGER_STATE    channelA;
    PS6000_TRIGGER_STATE    channelB;
    PS6000_TRIGGER_STATE    channelC;
    PS6000_TRIGGER_STATE    channelD;
    PS6000_TRIGGER_STATE    external;
    PS6000_TRIGGER_STATE    aux;
    PS6000_TRIGGER_STATE    pulseWidthQualifier;
} PS6000_TRIGGER_CONDITIONS
```

Each structure is the logical AND of the states of the scope's inputs. The [ps6000SetTriggerChannelConditions](#) 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.

<b>Elements</b>	<p><code>channelA</code>, <code>channelB</code>, <code>channelC</code>, <code>channelD</code>, <code>aux</code>, <code>pulseWidthQualifier</code>: the type of condition that should be applied to each channel. Use these constants:</p> <p><a href="#">PS6000_CONDITION_DONT_CARE</a>  <a href="#">PS6000_CONDITION_TRUE</a>  <a href="#">PS6000_CONDITION_FALSE</a></p> <p>The channels that are set to <a href="#">PS6000_CONDITION_TRUE</a> or <a href="#">PS6000_CONDITION_FALSE</a> must all meet their conditions simultaneously to produce a trigger. Channels set to <a href="#">PS6000_CONDITION_DONT_CARE</a> are ignored.</p> <p><code>external</code>: not used</p>
-----------------	---

## 3.50 ps6000SetTriggerChannelDirections

[PICO\\_STATUS](#) ps6000SetTriggerChannelDirections

```
(
    int16_t                handle,
    PS6000_THRESHOLD_DIRECTION channelA,
    PS6000_THRESHOLD_DIRECTION channelB,
    PS6000_THRESHOLD_DIRECTION channelC,
    PS6000_THRESHOLD_DIRECTION channelD,
    PS6000_THRESHOLD_DIRECTION ext,
    PS6000_THRESHOLD_DIRECTION aux
)
```

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

<b>Applicability</b>	All modes
<b>Arguments</b>	<p>handle, identifies the device</p> <p>channelA, channelB, channelC, channelD, aux, 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 <a href="#">direction</a> argument to <a href="#">ps6000SetPulseWidthQualifier</a> for more information.</p> <p>ext: not used</p>
<b>Returns</b>	PICO_OK PICO_INVALID_HANDLE PICO_USER_CALLBACK PICO_INVALID_PARAMETER

### [PS6000\\_THRESHOLD\\_DIRECTION](#) constants

Constant	Trigger type	Threshold	Polarity
PS6000_ABOVE	Gated	Upper	Above
PS6000_ABOVE_LOWER	Gated	Lower	Above
PS6000_BELOW	Gated	Upper	Below
PS6000_BELOW_LOWER	Gated	Lower	Below
PS6000_RISING	Threshold	Upper	Rising
PS6000_RISING_LOWER	Threshold	Lower	Rising
PS6000_FALLING	Threshold	Upper	Falling
PS6000_FALLING_LOWER	Threshold	Lower	Falling
PS6000_RISING_OR_FALLING	Threshold	Lower (for rising edge) Upper (for falling edge)	
PS6000_INSIDE	Window-qualified	Both	Inside
PS6000_OUTSIDE	Window-qualified	Both	Outside
PS6000_ENTER	Window	Both	Entering
PS6000_EXIT	Window	Both	Leaving
PS6000_ENTER_OR_EXIT	Window	Both	Either entering or leaving
PS6000_POSITIVE_RUNT	Window-qualified	Both	Entering from below
PS6000_NEGATIVE_RUNT	Window-qualified	Both	Entering from above
PS6000_NONE	None	None	None

## 3.51 ps6000SetTriggerChannelProperties

