PicoScope® 6000E Series

(ps6000a API)

Programmer's Guide

Contents

1   Introduction

1.1   Welcome

The **PicoScope 6000E 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 ps6000a API (application programming interface) for the PicoScope 6000E Series scopes. For more information on the hardware, see the *PicoScope 6000E Series Data Sheet*.

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. 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 `ps6000a.dll` dynamic link library in the `lib` subdirectory of your Pico Technology SDK installation directory allows you to program a PicoScope 6000E 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 available on the *picotech* channel of GitHub. 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 6000E 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
Operating system	All desktop versions of Windows with mainstream support. 32-bit and 64-bit versions.
Processor, Memory, Free disk space	As required by the operating system
Ports	USB 2.0 or 3.0 port

Using with Linux
Beta drivers are available for various Linux distributions. Instructions are available on our website.

Using with macOS
A software development kit (SDK) for macOS can be downloaded from our website.

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 ps6000a driver offers three different methods of recording data, all of which support USB 1.1, USB 2.0, and USB 3.0. Your oscilloscope will operate very slowly on a USB 1.1 port; USB 2.0 or faster is strongly recommended for best performance.

2.2   Driver

Your application will communicate with a PicoScope 6000 library called `ps6000a.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 depends on another library, `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

You can set a device input channel to any voltage range from ±10 mV to ±20 V with the ps6000aSetChannelOn() function. Each sample is scaled to 16 bits. The minimum and maximum values returned to your application depend on the sampling resolution in use and can be queried by ps6000aGetAdcLimits(). This function replies with the following values:

Resolution	8 bits	10 bits	12 bits
Voltage	Value returned		
maximum	+32 512 (0x7F00)	+32 704 (0x7FC0)	+32 736 (0x7FE0)
zero	0	0	0
minimum	−32 512 (0x8100)	−32 704 (0x8040)	−32 736 (0x8020)

Example at 8-bit resolution

1. Call ps6000aSetChannelOn() with range set to PICO_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.

+1 V	7F00	+32 512
+500 mV	3F80	+16 256
0 V	0000	0
−500 mV	C080	−16 256
−1 V	8100	−32 512
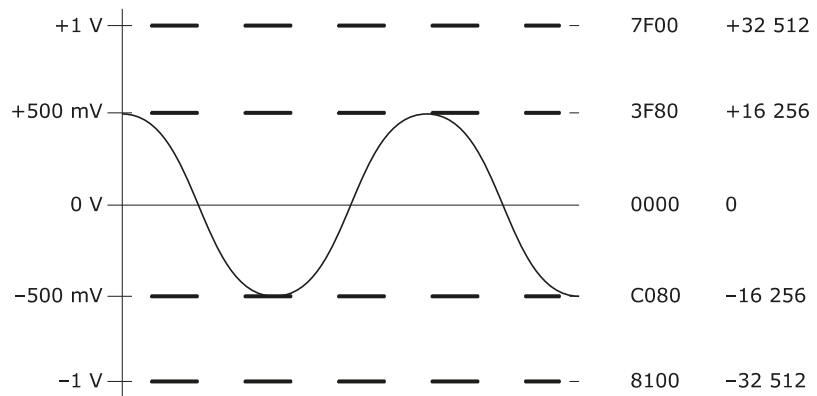
Digital inputs (with optional MSO pods)

See ps6000aSetDigitalPort().

2.4   MSO digital data

Applicability

Any device with MSO pods attached. MSO pods are automatically recognized by the driver when connected.

A PicoScope MSO has two 8-bit digital ports—**Digital 1** and **Digital 2**—making a total of 16 digital channels.

Use the ps6000aSetDataBuffer() and ps6000aSetDataBuffers() functions to set up buffers into which the driver will write data from each port individually. For compatibility with the analog channels, each buffer is an array of 16-bit words. The 8-bit port data occupies the lower 8 bits of the word. The upper 8 bits of the word are undefined.

	Digital 2 buffer	**Digital 1 buffer**
Sample$_0$	[XXXXXXXX,2D7...2D0]$_0$	[XXXXXXXX,1D7...1D0]$_0$
...	...	...
Sample$_{n-1}$	[XXXXXXXX,2D7...2D0]$_{n-1}$	[XXXXXXXX,1D7...1D0]$_{n-1}$

Retrieving stored digital data

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

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

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

// Mask Digital 1 values to get lower 8 bits,
// then OR with shifted Digital 2 bits to get 16-bit word
portValue |= 0x00ff & appDigiBuffers[0][i];

for (bit = 0; bit < 16; bit++)
{
  // Shift value 32768 (binary 1000 0000 0000 0000).
  // AND with value to get 1 or 0 for channel.
  // Order will be 2D7 to 2D0, then 1D7 to 1D0.

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

2.5   Triggering

PicoScope 6000E 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:

- ps6000aSetTriggerChannelConditions()
- ps6000aSetTriggerChannelDirections()
- ps6000aSetTriggerChannelProperties()
- ps6000aSetTriggerDelay() (optional)

These can be run collectively by calling ps6000aSetSimpleTrigger(), 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 functions:

- ps6000aSetPulseWidthQualifierProperties()
- ps6000aSetPulseWidthQualifierConditions()
- ps6000aSetPulseWidthQualifierDirections()

2.6 Sampling modes

PicoScope 6000E 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

  The driver can return data asynchronously using a callback, which 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.

  If you do not wish to use a callback, you can poll the driver instead.

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.** This mode enables long periods of data collection. In raw mode (no downsampling) it provides fast data transfer of unlimited amounts of data at up to 312 MB/s (3.2 ns per sample) in 8-bit mode with USB 3.0.

  If downsampling is enabled, raw data can be sampled at up to 1.25 GS/s for a single channel in 8-bit mode. Downsampled data is returned while capturing is in progress, at up to 312 MB/s. The raw data can then be retrieved after the capture is complete. The number of raw samples is limited by the memory available on the device, the selected resolution and the number of channels enabled.

  Triggering is supported in this mode.

Note: The oversampling feature of older PicoScope oscilloscopes has been replaced by PICO_RATIO_MODE_AVERAGE.

2.6.1 Block mode

In **block mode**, the computer prompts a PicoScope 6000E 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 `ps6000aMemorySegments()`) and the sampling resolution.

- **Sampling rate.** A PicoScope 6000E 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 6000E Series Data Sheet* for the specifications that apply to your scope model.

- **Setup time.** The driver normally performs a number of setup operations, which can take up to 50 milliseconds, before collecting each block of data. If you need to collect data with the minimum time interval between blocks, use rapid block mode and avoid calling setup functions between calls to `ps6000aRunBlock()`, `ps6000aStop()` and `ps6000aGetValues()`.

- **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 `ps6000aMemorySegments()` or `ps6000aMemorySegmentsBySamples()`.

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

See Using block mode for programming details.

2.6.1.1   Using block mode

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

1. Open the oscilloscope using `ps6000aOpenUnit()`.
2. Select channel ranges and AC/DC/50 Ω coupling using `ps6000aSetChannelOn()` and `ps6000aSetChannelOff()`.
3. Using `ps6000aGetTimebase()`, select timebases until the required nanoseconds per sample is located.
4. Use the trigger setup functions `ps6000aSetTriggerChannelConditions()`, `ps6000aSetTriggerChannelDirections()` and `ps6000aSetTriggerChannelProperties()` to set up the trigger if required.
5. Start the oscilloscope running using `ps6000aRunBlock()`.
6. Wait until the oscilloscope is ready using the `ps6000aBlockReady()` callback (or poll using `ps6000aIsReady()`).
7. Use `ps6000aSetDataBuffer()` 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 `ps6000aGetValues()`.
9. Display the data.
10. Repeat steps 5 to 9.
11. Stop the oscilloscope using `ps6000aStop()`.
12. Request new views of stored data using different downsampling parameters: see Retrieving stored data.
13. Close the device using `ps6000aCloseUnit()`.
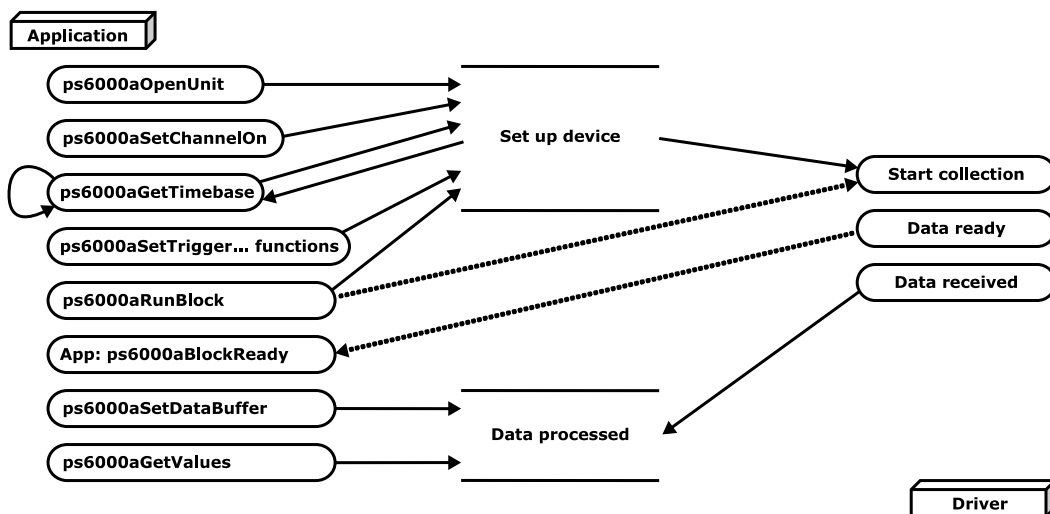
2.6.1.2   Asynchronous calls in block mode

`ps6000aGetValues()` may take a long time to complete if a large amount of data is being collected. To avoid hanging the calling thread, it is possible to call `ps6000aGetValuesAsync()` instead. This immediately returns control to the calling thread, which then has the option of waiting for the data or calling `ps6000aStop()` to abort the operation.

2.6.2   Rapid block mode

In normal [block mode,](#) the PicoScope 6000E 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.

2.6.2.1   Using rapid block mode

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

Without aggregation
1.   Open the oscilloscope using `ps6000aOpenUnit()`.
2.   Select channel ranges and AC/DC coupling using `ps6000aSetChannelOn()` and `ps6000aSetChannelOff()`.
3.   Set the number of memory segments equal to or greater than the number of captures required using `ps6000aMemorySegments()`. Use `ps6000aSetNoOfCaptures()` before each run to specify the number of waveforms to capture.
4.   Using `ps6000aGetTimebase()`, select timebases until the required nanoseconds per sample is located.
5.   Use the trigger setup functions `ps6000aSetTriggerChannelConditions()`, `ps6000aSetTriggerChannelDirections()` and `ps6000aSetTriggerChannelProperties()` to set up the trigger if required.
6.   Start the oscilloscope running using `ps6000aRunBlock()`.
7.   Wait until the oscilloscope is ready using the `ps6000aBlockReady()` callback.
8.   Use `ps6000aSetDataBuffer()` 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 `ps6000aGetValuesBulk()`.
10.   Retrieve the time offset for each data segment using `ps6000aGetValuesTriggerTimeOffsetBulk()`.
11.   Display the data.
12.   Repeat steps 6 to 11 if necessary.
13.   Stop the oscilloscope using `ps6000aStop()`.
14.   Close the device using `ps6000aCloseUnit()`.

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

8a.   Call `ps6000aSetDataBuffers()` to set up one pair of buffers for every waveform segment required.
9a.   Call `ps6000aGetValuesBulk()` for each pair of buffers.
10a.   Retrieve the time offset for each data segment using `ps6000aGetValuesTriggerTimeOffsetBulk()`.

Continue from step 11 above.

ps6000apg-1   15

2.6.2.2   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
ps6000aSetNoOfCaptures(handle, MAX_WAVEFORMS);

pParameter = false;
ps6000aRunBlock
(
  handle,
  0,                  // noOfPreTriggerSamples
  10000,              // noOfPostTriggerSamples
  1,                  // timebase to be used
  &timeIndisposedMs,
  0,                  // segment index
  lpReady,
  &pParameter
);
```

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

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

PICO_ACTION action = PICO_CLEAR_ALL | PICO_ADD;
int32_t first_segment_to_read = 10;

for (int32_t i = 0; i < 10; i++)
{
  for (int32_t c = PICO_CHANNEL_A; c <= PICO_CHANNEL_D; c++)
  {
    ps6000aSetDataBuffer
    (
      handle,
      c,
      buffer[c][i],
      MAX_SAMPLES,
      PICO_INT16_T,
      first_segment_to_read + i,
      PICO_RATIO_MODE_RAW,
      action
    );
    action = PICO_ADD;
  }
}
```

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

```
ps6000aGetValuesBulk
(
  handle,
  0,                      // startIndex
  &noOfSamples,           // set to MAX_SAMPLES on entering the function
  10,                     // fromSegmentIndex
  19,                     // toSegmentIndex
  1,                      // downsampling ratio
  PICO_RATIO_MODE_RAW,    // downsampling ratio mode
  overflow                // indices 0 to 9 will be populated (index always
    starts from 0)
)
```

Comments: the number of samples could be up to `noOfPreTriggerSamples + noOfPostTriggerSamples`, the values set in `ps6000aRunBlock()`. The samples are returned starting from the sample index. This function does not support aggregation. The above segments start at 10 and finish at 19 inclusive. It is possible for `fromSegmentIndex` to wrap around to `toSegmentIndex`, for example by setting `fromSegmentIndex` to 98 and `toSegmentIndex` to 7.

```
ps6000aGetValuesTriggerTimeOffsetBulk
(
  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`, for example if `fromSegmentIndex` is set to 98 and `toSegmentIndex` to 7.

2.6.2.3   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
ps6000aSetNoOfCaptures(handle, MAX_WAVEFORMS);

pParameter = false;
ps6000aRunBlock
(
  handle,
  0,                 // noOfPreTriggerSamples,
  1000000,           // noOfPostTriggerSamples,
  1,                 // timebase to be used,
  &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.

