PicoScope® 3000E Series

psospa API

Programmer's Guide

Contents

1   Introduction

The [PicoScope 3000E Series](#) of oscilloscopes from Pico Technology is a range of compact high-performance units designed to replace traditional benchtop oscilloscopes.

This manual explains how to use the `psospa` API (application programming interface) for the PicoScope 3000E Series oscilloscopes.

For more information on the hardware, see the *[PicoScope 3000E Series Data Sheet](#)*.

---

psospa-3 programmer's guide (Available [online](#) and as a [PDF](#))

psospa-3   6

1.1   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.

1.2   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. *Linux* is the registered trademark of Linus Torvalds, registered in the U.S. and other countries. *macOS* is a trademark of Apple Inc., registered in the U.S. 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 `psospa` library allows you to program a PicoScope 3000E 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 types.
- 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

To ensure that your PicoScope 3000E 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	Microsoft Windows 10 or 11, 64-bit only Linux: Ubuntu or openSUSE, 64-bit only macOS, 64-bit only
Processor, memory, free disk space	As required by the operating system.
Ports	USB 2.0 or 3.0 port

The software development kit or driver libraries for all supported operating systems can be found at picotech.com/downloads

The `psospa` driver offers three different methods of recording data, all of which support USB 2.0 and USB 3.0. A USB 3.0 port will offer the best performance especially in streaming mode or when retrieving large amounts of data from the oscilloscope.

2.2 Driver

Your application will communicate with a PicoScope library called `psospa`. The driver exports the 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 driver names for each supported operating system are listed in the following table:

Windows: `psospa.dll`

macOS: `libpsospa.dylib`

Linux: `libpsospa.so`

psospa-3   8

The API depends on OS-specific low-level drivers. These drivers are installed by the SDK and configured when you plug the oscilloscope into a USB port for the first time. Your application does not call these drivers directly.

If you want to deploy your application which uses the `psospa` driver on other computer systems, you'll need to include these dependencies in your package / installer or, in the case of Linux systems, list `psospa` as a dependency of your package and ensure the Pico package repository is available on the target system.

2.3   Voltage ranges

You can set a device input channel to any available voltage range with the psospaSetChannelOn() function.

The voltage range is chosen by specifying its minimum and maximum range and units, which must correspond to one of the available voltage ranges specified in the data sheet for your oscilloscope.

By default, 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 psospaGetAdcLimits(). This function replies with the following values:

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

Example at 8-bit resolution

1. Call psospaSetChannelOn() with arguments:

```
rangeMin = −1 000 000 000
rangeMax = +1 000 000 000
rangeType = PICO_PROBE_NONE_NV
```

to select the ±1 V range.

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 below:
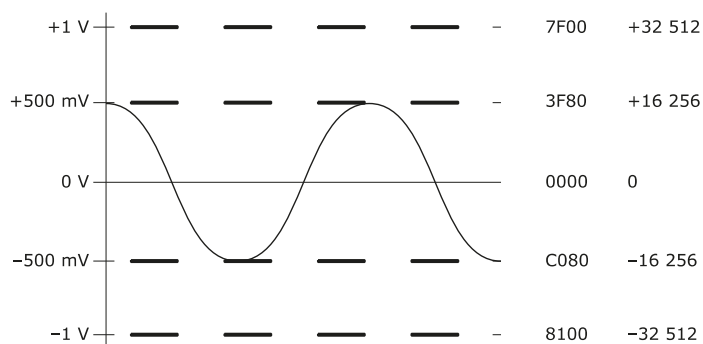
Digital inputs:

See psospaSetDigitalPortOn() and psospaSetDigitalPortOff()

2.4   MSO digital data

Applicability

A PicoScope 3000E Series MSO model has two 8-bit digital ports—**PORT0** and **PORT1**—making a total of 16 digital channels.

Use the [psospaSetDataBuffer()](#) and [psospaSetDataBuffers()](#) 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.