```
PICO_STATUS ps6000SetTriggerChannelProperties
(
    int16_t                handle,
    PS6000_TRIGGER_CHANNEL_PROPERTIES * channelProperties
    int16_t                nChannelProperties
    int16_t                auxOutputEnable,
    uint32_t               autoTriggerMilliseconds
)
```

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

<b>Applicability</b>	All modes
<b>Arguments</b>	<p><code>handle</code>, identifies the device</p> <p><code>channelProperties</code>, a pointer to an array of <a href="#">TRIGGER_CHANNEL_PROPERTIES</a> 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 <code>NULL</code> is passed, triggering is switched off.</p> <p><code>nChannelProperties</code>, the size of the <code>channelProperties</code> array. If zero, triggering is switched off.</p> <p><code>auxOutputEnable</code>: not used</p> <p><code>autoTriggerMilliseconds</code>, 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.</p>
<b>Returns</b>	PICO_OK PICO_INVALID_HANDLE PICO_USER_CALLBACK PICO_TRIGGER_ERROR PICO_MEMORY_FAIL PICO_INVALID_TRIGGER_PROPERTY PICO_DRIVER_FUNCTION PICO_INVALID_PARAMETER

### 3.51.1 TRIGGER\_CHANNEL\_PROPERTIES structure

A structure of this type is passed to [ps6000SetTriggerChannelProperties](#) in the `channelProperties` argument to specify the trigger mechanism, and is defined as follows:

```
typedef struct tTriggerChannelProperties
{
    int16_t          thresholdUpper;
    uint16_t         hysteresisUpper;
    int16_t          thresholdLower;
    uint16_t         hysteresisLower;
    PS6000_CHANNEL   channel;
    PS6000_THRESHOLD_MODE thresholdMode;
} PS6000_TRIGGER_CHANNEL_PROPERTIES
```

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

There are two trigger thresholds called Upper and Lower. Each trigger type uses one or other of these thresholds, or both, as specified in [ps6000SetTriggerChannelDirections](#). Each trigger threshold has its own hysteresis setting.

<b>Elements</b>	<p><code>thresholdUpper</code>, the upper threshold at which the trigger fires. It is scaled in 16-bit <a href="#">ADC counts</a> at the currently selected range for that channel. Use when "Upper" or "Both" is specified in <a href="#">ps6000SetTriggerChannelDirections</a>.</p> <p><code>hysteresisUpper</code>, the distance by which the signal must fall below the upper threshold (for rising edge triggers) or rise above the upper threshold (for falling edge triggers) in order to rearm the trigger for the next event. It is scaled in 16-bit counts.</p> <p><code>thresholdLower</code>, lower threshold (see <code>thresholdUpper</code>). Use when "Lower" or "Both" is specified in <a href="#">ps6000SetTriggerChannelDirections</a>.</p> <p><code>hysteresisLower</code>, lower threshold hysteresis (see <code>hysteresisUpper</code>)</p> <p><code>channel</code>, the channel to which the properties apply. This can be one of the four input channels listed under <a href="#">ps6000SetChannel</a>, or <a href="#">PS6000_TRIGGER_AUX</a> for the AUX input.</p> <p><code>thresholdMode</code>, either a level or window trigger. Use one of these constants:              <code>PS6000_LEVEL</code>              <code>PS6000_WINDOW</code></p>
-----------------	--

## 3.52 ps6000SetTriggerDelay

```
PICO\_STATUS ps6000SetTriggerDelay  
(  
    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.

<b>Applicability</b>	<a href="#">Block</a> and <a href="#">rapid block</a> modes
<b>Arguments</b>	<code>handle</code> , identifies the device  <code>delay</code> , the time between the trigger occurring and the first sample. For example, if <code>delay</code> =100 then the scope would wait 100 sample periods before sampling. At a <a href="#">timebase</a> of 5 GS/s, or 200 ps per sample ( <code>timebase</code> = 0), the total delay would then be 100 x 200 ps = 20 ns. Range: 0 to <a href="#">MAX_DELAY_COUNT</a>
<b>Returns</b>	PICO_OK PICO_INVALID_HANDLE PICO_USER_CALLBACK PICO_DRIVER_FUNCTION