```
PICO_ACTION action = PICO_CLEAR_ALL | PICO_ADD;

for (int32_t c = PICO_CHANNEL_A; c <= PICO_CHANNEL_D; c++)
{
  ps6000aSetDataBuffers
  (
    handle,
    c,
    bufferMax[c],
    bufferMin[c]
    MAX_SAMPLES,
    PICO_INT16_T,
    0,
    PICO_RATIO_MODE_AGGREGATE,
    action
  );
  action = PICO_ADD;
}
```
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++)
{
  ps6000aGetValues
```

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

  ps6000aGetTriggerTimeOffset
  (
    handle,
    &time,
    &timeUnits,
    index
  )
}
```

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

2.6.3   Streaming mode

Streaming mode can capture data without the gaps that occur between blocks when using [block mode.](block) This makes it suitable for **high-speed data acquisition**, allowing you to capture long data sets limited only by the computer's memory. (At the highest sampling rates, the size of the device's capture buffer may limit the capture size.)

The device can return either raw or downsampled data to your application while streaming is in progress. When downsampled data is returned, the raw samples remain stored on the device.

- **Downsampling.** The driver can return either raw or downsampled data. You should set up the number of buffers needed to accept the requested data. Aggregation requires two buffers, one for the minimum values and one for the maximum values. Other downsampling modes require only a single buffer.

See [Using streaming mode](using) for programming details.

2.6.3.1   Using streaming mode

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

1.   Open the oscilloscope using `ps6000aOpenUnit()`.
2.   Select channels, ranges and AC/DC/50 Ω coupling using `ps6000aSetChannelOn()` and `ps6000aSetChannelOff()`.
3.   Use the trigger setup functions `ps6000aSetTriggerChannelConditions()`, `ps6000aSetTriggerChannelDirections()` and `ps6000aSetTriggerChannelProperties()` to set up the trigger if required.
4.   Call `ps6000aSetDataBuffer()` to tell the driver where your data buffer is.
5.   Set up aggregation and start the oscilloscope running using `ps6000aRunStreaming()`.
6.   Call `ps6000aGetStreamingLatestValues()` to get data. If the function runs out of buffer space, return to step 4.
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 `ps6000aStop()`, even if `autoStop` is enabled.
9.   Request new views of stored data using different downsampling parameters: see <u>Retrieving stored data</u>.
10.   Close the device using `ps6000aCloseUnit()`.
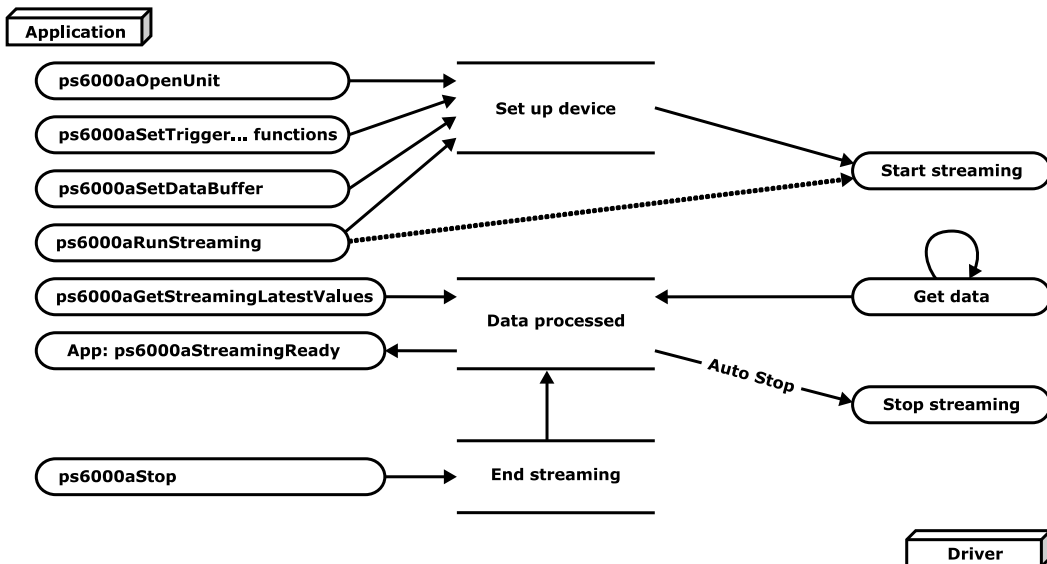
2.6.4   Retrieving stored data

You can retrieve data from the `ps6000a` driver with a different [downsampling](#) factor when [ps6000aRunBlock()](#) or [ps6000aRunStreaming()](#) has already been called and has successfully captured all the data. Use [ps6000aGetValuesAsync()](#).

```
Application

   ps6000aSetDataBuffer  ─────────────┐
                                      ▼
   ps6000aGetValuesAsync  ────────►  Data processed

   App: ps6000aDataReady  ◄───────────┘
```
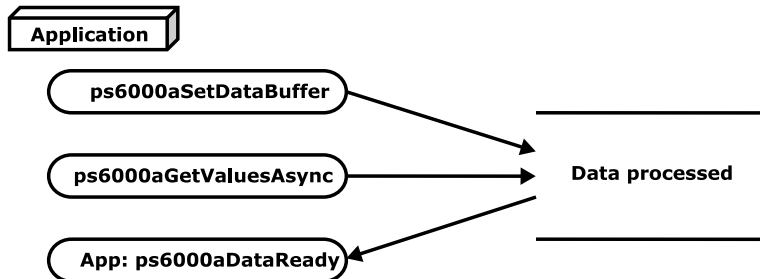
2.7   Timebases

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

Applicability	Calls to [ps6000aGetTimebase()](#)

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 <u>PicoScope 6000E Series oscilloscopes</u> at the same time, depending on the capabilities of the PC. Each oscilloscope must be connected to a separate USB port. The <u>ps6000aOpenUnit()</u> 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 ps6000aBlockReady(...)
// define callback function specific to application

handle1 = ps6000aOpenUnit()
handle2 = ps6000aOpenUnit()

ps6000aSetChannelOn(handle1)
// set up unit 1
ps6000aRunBlock(handle1)

ps6000aSetChannelOn(handle2)
// set up unit 2
ps6000aRunBlock(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 **10 MHz** clock reference input or a trigger into the **Aux Trig** input to provide some degree of synchronization between multiple oscilloscopes.

2.9   Handling intelligent probe interactions

The PicoScope 6000E Series has an intelligent probe interface, which supplies power to the probe as well as allowing the scope to configure and interrogate the probe. Your application can choose to be alerted whenever a probe is connected or disconnected, or when its status changes.

Probe interactions use a callback mechanism, available in C and similar languages.

Applicability	All models
Note	In addition to `ps6000aApi.h`, you must also include `PicoDeviceEnums.h`. This file contains definitions of enumerated types that describe the intelligent probes.

Procedure
1. Define your own function to receive probe interaction callbacks.
2. Call `ps6000aOpenUnit()` to obtain a device handle.
3. Call `ps6000aSetProbeInteractionCallback()` to register your probe interaction callback function.
4. Capture data using the desired sampling mode. See Sampling modes for details.
5. Call `ps6000aCloseUnit()` to release the device handle. The makes the scope device available to other applications.
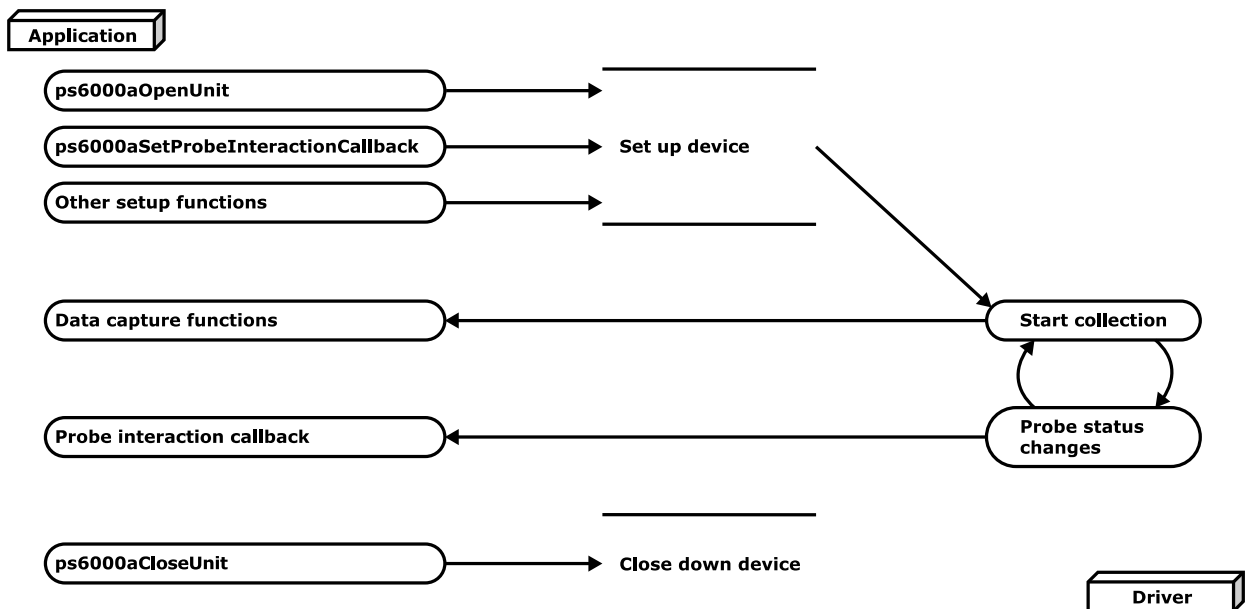
3   API functions

The PicoScope 6000E Series API exports the following functions for you to use in your own applications for Microsoft Windows. Similar APIs are available for other platforms: see [www.picotech.com > Downloads](http://www.picotech.com) for details. All functions are C functions using the standard call naming convention (`__stdcall`). They are all exported with both decorated and undecorated names.

3.1   ps6000aChannelCombinationsStateless - get possible channel combinations

```
PICO_STATUS ps6000aChannelCombinationsStateless
(
  int16_t                  handle,
  PICO_CHANNEL_FLAGS     * channelFlagsCombinations,
  uint32_t               * nChannelCombinations,
  PICO_DEVICE_RESOLUTION   resolution,
  uint32_t                 timebase
)
```

This function returns a list of the possible channel combinations given a proposed configuration (resolution and timebase) of the oscilloscope. It does not change the configuration of the oscilloscope.

Applicability

All modes

Arguments

`handle,` the device identifier returned by `ps6000aOpenUnit()`.

`* channelFlagsCombinations,` on exit, a list of possible channel combinations. See `PicoDeviceEnums.h`.

`* nChannelCombinations,` on exit, the length of the `channelFlagsCombinations` list.

`resolution,` the proposed vertical resolution of the oscilloscope.

`timebase,` the proposed timebase number.

Returns

```
PICO_OK
PICO_INVALID_HANDLE
PICO_NULL_PARAMETER
PICO_INVALID_PARAMETER
```

3.2   ps6000aCheckForUpdate - is firmware update available?

```
PICO_STATUS ps6000aCheckForUpdate
(
  int16_t              handle,
  PICO_FIRMWARE_INFO * firmwareInfos,
  int16_t            * nFirmwareInfos,
  uint16_t           * updatesRequired
)
```

This function checks whether a firmware update for the device is available.

Applicability

All modes

Arguments

`handle`, the device identifier returned by `ps6000aOpenUnit()`.

`firmwareInfos`, on exit, a list of `PICO_FIRMWARE_INFO` structures containing firmware updates information.

`nFirmwareInfos`, on exit, the number of structures in the `firmwareInfos` list.

`* updatesRequired`, on exit, whether the device needs updates (1) or not (0).

Returns

```
PICO_OK
PICO_HANDLE_INVALID
PICO_USER_CALLBACK
PICO_DRIVER_FUNCTION
```

3.3   ps6000aCloseUnit - close a scope device

```
PICO_STATUS ps6000aCloseUnit
(
  int16_t    handle
)
```

This function shuts down a PicoScope 6000E Series oscilloscope.

Applicability

All modes

Arguments

handle,   the device identifier returned by ps6000aOpenUnit().

Returns

PICO_OK
PICO_HANDLE_INVALID
PICO_USER_CALLBACK
PICO_DRIVER_FUNCTION

3.4   ps6000aEnumerateUnits - get a list of unopened units

```
PICO_STATUS ps6000aEnumerateUnits
(
  int16_t   * count,
  int8_t    * serials,
  int16_t   * serialLth
)
```

This function counts the number of PicoScope 6000 (A API) units connected to the computer, and returns a list of serial numbers and other optional information as a string. Note that this function can only detect devices that are not yet being controlled by an application. To query opened devices, use ps6000aGetUnitInfo().

Applicability

All modes

Arguments

* count, on exit, the number of PicoScope 6000 (A API) units found.

* serials, if an empty string on entry, serials is populated on exit with a list of serial numbers separated by commas and terminated by a final null. Example:

> AQ005/139,VDR61/356,ZOR14/107

On entry, serials can optionally contain the following parameter(s) to request information:

> -v : model number
>
> -c : calibration date
>
> -h : hardware version
>
> -u : USB version
>
> -f : firmware version

Example (any separator character can be used):

> -v:-c:-h:-u:-f

On exit, with all the above parameters specified, each serial number has the requested information appended in the following format:

> AQ005/139[6425E,01Jan21,769,2.0,1.7.16.0]

serials can be NULL if device information or serial numbers are not required.

* serialLth, on entry, the length of the int8_t buffer pointed to by serials; on exit, the length of the string written to serials

Returns

```
PICO_OK
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   ps6000aFlashLed - flash the front-panel LED

```
PICO_STATUS ps6000aFlashLed
(
  int16_t    handle,
  int16_t    start
)
```

This function flashes the status/trigger LED on the front of the scope without blocking the calling thread. Calls to ps6000aRunStreaming() and ps6000aRunBlock() cancel any flashing started by this function. It is not possible to set the LED to be constantly illuminated, as this state is used to indicate that the scope has not been initialized.

Applicability

All modes

Arguments

handle, the device identifier returned by ps6000aOpenUnit().

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

Returns

PICO_OK
PICO_HANDLE_INVALID
PICO_BUSY
PICO_DRIVER_FUNCTION
PICO_NOT_RESPONDING

3.6   ps6000aGetAdcLimits - get min and max sample values

```
PICO_STATUS ps6000aGetAdcLimits
(
  int16_t                 handle,
  PICO_DEVICE_RESOLUTION  resolution,
  int16_t               * minValue,
  int16_t               * maxValue
)
```

This function gets the maximum and minimum sample values that the ADC can produce at a given resolution.

Applicability

All modes

Arguments

`handle,` the device identifier returned by `ps6000aOpenUnit()`.

`resolution,` the vertical resolution about which you require information.

`* minValue,` the minimum sample value.

`* maxValue,` the maximum sample value.