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

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

```
// Mask Port 1 values to get lower 8 bits
portValue = 0x00ff & sampleFromPort1Buffer;

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

// Mask Port 0 values to get lower 8 bits,
// then OR with shifted Port 1 bits to get 16-bit word
portValue |= 0x00ff & sampleFromPort0Buffer;

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

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

2.5   Triggering

PicoScope 3000E 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:

- psospaSetTriggerChannelConditions()
- psospaSetTriggerChannelDirections()
- psospaSetTriggerChannelProperties()
- psospaSetTriggerDigitalPortProperties()

These can be run collectively by calling psospaSetSimpleTrigger(), 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 triggering methods, including:

- 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:

- psospaSetPulseWidthQualifierProperties()
- psospaSetPulseWidthQualifierConditions()
- psospaSetPulseWidthQualifierDirections()
- psospaSetPulseWidthDigitalPortProperties()

Additional trigger parameters can be set using the functions:

- psospaSetTriggerDelay()
- psospaSetTriggerHoldoffCounterBySamples()

2.6   Sampling modes

PicoScope 3000E 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.

psospa-3   12

- **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 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 3000E 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 `psospaMemorySegments()`) and the sampling resolution.

- **Sampling rate.** A PicoScope 3000E 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 3000E 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 tens of 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 `psospaRunBlock()`, `psospaStop()` and `psospaGetValues()`.

- **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 `psospaMemorySegments()` or `psospaMemorySegmentsBySamples()`.

- **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 <u>block mode</u> using a single <u>memory segment</u>:

1.   Open the oscilloscope using `psospaOpenUnit()`.
2.   Select channel ranges and AC/DC/50 Ω coupling using `psospaSetChannelOn()` and `psospaSetChannelOff()`.
3.   Use `psospaNearestSampleIntervalStateless()` to get the closest valid sample interval to the requested.
4.   Use the trigger setup functions `psospaSetTriggerChannelConditions()`, `psospaSetTriggerChannelDirections()`, `psospaSetTriggerChannelProperties()` or `psospaSetSimpleTrigger()` to set up the trigger if required.
5.   Start the oscilloscope running using `psospaRunBlock()`.
6.   Wait until the oscilloscope is ready using the `psospaBlockReady()` callback (or poll using `psospaIsReady()`).
7.   Use `psospaSetDataBuffer()` 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 `psospaGetValues()`.
9.   Display or process the data.
10.   Repeat steps 5 to 9.
11.   Stop the oscilloscope using `psospaStop()`.
12.   Request new views of stored data using different downsampling parameters: see <u>Retrieving stored data</u>.
13.   Close the device using `psospaCloseUnit()`.
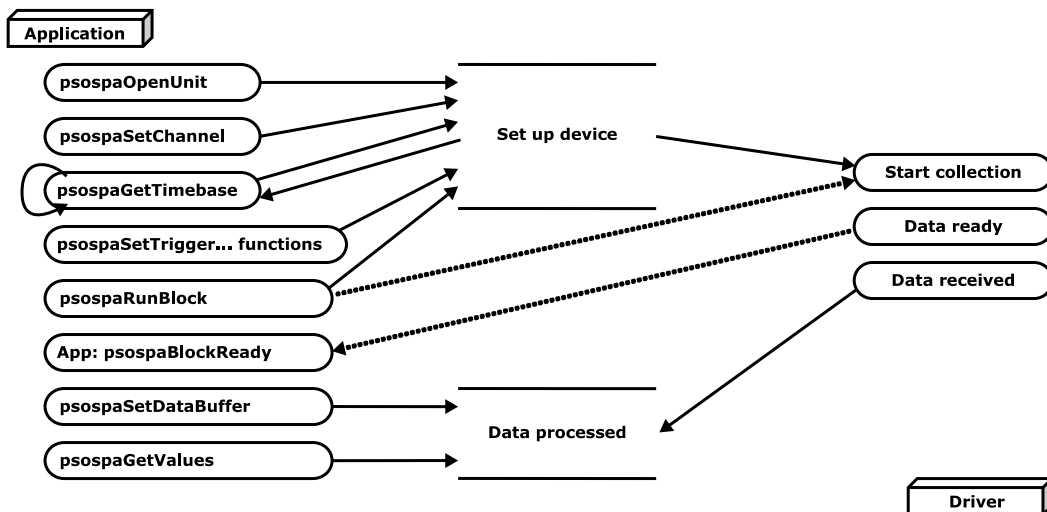
2.6.1.2   Asynchronous calls in block mode

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

2.6.2   Rapid block mode

In normal block mode, the PicoScope 3000E 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 `psospaOpenUnit()`.
2.   Select channel ranges and AC/DC coupling using `psospaSetChannelOn()` and `psospaSetChannelOff()`.
3.   Use `psospaNearestSampleIntervalStateless()`, to find a valid sampling interval to use.
4.   Use the trigger setup functions `psospaSetTriggerChannelConditions()`, `psospaSetTriggerChannelDirections()`, `psospaSetTriggerChannelProperties()` and `psospaSetSimpleTrigger()` to set up the trigger if required.
5.   Start the oscilloscope running using `psospaRunBlock()`.
6.   Wait until the oscilloscope is ready using the `psospaBlockReady()` callback.
7.   Use `psospaSetDataBuffer()` 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 4.
8.   Transfer the blocks of data from the oscilloscope using `psospaGetValuesBulk()`.
9.   Display or process the data.
10.   Repeat steps 5 to 9 if necessary.
11.   Stop the oscilloscope using `psospaStop()`.
12.   Close the device using `psospaCloseUnit()`.

With aggregation
To use rapid block mode with aggregation, follow steps 1 to 6 above and then proceed as follows:

7a.   Call `psospaSetDataBuffers()` to set up one pair of buffers for every waveform segment required.
8a.   Call `psospaGetValuesBulk()` for each pair of buffers.

Continue from step 9 above.

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 number of captures required)

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

psospaRunBlock
(
  handle,
  0,                      // noOfPreTriggerSamples
  10000,                  // noOfPostTriggerSamples
  400,                    // sample interval in picoseconds
  &timeIndisposedMs,
  0,                      // first segment index to capture
  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++)
  {
    psospaSetDataBuffer
    (
      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.