### 3.53 ps6000SigGenArbitraryMinMaxValues

```
PICO_STATUS ps6000SigGenArbitraryMinMaxValues
(
    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 [ps6000SetSigGenArbitrary](#) for setting up the arbitrary waveform generator (AWG). These values vary between different models in the PicoScope 6000 Series.

<b>Applicability</b>	All models with <a href="#">AWG</a>
<b>Arguments</b>	<p><code>handle</code>, identifies the device</p> <p><code>minArbitraryWaveformValue</code>, on exit, the lowest sample value allowed in the <code>arbitraryWaveform</code> buffer supplied to <a href="#">ps6000SetSigGenArbitrary</a>.</p> <p><code>maxArbitraryWaveformValue</code>, on exit, the highest sample value allowed in the <code>arbitraryWaveform</code> buffer supplied to <a href="#">ps6000SetSigGenArbitrary</a>.</p> <p><code>minArbitraryWaveformSize</code>, on exit, the minimum value allowed for the <code>arbitraryWaveformSize</code> argument supplied to <a href="#">ps6000SetSigGenArbitrary</a>.</p> <p><code>maxArbitraryWaveformSize</code>, on exit, the maximum value allowed for the <code>arbitraryWaveformSize</code> argument supplied to <a href="#">ps6000SetSigGenArbitrary</a>.</p>
<b>Returns</b>	<p><code>PICO_OK</code></p> <p><code>PICO_NOT_SUPPORTED_BY_THIS_DEVICE</code>, if the device does not have an arbitrary waveform generator.</p> <p><code>PICO_NULL_PARAMETER</code>, if all the parameter pointers are NULL.</p> <p><code>PICO_INVALID_HANDLE</code></p> <p><code>PICO_DRIVER_FUNCTION</code></p>

## 3.54 ps6000SigGenFrequencyToPhase

[PICO\\_STATUS](#) ps6000SigGenFrequencyToPhase

```
(
    int16_t          handle,
    double           frequency,
    PS6000_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 [ps6000SetSigGenArbitrary](#).

<b>Applicability</b>	All models with <a href="#">AWG</a>
<b>Arguments</b>	<p><code>handle</code>, identifies the device</p> <p><code>frequency</code>, the required AWG output frequency</p> <p><code>indexMode</code>, see <a href="#">AWG index modes</a></p> <p><code>bufferLength</code>, the number of samples in the AWG buffer</p> <p><code>phase</code>, on exit, the <code>deltaPhase</code> argument to be sent to the AWG setup function</p>
<b>Returns</b>	<p>PICO_OK</p> <p>PICO_NOT_SUPPORTED_BY_THIS_DEVICE, if the device does not have an AWG.</p> <p>PICO_SIGGEN_FREQUENCY_OUT_OF_RANGE, if the frequency is out of range.</p> <p>PICO_NULL_PARAMETER, if <code>phase</code> is a NULL pointer.</p> <p>PICO_SIG_GEN_PARAM, if <code>indexMode</code> or <code>bufferLength</code> is out of range.</p> <p>PICO_INVALID_HANDLE</p> <p>PICO_DRIVER_FUNCTION</p>



## 3.55 ps6000SigGenSoftwareControl

```
PICO\_STATUS ps6000SigGenSoftwareControl  
(  
    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](#).

<b>Applicability</b>	Use with <a href="#">ps6000SetSigGenBuiltIn</a> or <a href="#">ps6000SetSigGenArbitrary</a> .
<b>Arguments</b>	<p><code>handle</code>, identifies the device</p> <p><code>state</code>, sets the trigger gate high or low when the trigger type is set to either <code>SIGGEN_GATE_HIGH</code> or <code>SIGGEN_GATE_LOW</code>. Ignored for other trigger types.</p>
<b>Returns</b>	<p><code>PICO_OK</code> <code>PICO_INVALID_HANDLE</code> <code>PICO_NO_SIGNAL_GENERATOR</code> <code>PICO_SIGGEN_TRIGGER_SOURCE</code> <code>PICO_DRIVER_FUNCTION</code> <code>PICO_NOT_RESPONDING</code></p>

## 3.56 ps6000Stop

```
PICO\_STATUS ps6000Stop  
(  
    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.

When running the device in [streaming mode](#), you should always call this function at the after the end of a capture to ensure that the scope is ready for the next capture.

When running the device in [block mode](#), [ETS mode](#) or [rapid block mode](#), you can call this function to interrupt data capture.

If this function is called before a trigger event occurs, the oscilloscope may not contain valid data.