Returns

`PICO_OK`
`PICO_INVALID_HANDLE`
`PICO_DRIVER_FUNCTION`
`PICO_NULL_PARAMETER` (if both `maxValue` and `minValue` are `NULL`)

3.7   ps6000aGetAnalogueOffsetLimits - get analog offset information

```
PICO_STATUS ps6000aGetAnalogueOffsetLimits
(
  int16_t                  handle,
  PICO_CONNECT_PROBE_RANGE range
  PICO_COUPLING            coupling
  double                 * maximumVoltage,
  double                 * minimumVoltage
)
```

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

Applicability

All modes

Arguments

`handle`, the device identifier returned by `ps6000aOpenUnit()`.

`range`, the voltage range for which minimum and maximum voltages are required

`coupling`, the type of AC/DC/50 Ω coupling used

`* maximumVoltage`, on output, the maximum analog offset voltage allowed for the range. Set to `NULL` if not required.

`* minimumVoltage`, on output, the minimum analog offset voltage allowed for the range. Set to `NULL` if not required.

Returns

`PICO_OK`
`PICO_INVALID_HANDLE`
`PICO_DRIVER_FUNCTION`
`PICO_INVALID_VOLTAGE_RANGE`
`PICO_NULL_PARAMETER` (if both `maximumVoltage` and `minimumVoltage` are `NULL`)
`PICO_INVALID_COUPLING`

3.8   ps6000aGetDeviceResolution – retrieve the device resolution

```
PICO_STATUS ps6000aGetDeviceResolution
(
  int16_t                   handle,
  PICO_DEVICE_RESOLUTION * resolution
)
```

This function retrieves the vertical resolution of the oscilloscope.

Applicability

All modes

Arguments

`handle,` the device identifier returned by `ps6000aOpenUnit()`.

`* resolution, on exit,` the resolution of the device.

Returns

`PICO_OK` or other code from `PicoStatus.h`

3.9　ps6000aGetMaximumAvailableMemory - depending on hardware resolution

```
PICO_STATUS ps6000aGetMaximumAvailableMemory
(
  int16_t                 handle,
  uint64_t              * nMaxSamples,
  PICO_DEVICE_RESOLUTION   resolution
)
```

This function returns the maximum number of samples that can be stored at a given hardware resolution.

Applicability

All modes

Arguments

`handle`, the device identifier returned by `ps6000aOpenUnit()`.

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

`resolution`, the resolution in bits.

Returns

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

3.10   ps6000aGetMinimumTimebaseStateless - find fastest available timebase

```
PICO_STATUS ps6000aGetMinimumTimebaseStateless
(
  int16_t                handle,
  PICO_CHANNEL_FLAGS     enabledChannelFlags,
  uint32_t             * timebase,
  double               * timeInterval,
  PICO_DEVICE_RESOLUTION  resolution
)
```

This function returns the shortest timebase that could be selected with a proposed configuration of the oscilloscope. It does not set the oscilloscope to the proposed configuration.

Applicability

All modes

Arguments

`handle,` the device identifier returned by `ps6000aOpenUnit()`.

`enabledChannelFlags,` a bit field indicating which channels are enabled in the proposed configuration. Channel A is bit 0 and so on.

`* timebase,` on exit, the number of the shortest timebase possible with the proposed configuration.

`* timeInterval,` on exit, the sample period in seconds corresponding to `.timebase`.

`resolution,` the vertical resolution in the proposed configuration.

Returns

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

3.11   ps6000aGetNoOfCaptures - query how many captures made

```
PICO_STATUS ps6000aGetNoOfCaptures
(
  int16_t     handle,
  uint64_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 ps6000aStop().

The returned value (nCaptures) can then be used to iterate through the number of segments using ps6000aGetValues(), or in a single call to ps6000aGetValuesBulk() where it is used to calculate the toSegmentIndex parameter.

Applicability
All modes

Arguments
handle, the device identifier returned by ps6000aOpenUnit().

nCaptures, on output, the number of available captures that has been collected from calling ps6000aRunBlock().

Returns
PICO_OK
PICO_INVALID_HANDLE
PICO_NO_SAMPLES_AVAILABLE
PICO_NULL_PARAMETER
PICO_INVALID_PARAMETER
PICO_SEGMENT_OUT_OF_RANGE
PICO_TOO_MANY_SAMPLES

3.12   ps6000aGetNoOfProcessedCaptures - query how many captures processed

```
PICO_STATUS ps6000aGetNoOfProcessedCaptures
(
  int16_t     handle,
  uint64_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 `ps6000aRunBlock()`. It is for use in rapid block mode, alongside the `ps6000aGetValuesOverlapped()` function, when the driver is set to transfer data from the device automatically as soon as the `ps6000aRunBlock()` function is called. You can call `ps6000aGetNoOfProcessedCaptures()` during device capture, after collection has completed or after interrupting waveform collection by calling `ps6000aStop()`.

The returned value (`nProcessedCaptures`) can then be used to iterate through the number of segments using `ps6000aGetValues()`, or in a single call to `ps6000aGetValuesBulk()`, where it is used to calculate the `toSegmentIndex` parameter.

When capture is stopped
If `nProcessedCaptures` = 0, you will also need to call `ps6000aGetNoOfCaptures()`, in order to determine how many waveform segments were captured, before calling `ps6000aGetValues()` or `ps6000aGetValuesBulk()`.

Applicability

Rapid block mode

Arguments

`handle`,   the device identifier returned by `ps6000aOpenUnit()`.

`* nProcessedCaptures`, on exit, the number of waveforms captured and processed.

Returns

```
PICO_OK
PICO_INVALID_HANDLE
PICO_INVALID_PARAMETER
```

3.13   ps6000aGetStreamingLatestValues - read streaming data

```
PICO_STATUS ps6000aGetStreamingLatestValues
(
  int16_t                         handle,
  PICO_STREAMING_DATA_INFO        * streamingDataInfo,
  uint64_t                        nStreamingDataInfos,
  PICO_STREAMING_DATA_TRIGGER_INFO * triggerInfo
)
```

This function populates the `streamingDataInfo` structure with a description of the samples available and the `triggerInfo` structure to indicate that a trigger has occurred and at what location.

Applicability

Streaming mode only

Arguments

`handle`, the device identifier returned by `ps6000aOpenUnit()`.

`* streamingDataInfo`, a list of structures. See `PICO_STREAMING_DATA_INFO`.

`nStreamingDataInfos`, the number of structures in the `streamingDataInfo` list.

`* triggerInfo`, a list of structures containing trigger information. See `PICO_STREAMING_DATA_TRIGGER_INFO`.

Returns	PICO_OK
	PICO_WAITING_FOR_DATA_BUFFERS - indicates that you need to call `ps6000aSetDataBuffer()` again

3.13.1   PICO_STREAMING_DATA_INFO

A list of structures of this type is passed to <u>ps6000aGetStreamingLatestValues()</u> in the `streamingDataInfo` argument to specify parameters for streaming mode data capture. It is defined as follows:

```
typedef struct tPicoStreamingDataInfo
{
  PICO_CHANNEL     channel_;
  PICO_RATIO_MODE mode_;
  PICO_DATA_TYPE   type_;
  int32_t          noOfSamples_;
  uint64_t         bufferIndex_;
  int32_t          startIndex_;
  int16_t          overflow_;
} PICO_STREAMING_DATA_INFO;
```

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

Elements

`channel_`,   the oscilloscope channel that the parameters apply to.

`mode_`,   the downsampling mode to use.

`type_`,   the data type to use for the sample data.

`noOfSamples_`,   the number of samples made available by the driver.

`bufferIndex_`,   an index to the starting sample within the specified waveform buffer.

`startIndex_`,   an index to the waveform buffer within the capture buffer.

`overflow_`,   a flag indicating whether a sample value overflowed (1) or not (0).

3.13.2   PICO_STREAMING_DATA_TRIGGER_INFO

A structure of this type is returned by <u>ps6000aGetStreamingLatestValues()</u> in the `triggerInfo` argument to return information about trigger events.

```
typedef struct tPicoStreamingDataTriggerInfo
{
  uint64_t  triggerAt_;
  int16_t   triggered_;
  int16_t   autoStop_;
} PICO_STREAMING_DATA_TRIGGER_INFO;
```

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

Elements

`triggerAt_`,   an index to the sample on which the trigger occurred.

`triggered_`,   a flag indicating whether a trigger occurred (1) or did not occur (0).

`autoStop_`,   a flag indicating whether the oscilloscope was in autoStop mode (1) or not (0).

3.14   ps6000aGetTimebase - get available timebases

```
PICO_STATUS ps6000aGetTimebase
(
  int16_t     handle,
  uint32_t    timebase,
  uint64_t    noSamples,
  double    * timeIntervalNanoseconds,
  uint64_t  * maxSamples
  uint64_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 ps6000aSetChannelOn() or ps6000aSetChannelOff().

The easiest way to find a suitable timebase is to call ps6000aNearestSampleIntervalStateless(). Alternatively, you can estimate the timebase number that you require using the information in the timebase guide, then pass this timebase to ps6000aGetTimebase() and check the returned timeIntervalNanoseconds argument. Repeat until you obtain the time interval that you need.

Applicability

All modes

Arguments

handle, the device identifier returned by ps6000aOpenUnit().

timebase, see timebase guide.

noSamples, the number of samples required. This value is used to calculate the most suitable time interval.

timeIntervalNanoseconds, on exit, the time interval between readings at the selected timebase. Use NULL if not required.

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

segmentIndex, the index of the memory segment to use.

Returns

PICO_OK
PICO_INVALID_HANDLE
PICO_TOO_MANY_SAMPLES
PICO_INVALID_CHANNEL
PICO_INVALID_TIMEBASE
PICO_INVALID_PARAMETER
PICO_SEGMENT_OUT_OF_RANGE
PICO_DRIVER_FUNCTION

3.15   ps6000aGetTriggerInfo - get trigger timing information

```
PICO_STATUS ps6000aGetTriggerInfo
(
  int16_t              handle
  PICO_TRIGGER_INFO  * triggerInfo,
  uint64_t             firstSegmentIndex,
  uint64_t             segmentCount
)
```

This function gets trigger timing information from one or more buffer segments.

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

Applicability

Block mode, rapid block mode

Arguments

`handle,`   the device identifier returned by `ps6000aOpenUnit()`.

`* triggerInfo,`   a list of structures, one for each buffer segment, containing trigger information.

`firstSegmentIndex,`   the index of the first segment of interest.

`segmentCount,`   the number of segments of interest.

Returns

```
PICO_OK
PICO_INVALID_HANDLE
PICO_DEVICE_SAMPLING
PICO_SEGMENT_OUT_OF_RANGE
PICO_NULL_PARAMETER
PICO_NO_SAMPLES_AVAILABLE
PICO_DRIVER_FUNCTION
```

3.15.1   PICO_TRIGGER_INFO

A list of structures of this type containing trigger information is written by [ps6000aGetTriggerInfo()](#) to the `triggerInfo` location. The structure is defined as follows:

```
typedef struct tPicoTriggerInfo
{
  PICO_STATUS       status;
  uint64_t          segmentIndex;
  uint64_t          triggerIndex;
  double            triggerTime;
  PICO_TIME_UNITS   timeUnits;
  uint64_t          missedTriggers;
  uint64_t          timeStampCounter;
} PICO_TRIGGER_INFO;
```

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

Elements

`status,` indicates success or failure.

`segmentIndex,` the number of the segment.

`triggerIndex,` the index of the sample at which the trigger ocurred.

`triggerTime,` the time at which the trigger occurred.

`timeUnits,` the unit multiplier to use with `triggerTime`.

`missedTriggers,` the number of trigger events, if any, detected since the start of previous segment.

`timeStampCounter,` the time in samples from the first capture to the current capture. The status `PICO_DEVICE_TIME_STAMP_RESET` indicates that the trigger time has started over.

3.16   ps6000aGetTriggerTimeOffset - get timing corrections

```
PICO_STATUS ps6000aGetTriggerTimeOffset
(
  int16_t              handle
  int64_t            * time,
  PICO_TIME_UNITS    * timeUnits,
  uint64_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.

Applicability

Block mode, rapid block mode

Arguments

`handle`, the device identifier returned by `ps6000aOpenUnit()`.

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

`timeUnits`, on exit, the time units in which `time` is measured. The possible values are:

   PICO_FS
   PICO_PS
   PICO_NS
   PICO_US
   PICO_MS
   PICO_S

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

Returns

```
PICO_OK
PICO_INVALID_HANDLE
PICO_DEVICE_SAMPLING
PICO_SEGMENT_OUT_OF_RANGE
PICO_NULL_PARAMETER
PICO_NO_SAMPLES_AVAILABLE
PICO_DRIVER_FUNCTION
```

3.17   ps6000aGetUnitInfo - get information about device

```
PICO_STATUS ps6000aGetUnitInfo
(
  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. To find out about unopened devices, call ps6000aEnumerateUnits().

Applicability
All modes

Arguments
handle,   identifies the device from which information is required. If an invalid handle is passed, the error code from the last unit that failed to open is returned.

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

stringLength,   the maximum number of int8_t values that may be written to string.

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

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

Returns
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 ps6000a 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

3.18   ps6000aGetValues - get block mode data

```
PICO_STATUS ps6000aGetValues
(
  int16_t              handle,
  uint64_t             startIndex,
  uint64_t           * noOfSamples,
  uint64_t             downSampleRatio,
  PICO_RATIO_MODE      downSampleRatioMode,
  uint64_t             segmentIndex,
  int16_t            * overflow
)
```

This function returns block-mode data, with downsampling if requested, starting at the specified sample number. It is used to get the stored data from the oscilloscope after data collection has stopped.

Applicability

All modes.

Arguments

handle, the device identifier returned by ps6000aOpenUnit().

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

noOfSamples, on entry, the number of raw samples to be processed. 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.

downSampleRatio, the downsampling factor that will be applied to the raw data. Must be greater than zero.

downSampleRatioMode, which downsampling mode to use. The available values are:
  PICO_RATIO_MODE_AGGREGATE
  PICO_RATIO_MODE_DECIMATE
  PICO_RATIO_MODE_AVERAGE
  PICO_RATIO_MODE_TRIGGER - cannot be combined with any other ratio mode
  PICO_RATIO_MODE_RAW

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

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

Returns

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

3.18.1   Downsampling modes

Various methods of data reduction, or **downsampling**, are possible with the PicoScope 6000E 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 `ps6000aGetValues()`. The following modes are available:

`PICO_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.
`PICO_RATIO_MODE_AVERAGE`	Reduces every block of *n* values to a single value representing the average (arithmetic mean) of all the values.
`PICO_RATIO_MODE_DECIMATE`	Reduces every block of *n* values to just the first value in the block, discarding all the other values.
`PICO_RATIO_MODE_DISTRIBUTION`	Not implemented.
`PICO_RATIO_MODE_TRIGGER`	Gets 20 samples either side of the trigger point.
`PICO_RATIO_MODE_RAW`	No downsampling. Returns raw data values.

3.19   ps6000aGetValuesAsync - read data without blocking

```
PICO_STATUS ps6000aGetValuesAsync
(
  int16_t            handle,
  uint64_t           startIndex,
  uint64_t           noOfSamples,
  uint64_t           downSampleRatio,
  PICO_RATIO_MODE    downSampleRatioMode,
  uint64_t           segmentIndex,
  PICO_POINTER       lpDataReady,
  PICO_POINTER       pParameter
)
```

This function obtains data from the oscilloscope, with downsampling if requested, starting at the specified sample number. It delivers the data using a callback.

Applicability

Streaming mode and block mode

Arguments

```
handle,
startIndex,
noOfSamples,
downSampleRatio,
downSampleRatioMode,
segmentIndex: see ps6000aGetValues()
```

`lpDataReady`,   a pointer to the user-supplied function that will be called when the data is ready. This is usually a `PicoDataReadyUsingReads()` function. For compatibility with older applications the driver also supports a `ps6000aDataReady()` function.

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

Returns

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

3.20   ps6000aGetValuesBulk - read multiple segments

```
PICO_STATUS ps6000aGetValuesBulk
(
  int16_t           handle,
  uint64_t          startIndex,
  uint64_t        * noOfSamples,
  uint64_t          fromSegmentIndex,
  uint64_t          toSegmentIndex,
  uint64_t          downSampleRatio,
  PICO_RATIO_MODE   downSampleRatioMode,
  int16_t         * overflow
)
```

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

Applicability

Rapid block mode

Arguments

`handle, startIndex, noOfSamples, downSampleRatio, downSampleRatioMode, overflow`: see `ps6000aGetValues()`

`fromSegmentIndex, toSegmentIndex`: zero-based numbers of the first and last memory segments where the data is stored.

Returns

```
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
```

3.21   ps6000aGetValuesBulkAsync - read multiple segments without blocking

```
PICO_STATUS ps6000aGetValuesBulkAsync
(
  int16_t            handle,
  uint64_t           startIndex,
  uint64_t           noOfSamples,
  uint64_t           fromSegmentIndex,
  uint64_t           toSegmentIndex,
  uint64_t           downSampleRatio,
  PICO_RATIO_MODE    downSampleRatioMode,
  PICO_POINTER       lpDataReady,
  PICO_POINTER       pParameter
)
```

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.

Applicability

Rapid block mode

Arguments

```
handle,
startIndex,
noOfSamples,
downSampleRatio,
downSampleRatioMode: see ps6000aGetValues()

fromSegmentIndex,
toSegmentIndex: see ps6000aGetValuesBulk()

lpDataReady,
pParameter
```

Returns

```
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
```

3.22   ps6000aGetValuesOverlapped - get rapid block data

```
PICO_STATUS ps6000aGetValuesOverlapped
(
  int16_t              handle,
  uint64_t             startIndex,
  uint64_t           * noOfSamples,
  uint64_t             downSampleRatio,
  PICO_RATIO_MODE      downSampleRatioMode,
  uint64_t             fromSegmentIndex,
  uint64_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 ps6000aRunBlock(). The advantage of this method is that the driver makes contact with the scope only once, when you call ps6000aRunBlock(), compared with the two contacts that occur when you use the conventional ps6000aRunBlock(), ps6000aGetValues() calling sequence. This slightly reduces the dead time between successive captures in rapid block mode.

After calling ps6000aRunBlock(), you can optionally use ps6000aGetValues() to request further copies of the data. This might be required if you wish to display the data with different data reduction settings.

Applicability

Rapid block mode

Arguments

```
handle,
startIndex,
* noOfSamples,
downSampleRatio,
downSampleRatioMode: see ps6000aGetValues()

fromSegmentIndex,
toSegmentIndex,
* overflow, see ps6000aGetValuesBulk().
```

Returns

```
PICO_OK
PICO_INVALID_HANDLE
PICO_INVALID_PARAMETER
PICO_DRIVER_FUNCTION
```

3.22.1   Using GetValuesOverlapped()

1. Open the oscilloscope using `ps6000aOpenUnit()`.
2. Select channel ranges and AC/DC coupling using `ps6000aSetChannelOn()`.
3. Using `ps6000aGetTimebase()`, select timebases until the required nanoseconds per sample is located.
4. Use the trigger setup functions `ps6000aSetTriggerChannelConditions()`, `ps6000aSetTriggerChannelDirections()` and `ps6000aSetTriggerChannelProperties()` to set up the trigger if required.
5. Use `ps6000aSetDataBuffer()` to tell the driver where your memory buffer is.
6. Set up the transfer of the block of data from the oscilloscope using `ps6000aGetValuesOverlapped()`.
7. Start the oscilloscope running using `ps6000aRunBlock()`.
8. Wait until the oscilloscope is ready using the `ps6000aBlockReady()` callback (or poll using `ps6000aIsReady()`).
9. Display the data.
10. Repeat steps 7 to 9 if needed.
11. Stop the oscilloscope by calling `ps6000aStop()`.

A similar procedure can be used with rapid block mode.

3.23   ps6000aGetValuesTriggerTimeOffsetBulk - get trigger time offsets for multiple segments

```
PICO_STATUS ps6000aGetValuesTriggerTimeOffsetBulk
(
int16_t              handle,
int64_t            * times,
PICO_TIME_UNITS    * timeUnits,
uint64_t             fromSegmentIndex,
uint64_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 ps6000aGetTriggerTimeOffset() once for each waveform required. See ps6000aGetTriggerTimeOffset() for an explanation of trigger time offsets.

Applicability

Rapid block mode

Arguments

`handle`, the device identifier returned by `ps6000aOpenUnit()`.

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

`* timeUnits`, an array of integers. The array must be long enough to hold the number of requested times. On exit, `timeUnits[0]` will contain the time unit for `fromSegmentIndex` and the last element will contain the time unit for `toSegmentIndex`. `PICO_TIME_UNITS` values are listed under `ps6000aGetTriggerTimeOffset()`.

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

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

Returns

```
PICO_OK
PICO_INVALID_HANDLE
PICO_NULL_PARAMETER
PICO_DEVICE_SAMPLING
PICO_SEGMENT_OUT_OF_RANGE
PICO_NO_SAMPLES_AVAILABLE
PICO_DRIVER_FUNCTION
```

3.24   ps6000aIsReady - get status of block capture

```
PICO_STATUS ps6000aIsReady
(
  int16_t       handle,
  int16_t     * ready
)
```

This function may be used instead of a callback function to receive data from ps6000aRunBlock(). To use this method, pass a NULL pointer as the lpReady argument to ps6000aRunBlock(). You must then poll the driver to see if it has finished collecting the requested samples.

Applicability

Block mode

Arguments

handle,   the device identifier returned by ps6000aOpenUnit().

ready,   output: indicates the state of the collection. If zero, the device is still collecting. If non-zero, the device has finished collecting and ps6000aGetValues() can be used to retrieve the data.

Returns

3.25   ps6000aMemorySegments - set number of memory segments

```
PICO_STATUS ps6000aMemorySegments
(
  int16_t      handle
  uint64_t     nSegments,
  uint64_t  * nMaxSamples
)
```

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

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

Applicability

All modes

Arguments

`handle,` the device identifier returned by `ps6000aOpenUnit()`.

`nSegments,` the number of segments required. See data sheet for capacity of each model.

`* 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.

Returns

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

3.26   ps6000aMemorySegmentsBySamples - set size of memory segments

```
PICO_STATUS ps6000aMemorySegmentsBySamples
(
  int16_t      handle
  uint64_t     nSamples,
  uint64_t  *  nMaxSegments
)
```

This function sets the number of samples per memory segment. Like [ps6000aMemorySegments](#)() it controls the segmentation of the capture memory, but in this case you specify the number of samples rather than the number of segments.

Applicability
All modes

Arguments

handle,   the device identifier returned by [ps6000aOpenUnit()](#).

nSamples,   the number of samples required in each segment. See data sheet for capacity of each model. This is the total number over n channels, where n is the number of enabled channels or MSO ports rounded up to the next power of 2. For example, with 5 channels or ports enabled, n is 8. If $n > 1$, the number of segments available will be reduced accordingly.

* nMaxSegments,   on exit, the number of segments into which the capture memory has been divided.

Returns
PICO_OK
PICO_USER_CALLBACK
PICO_INVALID_HANDLE
PICO_TOO_MANY_SEGMENTS
PICO_MEMORY
PICO_DRIVER_FUNCTION

3.27   ps6000aNearestSampleIntervalStateless - get nearest sampling interval

```
PICO_STATUS ps6000aNearestSampleIntervalStateless
(
  int16_t                  handle,
  PICO_CHANNEL_FLAGS       enabledChannelFlags,
  double                   timeIntervalRequested,
  PICO_DEVICE_RESOLUTION   resolution,
  uint32_t               * timebase,
  double                 * timeIntervalAvailable
)
```

This function returns the nearest possible sample interval to the requested sample interval. It does not change the configuration of the oscilloscope.

Applicability
All modes

Arguments

handle, the device identifier returned by ps6000aOpenUnit().

enabledChannelFlags, see ps6000aGetMinimumTimebaseStateless().

timeIntervalRequested, the time interval, in seconds, that you would like to obtain.

resolution, the vertical resolution (number of bits) for which the oscilloscope will be configured.

* timebase, on exit, the number of the nearest available timebase.

* timeIntervalAvailable, on exit, the nearest available time interval, in seconds.

Returns
PICO_OK
PICO_INVALID_HANDLE
PICO_NO_SAMPLES_AVAILABLE
PICO_NULL_PARAMETER
PICO_INVALID_PARAMETER
PICO_SEGMENT_OUT_OF_RANGE
PICO_TOO_MANY_SAMPLES

3.28   ps6000aNoOfStreamingValues - get number of captured samples

```
PICO_STATUS ps6000aNoOfStreamingValues
(
  int16_t     handle,
  uint64_t  * noOfValues
)
```

This function returns the number of samples available after data collection in streaming mode. Call it after calling ps6000aStop().

Applicability

Streaming mode

Arguments

`handle`, the device identifier returned by `ps6000aOpenUnit()`.

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

Returns

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

3.29   ps6000aOpenUnit - open a scope device

```
PICO_STATUS ps6000aOpenUnit
(
  int16_t                * handle,
  int8_t                 * serial,
  PICO_DEVICE_RESOLUTION   resolution
)
```

This function opens a PicoScope 6000E 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.

Applicability

All modes

Arguments

`* handle`, on exit, the result of the attempt to open a scope:
- −1   : if the scope fails to open
- 0   : if no scope is found
- > 0 : a number that uniquely identifies the scope

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

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

`resolution`, the required vertical resolution (in bits).

Returns

```
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
```

3.30   ps6000aOpenUnitAsync - open unit without blocking

```
PICO_STATUS ps6000aOpenUnitAsync
(
  int16_t                * status,
  int8_t                 * serial,
  PICO_DEVICE_RESOLUTION   resolution
)
```

This function opens a scope without blocking the calling thread. You can find out when it has finished by periodically calling ps6000aOpenUnitProgress() until that function sets the complete flag to a non-zero value.

Applicability

All modes

Arguments

* status, a status code:

    0 if the open operation was disallowed because another open operation is in progress
    1 if the open operation was successfully started

* serial, see ps6000aOpenUnit().

resolution, the vertical resolution required.

Returns

PICO_OK
PICO_OPEN_OPERATION_IN_PROGRESS
PICO_OPERATION_FAILED

3.31   ps6000aOpenUnitProgress - get status of opening a unit

```
PICO_STATUS ps6000aOpenUnitProgress
(
  int16_t  * handle,
  int16_t  * progressPercent,
  int16_t  * complete
)
```

This function checks on the progress of a request made to ps6000aOpenUnitAsync() to open a scope.

Applicability

Use after ps6000aOpenUnitAsync()

Arguments

* handle, see ps6000aOpenUnit(). This handle is valid only if the function returns PICO_OK.

* progressPercent, on exit, 0 while the operation is in progress, 100 when the operation is complete.

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

Returns

PICO_OK
PICO_NULL_PARAMETER
PICO_OPERATION_FAILED

3.32   ps6000aPingUnit - check if device is still connected

```
PICO_STATUS ps6000aPingUnit
(
  int16_t    handle
)
```

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

Applicability

All modes

Arguments

handle, the device identifier returned by ps6000aOpenUnit().

Returns

PICO_OK
PICO_INVALID_HANDLE
PICO_DRIVER_FUNCTION
PICO_BUSY
PICO_NOT_RESPONDING

3.33 ps6000aQueryMaxSegmentsBySamples - get number of segments

```
PICO_STATUS ps6000aQueryMaxSegmentsBySamples
(
  int16_t                  handle,
  uint64_t                 nSamples,
  uint32_t                 nChannelEnabled,
  uint64_t               * nMaxSegments,
  PICO_DEVICE_RESOLUTION   resolution
)
```

This function returns the maximum number of memory segments available given the number of samples per segment.

Applicability

All modes

Arguments

`handle,` the device identifier returned by `ps6000aOpenUnit()`.

`nSamples,` the number of samples per segment.

`nChannelEnabled,` the number of channels enabled.

`* nMaxSegments,` on exit, the maximum number of segments that can be requested.

`resolution,` an enumerated type representing the hardware resolution.

Returns

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

3.34 ps6000aQueryOutputEdgeDetect – check if output edge detection is enabled

```
PICO_STATUS ps6000aQueryOutputEdgeDetect
(
  int16_t   handle,
  int16_t * state
)
```

This function reports whether output edge detection mode is currently enabled. The default state is enabled.

To switch output edge detection mode on or off, use ps6000aSetOutputEdgeDetect. See that function description for more details.

Applicability

All modes

Arguments

handle, the device identifier returned by ps6000aOpenUnit().

* state, on exit, the state of output edge detection:

    0 = off

    1 = on

Returns

PICO_OK or other code from PicoStatus.h

3.35   ps6000aRunBlock - start block mode capture