<b>Applicability</b>	All modes
<b>Arguments</b>	<code>handle</code> , identifies the device
<b>Returns</b>	PICO_OK PICO_INVALID_HANDLE PICO_USER_CALLBACK PICO_DRIVER_FUNCTION

## 3.57 ps6000StreamingReady

```
typedef void (CALLBACK *ps6000StreamingReady)
(
    int16_t      handle,
    uint32_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 [ps6000GetStreamingLatestValues](#), and the driver calls it back when streaming-mode data is ready. You can then download the data using the [ps6000GetValuesAsync](#) function.

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

<b>Applicability</b>	<a href="#">Streaming mode</a> only
<b>Arguments</b>	<p><code>handle</code>, identifies the device</p> <p><code>noOfSamples</code>, the number of samples to collect</p> <p><code>startIndex</code>, an index to the first valid sample in the buffer. This is the buffer that was previously passed to <a href="#">ps6000SetDataBuffer</a>.</p> <p><code>overflow</code>, 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.</p> <p><code>triggerAt</code>, an index to the buffer indicating the location of the trigger point relative to <code>startIndex</code>. This parameter is valid only when <code>triggered</code> is non-zero.</p> <p><code>triggered</code>, a flag indicating whether a trigger occurred. If non-zero, a trigger occurred at the location indicated by <code>triggerAt</code>.</p> <p><code>autoStop</code>, the flag that was set in the call to <a href="#">ps6000RunStreaming</a>.</p> <p><code>pParameter</code>, a void pointer passed from <a href="#">ps6000GetStreamingLatestValues</a>. The callback function can write to this location to send any data, such as a status flag, back to the application.</p>
<b>Returns</b>	nothing

## 3.58 Wrapper functions

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

The source code contained in the wrapper project contains a description of the functions and the input and output parameters.

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

The `ps6000Wrap.dll` wrapper DLL has a callback function for streaming data collection that copies data from the driver buffer specified to a temporary application buffer of the same size. To do this, the driver and application buffers must be registered with the wrapper and the corresponding channel(s) must be specified as being enabled. You should process the data in the temporary application buffer accordingly, for example by copying the data into a large array.

### Procedure:

1. Open the oscilloscope using [ps6000OpenUnit](#).
  - 1a. Inform the wrapper of the number of channels on the device by calling `setChannelCount`.
2. Select channels, ranges and AC/DC coupling using [ps6000SetChannel](#).
  - 2a. Inform the wrapper which channels have been enabled by calling `setEnabledChannels`.
3. Use the appropriate trigger setup functions. For programming languages that do not support structures, use the wrapper's advanced trigger setup functions.
4. Call [ps6000SetDataBuffer](#) (or for aggregated data collection [ps6000SetDataBuffers](#)) to tell the driver where your data buffer(s) is(are).
  - 4a. Register the data buffer(s) with the wrapper and set the application buffer(s) into which the data will be copied. Call `setAppAndDriverBuffers` (or `setMaxMinAppAndDriverBuffers` for aggregated data collection).
5. Start the oscilloscope running using [ps6000RunStreaming](#).
6. Loop and call `GetStreamingLatestValues` and `IsReady` to get data and flag when the wrapper is ready for data to be retrieved.
  - 6a. Call the wrapper's `AvailableData` function to obtain information on the number of samples collected and the start index in the buffer.
  - 6b. 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.
7. Process data returned to your application data buffers.
8. Call `AutoStopped` if the `autoStop` parameter has been set to `TRUE` in the call to [ps6000RunStreaming](#).

9. Repeat steps 6 to 8 until `AutoStopped` returns true or you wish to stop data collection.
10. Call [`ps6000Stop`](#), even if the `autoStop` parameter was set to `TRUE`.
11. To disconnect a device, call [`ps6000CloseUnit`](#).

## 4 Programming support and examples

Your Pico Technology SDK installation includes programming examples in various languages and development environments.