```
PICO_STATUS ps6000aRunBlock
(
  int16_t              handle,
  uint64_t             noOfPreTriggerSamples,
  uint64_t             noOfPostTriggerSamples,
  uint32_t             timebase,
  double            *  timeIndisposedMs,
  uint64_t             segmentIndex,
  ps6000aBlockReady    lpReady,
  PICO_POINTER         pParameter
)
```

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

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

Applicability

Block mode, rapid block mode

Arguments

`handle,`   the device identifier returned by `ps6000aOpenUnit()`.

`noOfPreTriggerSamples,`   the number of samples to return before the trigger event. If no trigger has been set, then this argument is added to `noOfPostTriggerSamples` to give the maximum number of data points (samples) to collect.

`noOfPostTriggerSamples,`   the number of samples to return after the trigger event. If no trigger event has been set, then this argument is added to `noOfPreTriggerSamples` to give the maximum number of data points to collect. If a trigger condition has been set, this specifies the number of data points to collect after a trigger has fired, and the number of samples to be collected is:

    noOfPreTriggerSamples + noOfPostTriggerSamples

`timebase,`   a number in the range 0 to $2^{32} - 1$. See the guide to calculating timebase values.

`* timeIndisposedMs,`   on exit, the time in milliseconds that the scope will spend collecting samples. This does not include any auto trigger timeout. If this pointer is null, nothing will be written here.

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

`lpReady,`   a pointer to the `ps6000aBlockReady()` callback function that the driver will call when the data has been collected. To use the `ps6000aIsReady()` polling method instead of a callback function, set this pointer to `NULL`.

`pParameter,`   a void pointer that is passed to the `ps6000aBlockReady()` callback function. The callback can use this pointer to return arbitrary data to the application.

Returns

PICO_OK
PICO_INVALID_HANDLE

PICO_USER_CALLBACK
PICO_SEGMENT_OUT_OF_RANGE
PICO_INVALID_CHANNEL
PICO_INVALID_TRIGGER_CHANNEL
PICO_INVALID_CONDITION_CHANNEL
PICO_TOO_MANY_SAMPLES
PICO_INVALID_TIMEBASE
PICO_NOT_RESPONDING
PICO_CONFIG_FAIL
PICO_INVALID_PARAMETER
PICO_NOT_RESPONDING
PICO_TRIGGER_ERROR
PICO_DRIVER_FUNCTION
PICO_EXTERNAL_FREQUENCY_INVALID
PICO_FW_FAIL
PICO_NOT_ENOUGH_SEGMENTS (in Bulk mode)
PICO_TRIGGER_AND_EXTERNAL_CLOCK_CLASH
PICO_PWQ_AND_EXTERNAL_CLOCK_CLASH
PICO_PULSE_WIDTH_QUALIFIER
PICO_SEGMENT_OUT_OF_RANGE (in Overlapped mode)
PICO_STARTINDEX_INVALID (in Overlapped mode)
PICO_INVALID_SAMPLERATIO (in Overlapped mode)
PICO_CONFIG_FAIL
PICO_SIGGEN_GATING_AUXIO_ENABLED (signal generator is set to trigger on AUX input with incompatible trigger type)

3.36   ps6000aRunStreaming - start streaming mode capture

```
PICO_STATUS ps6000aRunStreaming
(
  int16_t              handle,
  double             * sampleInterval,
  PICO_TIME_UNITS      sampleIntervalTimeUnits
  uint64_t             maxPreTriggerSamples,
  uint64_t             maxPostTriggerSamples,
  int16_t              autoStop,
  uint64_t             downSampleRatio,
  PICO_RATIO_MODE      downSampleRatioMode
)
```

This function tells the oscilloscope to start collecting data in streaming mode. The device can return either raw or downsampled data to your application while streaming is in progress. Call ps6000aGetStreamingLatestValues() 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 is the sum of `maxPreTriggerSamples` and `maxPostTriggerSamples`. If `autoStop` is false then this will become the maximum number of samples without downsampling.

When downsampled data is returned, the raw samples remain stored on the device. The maximum number of raw samples that can be retrieved after streaming has stopped is **(scope's memory size) / (resolution data size * channels),** where **channels** is the number of active channels rounded up to a power of 2.

Applicability

Streaming mode

Arguments

`handle`,   the device identifier returned by ps6000aOpenUnit().

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

`sampleIntervalTimeUnits`,   the unit of time used for `sampleInterval`. Use one of these values:
>   PICO_FS
>   PICO_PS
>   PICO_NS
>   PICO_US
>   PICO_MS
>   PICO_S

`maxPreTriggerSamples`,   the maximum number of raw samples before a trigger event for each enabled channel. If no trigger condition is set this argument is ignored.

`maxPostTriggerSamples`,   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.

`autoStop`,   a flag that specifies if the streaming should stop when all of `maxSamples` have been captured.

`downSampleRatio`, `downSampleRatioMode`: see ps6000aGetValues().

Returns

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)

3.37   ps6000aSetChannelOff - disable one channel

```
PICO_STATUS ps6000aSetChannelOff
(
  int16_t         handle,
  PICO_CHANNEL    channel
)
```

This function switches an analog input channel off. It has the opposite function to
ps6000aSetChannelOn().

Applicability

All modes

Arguments

handle,   the device identifier returned by ps6000aOpenUnit().

channel, see ps6000aSetChannelOn().

Returns

PICO_OK
PICO_USER_CALLBACK
PICO_INVALID_HANDLE
PICO_INVALID_CHANNEL
PICO_DRIVER_FUNCTION

3.38   ps6000aSetChannelOn - enable and set options for one channel

```
PICO_STATUS ps6000aSetChannelOn
(
  int16_t                   handle,
  PICO_CHANNEL              channel,
  PICO_COUPLING             coupling,
  PICO_CONNECT_PROBE_RANGE  range,
  double                    analogueOffset,
  PICO_BANDWIDTH_LIMITER    bandwidth
)
```

This function switches an analog input channel on and specifies its input coupling type, voltage range, analog offset and bandwidth limit. Some of the arguments within this function have model-specific values. Consult the relevant section below according to the model you have.

To switch off again, use ps6000aSetChannelOff().

For digital ports, see ps6000aSetDigitalPortOn().

Applicability

All modes

Arguments

handle, the device identifier returned by ps6000aOpenUnit().

channel, the channel to be configured. The values (subject to the number of channels on your oscilloscope model) are:

  PICO_CHANNEL_A, PICO_CHANNEL_B, PICO_CHANNEL_C, PICO_CHANNEL_D,
  PICO_CHANNEL_E, PICO_CHANNEL_F, PICO_CHANNEL_G, PICO_CHANNEL_H

coupling, the impedance and coupling type. The values supported are:

PICO_AC, 1 MΩ impedance, AC coupling. The channel accepts input frequencies from about 1 hertz up to its maximum -3 dB analog bandwidth.

PICO_DC, 1 MΩ impedance, DC coupling. The scope accepts all input frequencies from zero (DC) up to its maximum -3 dB analog bandwidth.

PICO_DC_50OHM, 50 Ω impedance, DC coupling. The higher-voltage input ranges may not be available in this mode - consult data sheet.

range, the input voltage range:

  PICO_10MV: ±10 mV
  PICO_20MV: ±20 mV
  PICO_50MV: ±50 mV
  PICO_100MV: ±100 mV
  PICO_200MV: ±200 mV
  PICO_500MV: ±500 mV
  PICO_1V: ±1 V
  PICO_2V: ±2 V
  PICO_5V: ±5 V
  PICO_10V: ±10 V *
  PICO_20V: ±20 V *

* not available when `coupling = PICO_DC_50R`

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

`bandwidth,` the bandwidth limiter setting:
    `PICO_BW_FULL:`   the scope's full specified bandwidth
    `PICO_BW_20MHZ:` −3 dB bandwidth limited to 20 MHz
    `PICO_BW_200MHZ:` −3 dB bandwidth limited to 200 MHz (for scopes with 750 MHz bandwidth and above)

Returns

`PICO_OK`
`PICO_USER_CALLBACK`
`PICO_INVALID_HANDLE`
`PICO_INVALID_CHANNEL`
`PICO_INVALID_VOLTAGE_RANGE`
`PICO_INVALID_COUPLING`
`PICO_COUPLING_NOT_SUPPORTED`
`PICO_INVALID_ANALOGUE_OFFSET`
`PICO_INVALID_BANDWIDTH`
`PICO_BANDWIDTH_NOT_SUPPORTED`
`PICO_DRIVER_FUNCTION`

3.39   ps6000aSetDataBuffer - provide location of data buffer

```
PICO_STATUS ps6000aSetDataBuffer
(
  int16_t              handle,
  PICO_CHANNEL         channel,
  PICO_POINTER         buffer,
  int32_t              nSamples,
  PICO_DATA_TYPE       dataType,
  uint64_t             waveform,
  PICO_RATIO_MODE      downSampleRatioMode,
  PICO_ACTION          action
)
```

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 ps6000aSetDataBuffers() instead.

The buffer persists between captures until it is replaced with another buffer or `buffer` is set to NULL. The buffer can be replaced at any time between calls to ps6000aGetValues().

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

Applicability

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

Arguments

`handle,` the device identifier returned by ps6000aOpenUnit().

`channel,` the channel you want to use with the buffer.

`buffer,` the location of the buffer.

`nSamples,` the length of the `buffer` array.

`dataType,` the data type that you wish to use for the sample values:
    PICO_INT8_T,   8-bit signed integer
    PICO_INT16_T,   16-bit signed integer
    PICO_INT32_T,   32-bit signed integer
    PICO_UINT32_T,   32-bit unsigned integer
    PICO_INT64_T,   64-bit signed integer

`waveform,` the segment index.

`downSampleRatioMode,` the downsampling mode. See ps6000aGetValues() for the available modes, but note that a single call to ps6000aSetDataBuffer() can only associate one buffer with one downsampling mode. If you intend to call ps6000aGetValues() with more than one downsampling mode activated, then you must call ps6000aSetDataBuffer() several times to associate a separate buffer with each downsampling mode.

`action,` the method to use when creating the buffer. The buffers are added to a unique list for the channel, data type and segment. Therefore you must use `PICO_CLEAR_ALL` to remove all buffers already written. `PICO_ACTION` values can be ORed together to allow clearing and adding in one call.

Returns

`PICO_OK`
`PICO_INVALID_HANDLE`
`PICO_INVALID_CHANNEL`
`PICO_RATIO_MODE_NOT_SUPPORTED`
`PICO_DRIVER_FUNCTION`
`PICO_INVALID_PARAMETER`

3.40   ps6000aSetDataBuffers - provide locations of both data buffers

```
PICO_STATUS ps6000aSetDataBuffers
(
  int16_t              handle,
  PICO_CHANNEL         channel,
  PICO_POINTER         bufferMax,
  PICO_POINTER         bufferMin,
  int32_t              nSamples,
  PICO_DATA_TYPE       dataType,
  uint64_t             waveform,
  PICO_RATIO_MODE      downSampleRatioMode,
  PICO_ACTION          action
)
```

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 ps6000aSetDataBuffer() instead.

Applicability

Block and streaming modes with aggregation.

Arguments

handle, the device identifier returned by ps6000aOpenUnit().

channel, the channel for which you want to set the buffers.

* bufferMax, a buffer to receive the maximum data values in aggregation mode, or the non-aggregated values otherwise.

* bufferMin, a buffer to receive the minimum aggregated data values. Not used in other downsampling modes.

nSamples,
dataType,
waveform, see ps6000aSetDataBuffer().

downSampleRatioMode, the downsampling mode. See ps6000aGetValues() for the available modes, but note that a single call to ps6000aSetDataBuffer() can only associate one buffer with one downsampling mode. If you intend to call ps6000aGetValues() with more than one downsampling mode activated, then you must call ps6000aSetDataBuffer() several times to associate a separate buffer with each downsampling mode.

action, see ps6000aSetDataBuffer()

Returns

```
PICO_OK
PICO_INVALID_HANDLE
PICO_INVALID_CHANNEL
PICO_RATIO_MODE_NOT_SUPPORTED
PICO_DRIVER_FUNCTION
PICO_INVALID_PARAMETER
```

3.41　ps6000aSetDeviceResolution – set the hardware resolution

```
PICO_STATUS ps6000aSetDeviceResolution
(
  int16_t                 handle,
  PICO_DEVICE_RESOLUTION  resolution
)
```

This function sets the sampling resolution of the device. At 10-bit and higher resolutions, the maximum capture buffer length is half that of 8-bit mode. When using 12-bit resolution only 2 channels can be enabled to capture data.

When you change the device resolution, the driver discards all previously captured data.

After changing the resolution and before calling ps6000aRunBlock() or ps6000aRunStreaming(), call ps6000aSetChannelOn() to set up the input channels.

Applicability

All modes.

Arguments

handle,　the device identifier returned by ps6000aOpenUnit().

resolution,　determines the resolution of the device when opened, the available values are one of the PICO_DEVICE_RESOLUTION.

Returns

PICO_INVALID_DEVICE_RESOLUTION if resolution is out of range.

3.41.1　PICO_DEVICE_RESOLUTION enumerated type

```
typedef enum enPicoDeviceResolution
{
  PICO_DR_8BIT  = 0,
  PICO_DR_12BIT = 1,
  PICO_DR_10BIT = 10,
} PICO_DEVICE_RESOLUTION;
```

These values specify the resolution of the sampling hardware in the oscilloscope. Each mode divides the input voltage range into a number of levels as listed below.

Applicability

Calls to ps6000aSetDeviceResolution() etc.

Values

PICO_DR_8BIT　　　　 – 8-bit resolution (256 levels)
PICO_DR_10BIT　　　　 – 10-bit resolution (1024 levels)
PICO_DR_12BIT　　　　 – 12-bit resolution (4096 levels)

3.42   ps6000aSetDigitalPortOff – switch off digital inputs

```
PICO_STATUS ps6000aSetDigitalPortOff
(
  int16_t                          handle,
  PICO_CHANNEL                     port
)
```

This function switches off one or more digital ports.

Applicability

Block and streaming modes with aggregation.

Arguments

handle, the device identifier returned by ps6000aOpenUnit().

port, see ps6000aSetDigitalPortOn().

Returns

3.43   ps6000aSetDigitalPortOn – set up and enable digital inputs

<u>PICO_STATUS</u> ps6000aSetDigitalPortOn
(
  int16_t   handle,
  PICO_CHANNEL   port,
  int16_t   * logicThresholdLevel,
  int16_t   logicThresholdLevelLength,
  PICO_DIGITAL_PORT_HYSTERESIS   hysteresis
)

This function switches on one or more digital ports and sets the logic thresholds.

Refer to the data sheet for the fastest sampling rates available with different combinations of analog and digital inputs. In most cases the fastest rates will be obtained by disabling all analog channels. When all analog channels are disabled you must also select 8-bit resolution to allow the digital inputs to operate alone.

Applicability

<u>Block</u> and <u>streaming</u> modes with <u>aggregation</u>.

Arguments

`handle`,   the device identifier returned by <u>ps6000aOpenUnit()</u>.

`port`,   identifies the MSO port:
    `PICO_DIGITAL_PORT0 = 128` (**Digital 1** port: digital channels 1D0−1D7)
    `PICO_DIGITAL_PORT1 = 129` (**Digital 2** port: digital channels 2D0−2D7)

`* logicThresholdLevel`,   on entry, a list of threshold voltages, one for each port pin, used to distinguish the 0 and 1 states. Range: −32 767 (−5 V) to 32 767 (+5 V).

`logicThresholdLevelLength`,   the number of items in the `logicThresholdLevel` list.

`hysteresis`, the hysteresis to apply to all channels in the port:
  `PICO_VERY_HIGH_400MV`
  `PICO_HIGH_200MV`
  `PICO_NORMAL_100MV`
  `PICO_LOW_50MV`

Returns

3.44   ps6000aSetNoOfCaptures - modify rapid block mode

PICO_STATUS ps6000aSetNoOfCaptures
(
  int16_t   handle,
  uint64_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.

Applicability

Rapid block mode

Arguments

handle,   the device identifier returned by ps6000aOpenUnit().

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

Returns

PICO_OK
PICO_INVALID_HANDLE
PICO_INVALID_PARAMETER
PICO_DRIVER_FUNCTION

3.45   ps6000aSetOutputEdgeDetect – change triggering behavior

```
PICO_STATUS ps6000aSetOutputEdgeDetect
(
  int16_t    handle,
  int16_t    state
)
```

This function enables or disables output edge detection mode for the logic trigger. Output edge detection is enabled by default and should be left enabled for normal operation.

The oscilloscope normally triggers only when the output of the trigger logic function changes state. For example, if the function is "A high AND B high", the oscilloscope triggers when A is high and B changes from low to high, but does not repeatedly trigger when A and B remain high. Calling ps6000aSetOutputEdgeDetect() with state = 0 changes this behavior so that the oscilloscope triggers continually while the logic trigger function evaluates to TRUE.

To find out whether output edge detection is enabled, use ps6000aQueryOutputEdgeDetect().

Applicability

Rapid block mode

Arguments

handle, the device identifier returned by ps6000aOpenUnit().

state, the desired state of output edge detection:
  0 = off
  1 = on

Returns

PICO_OK or other code from PicoStatus.h

3.46   ps6000aSetPulseWidthQualifierConditions - specify how to combine channels

```
PICO_STATUS ps6000aSetPulseWidthQualifierConditions
(
  int16_t                  handle,
  PICO_CONDITION         * conditions,
  int16_t                  nConditions,
  PICO_ACTION              action
)
```

This function is used to set conditions for the pulse width qualifier, which is an optional input to the triggering condition.

Applicability
All modes

Arguments

`handle,` the device identifier returned by `ps6000aOpenUnit()`.

`* conditions,` on entry, an array of structures specifiying the pulse width qualifier conditions. See `PICO_CONDITION`.

`nConditions,` the number of structures in the `conditions` array.

`action,` how to combine the array of conditions with existing pulse width qualifier conditions. See `ps6000aSetTriggerChannelConditions()` for the list of actions.

Returns
PICO_OK

3.47 ps6000aSetPulseWidthQualifierDirections - specify threshold directions

```
PICO_STATUS ps6000aSetPulseWidthQualifierDirections
(
    int16_t           handle,
    PICO_DIRECTION *  directions,
    int16_t           nDirections
)
```

This function is used to set directions for the pulse width qualifier, which is an optional input to the triggering condition.

Applicability
All modes

Arguments

handle, the device identifier returned by ps6000aOpenUnit().

* directions, an array of structures specifying the pulse width qualifier directions. See PICO_DIRECTION.

nDirections, the number of structures in the directions array.

Returns
PICO_OK

3.48   ps6000aSetPulseWidthQualifierProperties - specify threshold logic

```
PICO_STATUS ps6000aSetPulseWidthQualifierProperties
(
    int16_t                handle,
    uint32_t               lower,
    uint32_t               upper,
    PICO_PULSE_WIDTH_TYPE  type
)
```

This function is used to set parameters for the pulse width qualifier, which is an optional input to the triggering condition.

Applicability

All modes

Arguments

`handle`,   the device identifier returned by `ps6000aOpenUnit()`.

`lower`, the lower pulse width threshold.

`upper`, the upper pulse width threshold.

`type`, the pulse width qualifier type:
   `PICO_PW_TYPE_NONE = 0`, no pulse width qualifier required
   `PICO_PW_TYPE_LESS_THAN = 1`, pulse width must be less than threshold
   `PICO_PW_TYPE_GREATER_THAN = 2`, pulse width must be greater than threshold
   `PICO_PW_TYPE_IN_RANGE = 3`, pulse width must be between two thresholds
   `PICO_PW_TYPE_OUT_OF_RANGE = 4`, pulse width must not be between two thresholds

Returns

`PICO_OK`

3.49 ps6000aSetSimpleTrigger - set up triggering

```
PICO_STATUS ps6000aSetSimpleTrigger
(
   int16_t                    handle,
   int16_t                    enable,
   PICO_CHANNEL               source,
   int16_t                    threshold,
   PICO_THRESHOLD_DIRECTION   direction,
   uint64_t                   delay,
   uint32_t                   autoTriggerMicroSeconds
)
```

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.

Applicability

All modes

Arguments

`handle`, the device identifier returned by `ps6000aOpenUnit()`.

`enable`: disable (0) or enable (1) the trigger.

`source`: the channel on which to trigger. This can be any of the input channels listed under `ps6000aSetChannelOn()`.

`threshold`: the ADC count at which the trigger will fire.

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

`delay`: the time between the trigger occurring and the first sample being taken.

`autoTriggerMicroSeconds`: the number of microseconds the device will wait if no trigger occurs.

Returns

PICO_OK
PICO_INVALID_HANDLE
PICO_USER_CALLBACK
PICO_DRIVER_FUNCTION

3.50 ps6000aSetTriggerChannelConditions - set triggering logic

```
PICO_STATUS ps6000aSetTriggerChannelConditions
(
  int16_t              handle,
  PICO_CONDITION   * conditions,
  int16_t              nConditions,
  PICO_ACTION          action
)
```

This function sets up trigger conditions on the scope's inputs. The trigger is defined by one or more PICO_CONDITION 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 ps6000aSetSimpleTrigger().

Applicability

All modes

Arguments

`handle`, the device identifier returned by `ps6000aOpenUnit()`.

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

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

`action`, specifies how to apply the PICO_CONDITION array to any existing trigger conditions:
```
  PICO_CLEAR_ALL = 0x00000001
  PICO_ADD = 0x00000002
```

Returns

```
PICO_OK
PICO_INVALID_HANDLE
PICO_USER_CALLBACK
PICO_CONDITIONS
PICO_MEMORY_FAIL
PICO_DRIVER_FUNCTION
```

3.50.1   PICO_CONDITION structure

A structure of this type is passed to <u>ps6000aSetTriggerChannelConditions()</u> in the `conditions` argument to specify the trigger conditions, and is defined as follows:

```
typedef struct tPicoCondition
{
  PICO_CHANNEL       source;
  PICO_TRIGGER_STATE condition;
} PICO_CONDITION
```

Each structure is the logical AND of the states of the scope's inputs. The <u>ps6000aSetTriggerChannelConditions()</u> function can OR together a number of these structures to produce the final trigger condition, which can be any possible Boolean function of the scope's inputs.

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

Elements

`source`,   the signal that forms an input to the trigger condition:
　　`PICO_CHANNEL_A`, `PICO_CHANNEL_B`, `PICO_CHANNEL_C`, `PICO_CHANNEL_D`, `PICO_CHANNEL_E`, `PICO_CHANNEL_F`, `PICO_CHANNEL_G`, `PICO_CHANNEL_H`, one of the analog input channels
　　`PICO_PORT0`,   MSO port **Digital 1** (channels 1D0−1D7)
　　`PICO_PORT1`,   MSO port **Digital 2** (channels 2D0−2D7)
　　`PICO_TRIGGER_AUX`,   the **AUX** input
　　`PICO_PULSE_WIDTH_SOURCE`,   the output of the pulse width qualifier

`condition`,   the type of condition that should be applied to each channel. Use these constants:
　　<u>PICO_CONDITION_DONT_CARE</u>
　　<u>PICO_CONDITION_TRUE</u>
　　<u>PICO_CONDITION_FALSE</u>

The channels that are set to <u>PICO_CONDITION_TRUE</u> or <u>PICO_CONDITION_FALSE</u> must all meet their conditions simultaneously to produce a trigger. Channels set to <u>PICO_CONDITION_DONT_CARE</u> are ignored.

3.51   ps6000aSetTriggerChannelDirections - set trigger directions

```
PICO_STATUS ps6000aSetTriggerChannelDirections
(
  int16_t           handle,
  PICO_DIRECTION  * directions,
  int16_t           nDirections
)
```

This function sets the direction of the trigger for one or more channels.

Applicability

All modes

Arguments

`handle,` the device identifier returned by `ps6000aOpenUnit()`.

`* directions`, an array of structures specifying the trigger direction for each channel. See `PICO_DIRECTION`.

`nDirections`, the number of structures in the `directions` array.

Returns

`PICO_OK`
`PICO_INVALID_HANDLE`
`PICO_USER_CALLBACK`
`PICO_INVALID_PARAMETER`

3.51.1   PICO_DIRECTION structure

A structure of this type is passed to <u>ps6000aSetTriggerChannelDirections()</u> in the `directions` argument to specify the trigger directions, and is defined as follows:

```
typedef struct tPicoDirection
{
  PICO_CHANNEL            channel;
  PICO_THRESHOLD_DIRECTION direction;
  PICO_THRESHOLD_MODE      thresholdMode;
} PICO_DIRECTION
```

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

Elements

`channel`, the channel whose direction you want to set.

`direction`, the direction required for the channel.

`thresholdMode`, the type of threshold to use.

`PICO_THRESHOLD_DIRECTION` values:

Constant	Trigger type	Threshold	Polarity
`PICO_ABOVE = 0`	Gated	Upper	Above
`PICO_ABOVE_LOWER = 5`	Gated	Lower	Above
`PICO_BELOW = 1`	Gated	Upper	Below
`PICO_BELOW_LOWER = 6`	Gated	Lower	Below
`PICO_RISING = 2`	Threshold	Upper	Rising
`PICO_RISING_LOWER = 7`	Threshold	Lower	Rising
`PICO_FALLING = 3`	Threshold	Upper	Falling
`PICO_FALLING_LOWER = 8`	Threshold	Lower	Falling
`PICO_RISING_OR_FALLING = 4`	Threshold	Lower (for rising edge) Upper (for falling edge)	
`PICO_INSIDE = 0`	Window-qualified	Both	Inside
`PICO_OUTSIDE = 1`	Window-qualified	Both	Outside
`PICO_ENTER = 2`	Window	Both	Entering
`PICO_EXIT = 3`	Window	Both	Leaving
`PICO_ENTER_OR_EXIT = 4`	Window	Both	Either entering or leaving
`PICO_POSITIVE_RUNT = 9`	Window-qualified	Both	Entering from below
`PICO_NEGATIVE_RUNT`	Window-qualified	Both	Entering from above
`PICO_LOGIC_LOWER = 1000`	Logic	Lower	
`PICO_LOGIC_UPPER = 1001`	Logic	Upper	
`PICO_NONE = 2`	None	None	None

`PICO_THRESHOLD_MODE` values:

Constant	Mode
`PICO_LEVEL = 0`	Active when input is above or below a single threshold
`PICO_WINDOW = 1`	Active when input is between two thresholds

3.52 ps6000aSetTriggerChannelProperties - set up triggering

```
PICO_STATUS ps6000aSetTriggerChannelProperties
(
  int16_t                              handle,
  PICO_TRIGGER_CHANNEL_PROPERTIES   * channelProperties
  int16_t                              nChannelProperties
  int16_t                              auxOutputEnable,
  uint32_t                             autoTriggerMicroSeconds
)
```

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

Applicability

All modes

Arguments

handle, the device identifier returned by ps6000aOpenUnit().

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

nChannelProperties, the size of the channelProperties array. If zero, triggering is switched off.

auxOutputEnable: not used

autoTriggerMicroSeconds, the time in microseconds for which the scope device will wait before collecting data if no trigger event occurs. If this is set to zero, the scope device will wait indefinitely for a trigger.

Returns

PICO_OK
PICO_INVALID_HANDLE
PICO_USER_CALLBACK
PICO_TRIGGER_ERROR
PICO_MEMORY_FAIL
PICO_INVALID_TRIGGER_PROPERTY
PICO_DRIVER_FUNCTION
PICO_INVALID_PARAMETER

3.52.1   TRIGGER_CHANNEL_PROPERTIES structure

A structure of this type is passed to `ps6000aSetTriggerChannelProperties()` in the `channelProperties` argument to specify the trigger mechanism, and is defined as follows:

```
typedef struct tTriggerChannelProperties
{
    int16_t        thresholdUpper;
    uint16_t       thresholdUpperHysteresis;
    int16_t        thresholdLower;
    uint16_t       thresholdLowerHysteresis;
    PICO_CHANNEL   channel;
} PICO_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 `ps6000aSetTriggerChannelDirections()`. Each trigger threshold has its own hysteresis setting.

Elements

`thresholdUpper,` the upper threshold at which the trigger fires. It is scaled in 16-bit ADC counts at the currently selected range for that channel. Use when "Upper" or "Both" is specified in `ps6000aSetTriggerChannelDirections()`.

`hysteresisUpper,` 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.

`thresholdLower,` lower threshold (see `thresholdUpper`). Use when "Lower" or "Both" is specified in `ps6000aSetTriggerChannelDirections()`.

`hysteresisLower,` lower threshold hysteresis (see `hysteresisUpper`).

`channel,` the channel to which the properties apply. This can be one of the input channels listed under `ps6000aSetChannelOn()`.