## 5 Numeric data types

Here is a list of the sizes and ranges of the numeric data types used in the PicoScope 6000 Series API.

Type	Bits	Signed or unsigned?
<code>int16_t</code>	16	signed
<code>enum</code>	32	enumerated
<code>int32_t</code>	32	signed
<code>uint32_t</code>	32	unsigned
<code>float</code>	32	signed (IEEE 754)
<code>int64_t</code>	64	signed

## 6 Enumerated types and constants

The enumerated types and constants used in the PicoScope 6000 Series API driver are defined in the file `ps6000Api.h`, which is included in the SDK. We recommend that you refer to these constants by name unless your programming language allows only numerical values.



## 7 Driver status codes

Every function in the ps6000 driver returns a **driver status code** from the list of `PICO_STATUS` values in the file `PicoStatus.h`, which is included in the Pico Technology SDK. Not all codes in `PicoStatus.h` apply to the PicoScope 6000 Series.

## 8 Glossary

**Callback.** A mechanism that the PicoScope 6000 driver uses to communicate asynchronously with your application. At design time, you add a function (a *callback* function) to your application to deal with captured data. At run time, when you request captured data from the driver, you also pass it a pointer to your function. The driver then returns control to your application, allowing it to perform other tasks until the data is ready. When this happens, the driver calls your function in a new thread to signal that the data is ready. It is then up to your function to communicate this fact to the rest of your application.

**Driver.** A program that controls a piece of hardware. The driver for the PicoScope 6000 Series oscilloscopes is supplied in the form of a 32-bit Windows DLL, `ps6000.dll`. This is used by the PicoScope software, and by user-designed applications, to control the oscilloscopes.

**PC Oscilloscope.** A virtual instrument formed by connecting a PicoScope 6000 Series oscilloscope to a computer running the PicoScope software.

**PicoScope 6000 Series.** A range of PC Oscilloscopes from Pico Technology. The common features include 5 GS/s maximum sampling rate and 8-bit resolution. The scopes are available with a range of buffer sizes up to 2 GS.

**PicoScope software.** A software product that accompanies all Pico PC Oscilloscopes. It turns your PC into an oscilloscope, spectrum analyzer.

**PRBS (pseudo-random binary sequence).** A fixed, repeating sequence of binary digits that appears random when analyzed over a time shorter than the repeat period. The waveform swings between two values: logic high (binary 1) and logic low (binary 0).

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

**USB 2.0.** The second generation of USB interface. The port supports a data transfer rate of up to 480 megabits per second.

**USB 3.0.** A USB 3.0 port uses signaling speeds of up to 5 gigabits per second and is backwards-compatible with USB 2.0 and USB 1.1.

# Index

## A

- AC coupling 65
- Aggregation 21, 44
- Analog offset 32, 65
- API function calls 26
- Arbitrary waveform generator 80
  - index modes 82
- Averaging 44
- AWG
  - buffer lengths 95
  - sample values 95

## B

- Bandwidth limiter 65
- Block mode 10, 11, 12
  - asynchronous call 13
  - callback 27
  - polling status 53
  - running 61
  - using 13
- Buffers
  - overrun 10

## C

- Callback function
  - block mode 27
  - for data 29
  - streaming mode 99
- Channels
  - enabling 65
  - settings 65
- Clock, external 75
- Closing units 28
- Constants 104
- Coupling type, setting 65

## D

- Data acquisition 21
- Data buffers
  - declaring 68
  - declaring, aggregation mode 70
  - declaring, rapid block mode 69
  - setting up 71
- DC coupling 65
- Decimation 44

- Disk space 9
- Distribution 44
- Downsampling 43
  - maximum ratio 33
  - modes 44
- Driver 10
  - status codes 105

## E

- Enabling channels 65
- Enumerated types 104
- Enumerating oscilloscopes 30
- ETS
  - overview 19
  - setting time buffers 73, 74
  - setting up 72
  - using 20
- External clock 75

## F

- Function calls 26
- Functions
  - ps6000BlockReady 27
  - ps6000CloseUnit 28
  - ps6000DataReady 29
  - ps6000EnumerateUnits 30
  - ps6000FlashLed 31