3.53   ps6000aSetTriggerDelay - set post-trigger delay

```
PICO_STATUS ps6000aSetTriggerDelay
(
  int16_t      handle,
  uint64_t     delay
)
```

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

Applicability

[Block](#) and [rapid block](#) modes

Arguments

`handle`, the device identifier returned by [`ps6000aOpenUnit()`](#).

`delay`, the time between the trigger occurring and the first sample. For example, if `delay=100`, the scope would wait 100 sample periods before sampling. At a [timebase](#) of 5 GS/s, or 200 ps per sample (`timebase=0`), the total delay would then be 100 x 200 ps = 20 ns.
Range: 0 to [`MAX_DELAY_COUNT`](#).

Returns

```
PICO_OK
PICO_INVALID_HANDLE
PICO_USER_CALLBACK
PICO_DRIVER_FUNCTION
```

3.54   ps6000aSetTriggerDigitalPortProperties - set port directions

```
PICO_STATUS ps6000aSetTriggerDigitalPortProperties
(
  int16_t                          handle,
  PICO_CHANNEL                     port,
  PICO_DIGITAL_CHANNEL_DIRECTIONS * directions,
  int16_t                          nDirections
)
```

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

Applicability

All modes

Arguments

`handle,` the device identifier returned by `ps6000aOpenUnit()`.

`port,` identifies the digital port on the oscilloscope:
  `PICO_PORT0`: **Digital 1** port (channels 1D0–1D7)
  `PICO_PORT1`: **Digital 2** port (channels 2D0–2D7)

`* directions,` an array of structures specifying the channel directions.

`nDirections,` the number of items in the `directions` array.

Returns

`PICO_OK`

3.54.1   PICO_DIGITAL_CHANNEL_DIRECTIONS structure

A list of structures of this type is passed to `ps6000aSetTriggerDigitalPortProperties()` in the `directions` argument to specify the digital channel trigger directions, and is defined as follows:

```
typedef struct tDigitalChannelDirections
{
   PICO_PORT_DIGITAL_CHANNEL          channel;
   PICO_DIGITAL_DIRECTION             direction;
} PICO_DIGITAL_CHANNEL_DIRECTIONS
```

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

Elements

`channel`, identifies the digital channel from `PICO_PORT_DIGITAL_CHANNEL0` up to `PICO_PORT_DIGITAL_CHANNEL7`.

`direction`, the trigger direction from the following list:

`PICO_DIGITAL_DONT_CARE:`	channel has no effect on trigger
`PICO_DIGITAL_DIRECTION_LOW:`	channel must be low to trigger
`PICO_DIGITAL_DIRECTION_HIGH:`	channel must be high to trigger
`PICO_DIGITAL_DIRECTION_RISING:`	channel must transition from low to high to trigger
`PICO_DIGITAL_DIRECTION_FALLING:`	channel must transition from high to low to trigger
`PICO_DIGITAL_DIRECTION_RISING_OR_FALLING:`	any transition on channel causes a trigger

3.55   ps6000aSigGenApply - set output parameters

```
PICO_STATUS ps6000aSigGenApply
(
  int16_t             handle,
  int16_t             sigGenEnabled,
  int16_t             sweepEnabled,
  int16_t             triggerEnabled,
  int16_t             automaticClockOptimisationEnabled,
  int16_t             overrideAutomaticClockAndPrescale,
  double           * frequency,
  double           * stopFrequency,
  double           * frequencyIncrement,
  double           * dwellTime
)
```

This function controls a number of parameters for the signal generator.

Applicability

All modes

Arguments

`handle`,   the device identifier returned by `ps6000aOpenUnit()`.

`sigGenEnabled`, switches the signal generator on (1) or off (0).

`sweepEnabled,` switches sweep mode on (1) or off (0).

`triggerEnabled`, switches triggering on (1) or off (0).

`automaticClockOptimisationEnabled`, switches clock optimization on (1) or off (0).

`overrideAutomaticClockAndPrescale`, switches automatic clock and prescale override on or off:
  0 = override off: ignore parameters set by `ps6000aSigGenClockManual`() and use the specified start and stop frequencies.
  1 = override on: use parameters set by `ps6000aSigGenClockManual`() and ignore the specified start and stop frequencies.

`* frequency`, the signal generator frequency (or start frequency in sweep mode).

`* stopFrequency`, the signal generator frequency at the end of the sweep.

`* frequencyIncrement`, the frequency step size in sweep mode.

`* dwellTime`, the time in seconds between frequency steps in sweep mode.

Returns

```
PICO_OK
PICO_INVALID_HANDLE
PICO_DRIVER_FUNCTION
PICO_NOT_RESPONDING
```

3.56   ps6000aSigGenClockManual - control signal generator clock

```
PICO_STATUS ps6000aSigGenClockManual
(
  int16_t                 handle,
  double                  dacClockFrequency,
  uint64_t                prescaleRatio
)
```

This function allows direct control of the signal generator clock.

Applicability

All modes

Arguments

`handle,` the device identifier returned by `ps6000aOpenUnit()`.

`dacClockFrequency,` the clock frequency of the DAC (digital-to-analog converter).

`prescaleRatio,` the ratio to program into the prescaler. The prescaler allows the precise generation of low frequencies:

sample frequency = `dacClockFrequency / prescaleRatio`

Returns

PICO_OK
PICO_INVALID_HANDLE
PICO_DRIVER_FUNCTION
PICO_NOT_RESPONDING

3.57   ps6000aSigGenFilter - switch output filter on or off

```
PICO_STATUS ps6000aSigGenFilter
(
  int16_t                 handle,
  PICO_SIGGEN_FILTER_STATE   filterState
)
```

This function controls the filter on the output of the signal generator. The filter can be used to remove unwanted high-frequency synthesizer noise.

Applicability

All modes

Arguments

`handle`, the device identifier returned by `ps6000aOpenUnit()`.

`filterState`, can be set on or off, or put in automatic mode.

Returns

```
PICO_OK
PICO_INVALID_HANDLE
PICO_DRIVER_FUNCTION
PICO_NOT_RESPONDING
```

3.58   ps6000aSigGenFrequency - set output frequency

<u>PICO_STATUS</u> ps6000aSigGenFrequency
(
  int16_t   handle,
  double   frequencyHz
)

This function sets the frequency of the signal generator.

Applicability
All modes

Arguments

handle,   the device identifier returned by <u>ps6000aOpenUnit()</u>.

frequencyHz, the desired frequency in hertz.

Returns

PICO_OK or a code from PicoStatus.h

3.59   ps6000aSigGenFrequencyLimits - get limits in sweep mode

```
PICO_STATUS ps6000aSigGenFrequencyLimits
(
  int16_t                handle,
  PICO_WAVE_TYPE         waveType,
  uint64_t             * numSamples,
  double               * startFrequency,
  int16_t                sweepEnabled,
  double               * manualDacClockFrequency,
  uint64_t             * manualPrescaleRatio,
  double               * maxStopFrequencyOut,
  double               * minFrequencyStepOut,
  double               * maxFrequencyStepOut,
  double               * minDwellTimeOut,
  double               * maxDwellTimeOut
)
```

This function queries the maximum and minimum values for the signal generator in frequency sweep mode.

Applicability

All models

Arguments

handle, the device identifier returned by ps6000aOpenUnit().

waveType, the waveform that you intend to use.

* numSamples, for arbitrary waveforms only, the number of samples in the AWG buffer.

* startFrequency, for fixed-frequency mode, the desired frequency; for frequency sweep mode, the desired start frequency.

sweepEnabled, whether sweep mode is required (1) or not required (0).

* manualDacClockFrequency, on exit, the clock frequency to be written to the DAC using ps6000aSigGenClockManual().

* manualPrescaleRatio, on exit, the prescale ratio to be written to the DAC using ps6000aSigGenClockManual().

* maxStopFrequencyOut, on exit, the highest possible stop frequency for frequency sweep mode.

* minFrequencyStepOut, on exit, the smallest possible frequency step for frequency sweep mode.

* maxFrequencyStepOut, on exit, the largest possible frequency step for frequency sweep mode.

* minDwellTimeOut, on exit, the smallest possible dwell time for frequency sweep mode.

* maxDwellTimeOut, on exit, the largest possible dwell time for frequency sweep mode.

Returns

PICO_OK

3.60   ps6000aSigGenFrequencySweep - set signal generator to frequency sweep mode

```
PICO_STATUS ps6000aSigGenFrequencySweep
(
  int16_t                   handle,
  double                    stopFrequencyHz,
  double                    frequencyIncrement,
  double                    dwellTimeSeconds,
  PICO_SWEEP_TYPE           sweepType
)
```

This function sets frequency sweep parameters for the signal generator. It assumes that you have previously called ps6000aSigGenFrequency() to set the start frequency.

Applicability

Signal generator.

Arguments

handle, the device identifier returned by ps6000aOpenUnit().

stopFrequencyHz, the frequency in hertz at which the sweep should stop.

frequencyIncrement, the amount by which the frequency should change, in hertz, at each step of the sweep.

dwellTimeSeconds, the time for which the generator should wait between frequency steps.

sweepType, the direction of the sweep, from the following list:
  PICO_UP = 0, to sweep from startFrequency up to stopFrequency and then repeat.
  PICO_DOWN = 1, to sweep from startFrequency down to stopFrequency and then repeat.
  PICO_UPDOWN = 2, to sweep from startFrequency up to stopFrequency, then down to startFrequency, and then repeat.
  PICO_DOWNUP = 3, to sweep from startFrequency down to stopFrequency, then up to startFrequency, and then repeat.

Returns

PICO_OK or a code from PicoStatus.h

3.61 ps6000aSigGenLimits - get signal generator parameters

```
PICO_STATUS ps6000aSigGenLimits
(
  int16_t              handle,
  PICO_SIGGEN_PARAMETER  parameter,
  double               * minimumPermissibleValue,
  double               * maximumPermissibleValue,
  double               * step
)
```

This function queries the maximum and minimum allowable values for a given signal generator parameter.

Applicability

All models

Arguments

`handle,` the device identifier returned by `ps6000aOpenUnit()`.

`parameter`, one of the following enumerated values:

    PICO_SIGGEN_PARAM_OUTPUT_VOLTS   = 0, the signal generator output voltage
    PICO_SIGGEN_PARAM_SAMPLE   = 1, the value of a sample in the arbitrary waveform buffer
    PICO_SIGGEN_PARAM_BUFFER_LENGTH = 2, the length of the arbitrary waveform buffer ,in samples

`* minimumPermissibleValue`, on exit, the minimum value

`* maximumPermissibleValue`, on exit, the maximum value

`* step`, on exit, the smallest increment in the parameter that will cause a change in the signal generator output.

Returns

PICO_OK

3.62   ps6000aSigGenPause - stop the signal generator

```
PICO_STATUS ps6000aSigGenPause
(
  int16_t                 handle
)
```

This function stops the signal generator. The output will remain at a constant voltage until the generator is restarted with ps6000aSigGenRestart().

Applicability

All modes

Arguments

`handle`, the device identifier returned by ps6000aOpenUnit().

Returns

`PICO_OK` or a code from `PicoStatus.h`

3.63   ps6000aSigGenPhase - set signal generator phase

```
PICO_STATUS ps6000aSigGenPhase
(
  int16_t                        handle,
  uint64_t                       deltaPhase
)
```

This function sets the phase of the signal generator. Phase is an angle relative to the moment when the signal generator was last switched on.

Applicability

All modes

Arguments

`handle,` the device identifier returned by `ps6000aOpenUnit()`.

`deltaPhase,` the desired phase in degrees.

Returns

`PICO_OK` or a code from `PicoStatus.h`

3.64   ps6000aSigGenPhaseSweep - set signal generator to sweep in phase

```
PICO_STATUS ps6000aSigGenPhaseSweep
(
  int16_t                  handle,
  uint64_t                 stopDeltaPhase,
  uint64_t                 deltaPhaseIncrement,
  uint64_t                 dwellCount,
  PICO_SWEEP_TYPE          sweepType
)
```

This function sets phase sweep parameters for the signal generator. It assumes that you have previously called ps6000aSigGenPhase() to set the start phase.

Applicability

All modes

Arguments

---
handle,   the device identifier returned by ps6000aOpenUnit().

stopDeltaPhase, the phase in degrees at which the sweep should stop.

deltaPhaseIncrement, the amount by which the phase should change, in degrees, at each step of the sweep.

dwellCount, the number of samples that the generator should wait between phase steps.

sweepType,   the direction of the sweep, from the following list:
    PICO_UP = 0,   to sweep from deltaPhase up to stopDeltaPhase and then repeat.
    PICO_DOWN = 1,   to sweep from deltaPhase down to stopDeltaPhase and then repeat.
    PICO_UPDOWN = 2,   to sweep from deltaPhase up to stopDeltaPhase, then down to deltaPhase, and then repeat.
    PICO_DOWNUP = 3,   to sweep from deltaPhase down to stopDeltaPhase, then up to deltaPhase, and then repeat.

Returns

PICO_OK or a code from PicoStatus.h

3.65   ps6000aSigGenRange - set signal generator output voltages

```
PICO_STATUS ps6000aSigGenRange
(
  int16_t   handle,
  double    peakToPeakVolts,
  double    offsetVolts
)
```

This function sets the amplitude (peak to peak measurement) and offset (voltage corresponding to data value of zero) of the signal generator.

Applicability
All modes

Arguments

`handle,`   the device identifier returned by `ps6000aOpenUnit()`.

`peakToPeakVolts,` the signal generator's peak-to-peak output range in volts (V).

`offsetVolts,` the signal generator's output offset in volts (V).

Returns
PICO_OK
PICO_INVALID_HANDLE
PICO_SIGGEN_TRIGGER_SOURCE
PICO_DRIVER_FUNCTION
PICO_NOT_RESPONDING

3.66 ps6000aSigGenRestart - continue after pause

```
PICO_STATUS ps6000aSigGenRestart
(
  int16_t                    handle
)
```

This function restarts the signal generator after it was paused with ps6000aSigGenPause().

Applicability
All modes

Arguments
handle, the device identifier returned by ps6000aOpenUnit().

Returns
PICO_OK or a code from PicoStatus.h

3.67 ps6000aSigGenSoftwareTriggerControl - set software triggering

```
PICO_STATUS ps6000aSigGenSoftwareTriggerControl
(
  int16_t                    handle,
  PICO_SIGGEN_TRIG_TYPE      triggerState
)
```

This function sets the trigger type (edge or level) and polarity for software triggering of the signal generator.

Applicability
All modes

Arguments
handle, the device identifier returned by ps6000aOpenUnit().

triggerState,
 PICO_SIGGEN_RISING = 0, rising edge trigger
 PICO_SIGGEN_FALLING = 1, falling edge trigger
 PICO_SIGGEN_GATE_HIGH = 2, trigger when high
 PICO_SIGGEN_GATE_LOW = 3, trigger when low

Returns
PICO_OK
PICO_INVALID_HANDLE
PICO_SIGGEN_TRIGGER_SOURCE
PICO_DRIVER_FUNCTION
PICO_NOT_RESPONDING

3.68   ps6000aSigGenTrigger - choose the trigger event

```
PICO_STATUS ps6000aSigGenTrigger
(
  int16_t                  handle,
  PICO_SIGGEN_TRIG_TYPE    triggerType,
  PICO_SIGGEN_TRIG_SOURCE  triggerSource,
  uint64_t                 cycles,
  uint64_t                 autoTriggerPicoSeconds
)
```

This function sets up triggering for the signal generator. This feature causes the signal generator to start and stop under the control of a signal or event.

Applicability

All modes

Arguments

`handle,` the device identifier returned by [ps6000aOpenUnit()](#).

`triggerType,` whether an edge trigger (starts on a specified edge) or a gated trigger (runs while trigger is in the specified state).

`triggerSource,` the signal used as a trigger.

`cycles,` the number of waveform cycles to generate after the trigger edge or after entering the active trigger state. Set to zero to make the signal generator run indefinitely.

`autoTriggerPicoSeconds,` the length of time in picoseconds (ps) to wait for a trigger before starting the signal generator. Set to zero to make the signal generator wait indefinitely for a trigger.

Returns

```
PICO_OK
PICO_INVALID_HANDLE
PICO_SIGGEN_TRIGGER_SOURCE
PICO_DRIVER_FUNCTION
PICO_NOT_RESPONDING
```

3.69   ps6000aSigGenWaveform - choose signal generator waveform

```
PICO_STATUS ps6000aSigGenWaveform
(
  int16_t           handle,
  PICO_WAVE_TYPE    waveType,
  int16_t         * buffer,
  uint64_t          bufferLength
)
```

This function specifies which waveform the signal generator will produce.

Applicability

All modes

Arguments

`handle,` the device identifier returned by `ps6000aOpenUnit()`.

`waveType,` specifies the type of waveform to generate.

`* buffer,` an array of sample values to be used by the arbitrary waveform generator (AWG). Used only when `waveType = PICO_ARBITRARY`.

`bufferLength,` the number of samples in the `buffer` array. Used only when `waveType = PICO_ARBITRARY`.

Returns

`PICO_OK`
`PICO_INVALID_HANDLE`
`PICO_SIGGEN_TRIGGER_SOURCE`
`PICO_DRIVER_FUNCTION`
`PICO_NOT_RESPONDING`

3.70   ps6000aSigGenWaveformDutyCycle - set duty cycle

```
PICO_STATUS ps6000aSigGenWaveformDutyCycle
(
  int16_t   handle,
  double    dutyCyclePercent
)
```

This function sets the duty cycle of the signal generator waveform in square wave and triangle wave modes.

The duty cycle of a pulse waveform is defined as the time spent in the high state divided by the period. Set to 50% to obtain a square wave.

Applicability

Square wave and triangle wave outputs only.

Arguments

handle, the device identifier returned by ps6000aOpenUnit().

dutyCyclePercent, the percentage duty cycle of the waveform from 0.0 to 100.0.

Returns

PICO_OK
PICO_INVALID_HANDLE
PICO_SIGGEN_TRIGGER_SOURCE
PICO_DRIVER_FUNCTION
PICO_NOT_RESPONDING

3.71   ps6000aStartFirmwareUpdate - update the device firmware

```
PICO_STATUS ps6000aStartFirmwareUpdate
(
  int16_t                 handle,
  PicoUpdateFirmwareProgress progress
)
```

This function updates the device's firmware (the embedded instructions stored in nonvolatile memory in the device). Updates may fix bugs or add new features.

Applicability
All modes

Arguments

handle, the device identifier returned by ps6000aOpenUnit().

progress, a user-supplied function that receives callbacks when the status of the update changes. See PicoUpdateFirmwareProgress().

Returns

PICO_OK
PICO_HANDLE_INVALID
PICO_USER_CALLBACK
PICO_DRIVER_FUNCTION

3.72   ps6000aStop - stop sampling

```
PICO_STATUS ps6000aStop
(
  int16_t     handle
)
```

This function stops the scope device from sampling data.

When running the device in streaming mode, always call this function after the end of a capture to ensure that the scope is ready for the next capture.

When running the device in block 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.

Applicability

All modes

Arguments

handle,   the device identifier returned by ps6000aOpenUnit().

Returns

```
PICO_OK
PICO_INVALID_HANDLE
PICO_USER_CALLBACK
PICO_DRIVER_FUNCTION
```

3.73   ps6000aTriggerWithinPreTriggerSamples - switch feature on or off

```
PICO_STATUS ps6000aTriggerWithinPreTriggerSamples
(
  int16_t                           handle,
  PICO_TRIGGER_WITHIN_PRE_TRIGGER   state
)
```

This function controls a special triggering feature.

Applicability

All modes

Arguments

`handle,` the device identifier returned by `ps6000aOpenUnit()`.

`state,` 0 to enable, 1 to disable.

Returns

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

4   Callbacks

4.1   ps6000aBlockReady - indicate when block-mode data ready

```
typedef void (CALLBACK *ps6000aBlockReady)
(
  int16_t        handle,
  PICO_STATUS    status,
  PICO_POINTER   pParameter
)
```

This [callback](#) function is part of your application. You register it with the PicoScope 6000E Series driver using [ps6000aRunBlock()](#) and the driver calls it back when block-mode data is ready. You can then download the data using the [ps6000aGetValues()](#) function.

Applicability

[Block mode](#) only

Arguments

`handle`,   the device identifier returned by [ps6000aOpenUnit()](#).

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

`pParameter`,   a pointer passed from [ps6000aRunBlock()](#). Your callback function can write to this location to send any data, such as a status flag, back to your application.

Returns

nothing

4.2   ps6000aDataReady - indicate when post-collection data ready

```
typedef void *ps6000aDataReady
(
  int16_t           handle,
  PICO_STATUS       status,
  uint64_t          noOfSamples,
  int16_t           overflow,
  PICO_POINTER      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 ps6000aGetValuesAsync() and the driver calls your function back when the data is ready.

Applicability
All modes

Arguments
handle, the device identifier returned by ps6000aOpenUnit().

status, a PICO_STATUS code returned by the driver.

noOfSamples, the number of samples collected.

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

pParameter, a void pointer passed from ps6000aGetValuesAsync(). The callback function can write to this location to send any data, such as a status flag, back to the application. The data type is defined by the application programmer.

Returns
nothing

4.3 PicoUpdateFirmwareProgress - get status of firmware update

```
typedef void (CALLBACK * PicoUpdateFirmwareProgress)
(
  int16_t        handle,
  uint16_t       progress
)
```

You should write this [callback](#) function and register it with the driver using [ps6000aStartFirmwareUpdate()](#). The driver calls it back when the firmware update status changes.

Applicability
All modes

Arguments
`handle`, the device identifier returned by [ps6000aOpenUnit()](#).

`progress`, a progress indicator.

Returns
nothing

4.4   PicoProbeInteractions() – callback for PicoConnect probe events

```
typedef void (PREF4 *PicoProbeInteractions)
(
  int16_t                        handle,
  PICO_STATUS                    status,
  PICO_USER_PROBE_INTERACTIONS   * probes,
  uint32_t                       nProbes
)
```

This callback function handles notifications of probe changes on scope devices that support Pico intelligent probes.

If you wish to use this feature, you must create this function as part of your application. You register it with the `ps6000a` driver using `ps6000aSetProbeInteractionCallback()` and the driver calls it back whenever a probe generates an error. See [Handling PicoConnect probe interactions](#) for more information on this process.

Applicability
All modes

Arguments

`handle`, the device identifier returned by [ps6000aOpenUnit()](#).

`status`, indicates success or failure. If multiple errors have occurred, the most general error is returned here. Probe-specific errors are returned in the `status` field of the relevant elements of the probes array.

`probes`, on entry, pointer to an array of [PICO_USER_PROBE_INTERACTIONS](#) structures.

`nProbes`, the number of elements in the `probes` array.

Returns
nothing

4.4.1   PICO_USER_PROBE_INTERACTIONS structure

A structure of this type is passed to the <u>PicoProbeInteractions()</u> callback function. It is defined as follows:

```
typedef struct tPicoUserProbeInteractions
{
  uint16_t                    connected;

  PICO_CHANNEL                channel;
  uint16_t                    enabled;

  PicoConnectProbe            probeName;

  uint8_t                     requiresPower_;
  uint8_t                     isPowered_;

  PICO_STATUS                 status_;

  PICO_CONNECT_PROBE_RANGE    probeOff;

  PICO_CONNECT_PROBE_RANGE    rangeFirst_;
  PICO_CONNECT_PROBE_RANGE    rangeLast_;
  PICO_CONNECT_PROBE_RANGE    rangeCurrent_;

  PICO_COUPLING               couplingFirst_;
  PICO_COUPLING               couplingLast_;
  PICO_COUPLING               couplingCurrent_;

  PICO_BANDWIDTH_LIMITER_FLAGS filterFlags_;
  PICO_BANDWIDTH_LIMITER_FLAGS filterCurrent_;

  PICO_BANDWIDTH_LIMITER      defaultFilter_;
} PICO_USER_PROBE_INTERACTIONS;
```

Elements

`connected`, indicates whether the probe is connected or not. The driver saves information on disconnected probes in case they are reconnected, in which case it reapplies the previous settings.

`channel`, the scope channel to which the probe is connected.

`enabled`, indicates whether the probe is switched on or off.

`probeName`, identifies the type of probe from the `PICO_CONNECT_PROBE` enumerated list.

`requiresPower_`, indicates whether the probe draws power from the scope.

`isPowered_`, indicates whether the probe is receiving power.

`status_`, a status code indicating success or failure. See `PicoStatus.h` for definitions.

`probeOff`, the range in use when the probe was last switched off.

`rangeFirst_`, the first applicable range in the `PICO_CONNECT_PROBE_RANGE` enumerated list.

`rangeLast_`, the last applicable range in the `PICO_CONNECT_PROBE_RANGE` enumerated list.

`rangeCurrent_`, the range currently in use.

`couplingFirst_`, the first applicable coupling type in the `PS4000A_COUPLING` list.

`couplingLast_`, the last applicable coupling type in the `PS4000A_COUPLING` list.

`couplingCurrent_`, the coupling type currently in use.

`filterFlags_`, a bit field indicating which bandwidth limiter options are available.

`filterCurrent_`, the bandwidth limiter option currently selected.

`defaultFilter_`, the default bandwidth limiter option for this type of probe.

4.5 PicoAWGOverrangeInteractions

```
typedef void (PREF5 *PicoAWGOverrangeInteractions)
(
  int16_t      handle,
  PICO_STATUS  status
)
```

This callback function handles notifications of AWG overrange events.

If you wish to use this feature, create this function as part of your application. Register it with the `ps6000a` driver using `ps6000aSetAWGOverrangeInteractionCallback()` and the driver calls it back whenever the external reference input changes its status.

Applicability

When arbitrary waveform generator is in use.

Arguments

`handle`, the device identifier returned by [ps6000aOpenUnit()](#).

`status`, a status code from `PicoStatus.h`.

Returns

nothing

5   Reference

5.1   Numeric data types

Here is a list of the numeric data types used in the ps6000a API:

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

5.2   Enumerated types and constants

The enumerated types and constants used in the PicoScope 6000E Series API driver are defined in the file ps6000aApi.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.

5.3   Driver status codes

Every function in the ps6000a 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 6000E Series.

5.4   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 6000E Series oscilloscopes is supplied in the form of 32-bit and 64-bit Windows DLLs called `ps6000a.dll.` These are used by your application to control the oscilloscope.

PicoScope 6000E Series. A range of PC Oscilloscopes from Pico Technology, all with a maximum sampling rate of 5 GS/s. Sampling resolutions range from 8 to 12 bits and capture memory sizes from 1 to 4 GS.

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 2.0. The second generation of USB (universal serial bus) 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.

Index

A

AC coupling   70
ADC limits   31
Aggregation   20, 47
Analog offset   70
Analog offset limits   32
API function calls   25
Averaging   47
AWG
    overrange   118

B

Bandwidth limiter   70
Block mode   11, 12
    asynchronous call   14
    callback   112
    polling status   54
    running   65
    using   14

C

Callback function
    AWG overrage   118
    block mode   112
    firmware update   114
    for data   113
    probe interactions   115
Channels
    disabling   69
    enabling   70
    get available combinations   26
    settings   70
Closing units   28
Constants   119
Coupling type, setting   70

D

Data acquisition   20
Data buffers
    declaring   72
    declaring, aggregation mode   74
DC coupling   70
Decimation   47
Digital channels
    data   10

PORT0, PORT1   10
    ports   10
Digital ports
    enabling   77
    setting up   76, 77
    switching off   76
Disk space   7
Distribution   47
Downsampling   46
    modes   47
Driver   7
    status codes   120

E

Enumerated types   119
Enumerating oscilloscopes   29

F

Firmware update
    get progress   114
Firmware updates
    apply   109
Function calls   25
Functions
    PicoAWGOverrangeInteractions   118
    PicoProbeInteractions   115
    PicoUpdateFirmwareProgress   114
    ps6000aBlockReady   112
    ps6000aChannelCombinationsStateless   26
    ps6000aCheckForUpdate   27
    ps6000aCloseUnit   28
    ps6000aDataReady   113
    ps6000aEnumerateUnits   29
    ps6000aFlashLed   30
    ps6000aGetAdcLimits   31
    ps6000aGetAnalogueOffsetLimits   32
    ps6000aGetDeviceResolution   33
    ps6000aGetMaximumAvailableMemory   34
    ps6000aGetMinimumTimebaseStateless   35
    ps6000aGetNoOfCaptures   36
    ps6000aGetNoOfProcessedCaptures   37
    ps6000aGetStreamingLatestValues   38
    ps6000aGetTimebase   41
    ps6000aGetTriggerInfo   42
    ps6000aGetTriggerTimeOffset   44
    ps6000aGetUnitInfo   45
    ps6000aGetValues   46
    ps6000aGetValuesAsync   48
    ps6000aGetValuesBulk   49
    ps6000aGetValuesBulkAsync   50

ps6000apg-1