  - ps6000GetAnalogueOffset 32
  - ps6000GetMaxDownSampleRatio 33
  - ps6000GetNoOfCaptures 34
  - ps6000GetNoOfProcessedCaptures 35
  - ps6000GetStreamingLatestValues 36
  - ps6000GetTimebase 37
  - ps6000GetTimebase2 39
  - ps6000GetTriggerTimeOffset 40
  - ps6000GetTriggerTimeOffset64 41
  - ps6000GetUnitInfo 42
  - ps6000GetValues 43
  - ps6000GetValuesAsync 45
  - ps6000GetValuesBulk 46
  - ps6000GetValuesBulkAsync 47
  - ps6000GetValuesOverlapped 48
  - ps6000GetValuesOverlappedBulk 50
  - ps6000GetValuesTriggerTimeOffsetBulk 51
  - ps6000GetValuesTriggerTimeOffsetBulk64 52
  - ps6000IsReady 53
  - ps6000IsTriggerOrPulseWidthQualifierEnabled 54
  - ps6000MemorySegments 55
  - ps6000NoOfStreamingValues 56
  - ps6000OpenUnit 57

## Functions

- ps6000OpenUnitAsync 58
- ps6000OpenUnitProgress 59
- ps6000PingUnit 60
- ps6000RunBlock 61
- ps6000RunStreaming 63
- ps6000SetChannel 65
- ps6000SetDataBuffer 68
- ps6000SetDataBufferBulk 69
- ps6000SetDataBuffers 70
- ps6000SetDataBuffersBulk 71
- ps6000SetEts 72
- ps6000SetEtsTimeBuffer 73
- ps6000SetEtsTimeBuffers 74
- ps6000SetExternalClock 75
- ps6000SetNoOfCaptures 76
- ps6000SetPulseWidthQualifier 77
- ps6000SetSigGenArbitrary 80
- ps6000SetSigGenBuiltIn 84
- ps6000SetSigGenBuiltInV2 87
- ps6000SetSimpleTrigger 88
- ps6000SetTriggerChannelConditions 89
- ps6000SetTriggerChannelDirections 91
- ps6000SetTriggerChannelProperties 92
- ps6000SetTriggerDelay 94
- ps6000SigGenArbitraryMinMaxValues 95
- ps6000SigGenFrequencyToPhase 96
- ps6000SigGenSoftwareControl 97
- ps6000Stop 98
- ps6000StreamingReady 99

## H

- Hysteresis 93

## I

- Information, reading from units 42
- Input range, selecting 65

## L

- LED
  - flashing 31

## M

- Memory in scope 12
- Memory segments 55
- Microsoft Windows 9
- Multi-unit operation 25

## N

- Numeric data types 103

## O

- One-shot signals 19
- Opening a unit 57
  - checking progress 59
  - without blocking 58
- Operating system 9
- Oversampling 24

## P

- PICO\_STATUS enum type 105
- picopp.inf 10
- picopp.sys 10
- PicoScope 6000 Series 7
- PicoScope software 10
- Processor 9
- PS6000\_CONDITION\_constants 79, 90
- PS6000\_LEVEL constant 93
- PS6000\_LOST\_DATA constant 10
- PS6000\_MAX\_VALUE constant 10
- PS6000\_MIN\_VALUE constant 10
- PS6000\_PWQ\_CONDITIONS structure 79
- PS6000\_TIME\_UNITS constant 40
- PS6000\_TRIGGER\_CHANNEL\_PROPERTIES structure 93
- PS6000\_TRIGGER\_CONDITIONS structure 90
- PS6000\_WINDOW constant 93
- Pulse-width qualifier 77
  - conditions 79
  - requesting status 54

## R

- Rapid block mode 14
  - setting number of captures 76
  - using 14
- Resolution, vertical 24
- Retrieving data 43, 45
  - block mode, deferred 48
  - rapid block mode 46
  - rapid block mode with callback 47
  - rapid block mode, deferred 50
  - stored 22
  - streaming mode 36
- Retrieving times
  - rapid block mode 51, 52

selecting 65

## S

- Sampling rate
  - maximum 12
- Scaling 10
- Serial numbers 30
- Signal generator 13
  - arbitrary waveforms 80
  - built-in waveforms 84, 87
  - calculating phase 96
  - software trigger 97
- Software license conditions 8
- Status codes 105
- Stopping sampling 98
- Streaming mode 11, 21
  - callback 99
  - getting number of samples 56
  - retrieving data 36
  - running 63
  - using 22
- Synchronising units 25
- System memory 9
- System requirements 9

## T

- Threshold voltage 10
- Time buffers
  - setting for ETS 73, 74
- Timebase 24
  - calculating 37, 39
- Trademarks 8
- Trigger 10
  - channel properties 92
  - conditions 89, 90
  - delay 94
  - directions 91
  - pulse-width qualifier 77
  - pulse-width qualifier conditions 79
  - requesting status 54
  - setting up 88
  - time offset 40, 41

## U

- USB 9
  - hub 25

## V

- Vertical resolution 24
- Voltage ranges 10

## W

- Wrapper functions 100

**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](mailto:sales@picotech.com)  
[support@picotech.com](mailto:support@picotech.com)

[www.picotech.com](http://www.picotech.com)

**US headquarters:**

Pico Technology  
320 N Glenwood Blvd  
Tyler  
TX 75702  
USA

Tel: +1 800 591 2796

[sales@picotech.com](mailto:sales@picotech.com)  
[support@picotech.com](mailto: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](mailto:pico.asia-pacific@picotech.com)