```
psospaGetValuesBulk
(
  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 [psospaRunBlock()](). 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.

psospa-3   17

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
psospaSetNoOfCaptures(handle, MAX_WAVEFORMS);


pParameter = false;
psospaRunBlock
(
  handle,
  0,                      // noOfPreTriggerSamples,
  1000000,                // noOfPostTriggerSamples,
  400,                    // sample interval in picoseconds,
  &timeIndisposedMs,
  0,                      // first segment index to be captured,
  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++)
{
  psospaSetDataBuffers
  (
    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++)
{
  psospaGetValues
```

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

Comments: each waveform is retrieved one at a time from the driver with an aggregation of 1000. Alternatively, it would be equally valid to use <u>psospaGetValuesBulk()</u> to retrieve multiple waveforms at once as shown in the previous example.

2.6.3   Streaming mode

Streaming mode can capture data without the gaps that occur between blocks when using block mode. 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 and can be read after streaming is completed.

- **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 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 <u>`psospaOpenUnit()`</u>.
2.   Select channels, ranges and AC/DC/50 Ω coupling using <u>`psospaSetChannelOn()`</u> and <u>`psospaSetChannelOff()`</u>.
3.   Use <u>`psospaNearestSampleIntervalStateless()`</u> to get the closest valid sample interval to the requested.
4.   Use the trigger setup functions <u>`psospaSetTriggerChannelConditions()`</u>, <u>`psospaSetTriggerChannelDirections()`</u>, <u>`psospaSetTriggerChannelProperties()`</u> or <u>`psospaSetSimpleTrigger()`</u> to set up the trigger if required.
5.   Call <u>`psospaSetDataBuffer()`</u> to tell the driver where your data buffer is.
6.   Set up aggregation and start the oscilloscope running using <u>`psospaRunStreaming()`</u>.
7.   Call <u>`psospaGetStreamingLatestValues()`</u> to get data. If the function runs out of buffer space, call <u>`psospaSetDataBuffer()`</u> again to provide more buffers. You can provide the same buffer repeatedly, if you have finished processing the data already in the buffer before resubmitting it for further samples.
8.   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.
9.   Call <u>`psospaStop()`,</u> even if `autoStop` is enabled.
10.   Request new views of stored data using different downsampling parameters: see <u>Retrieving stored data</u>.
11.   Close the device using <u>`psospaCloseUnit()`</u>.
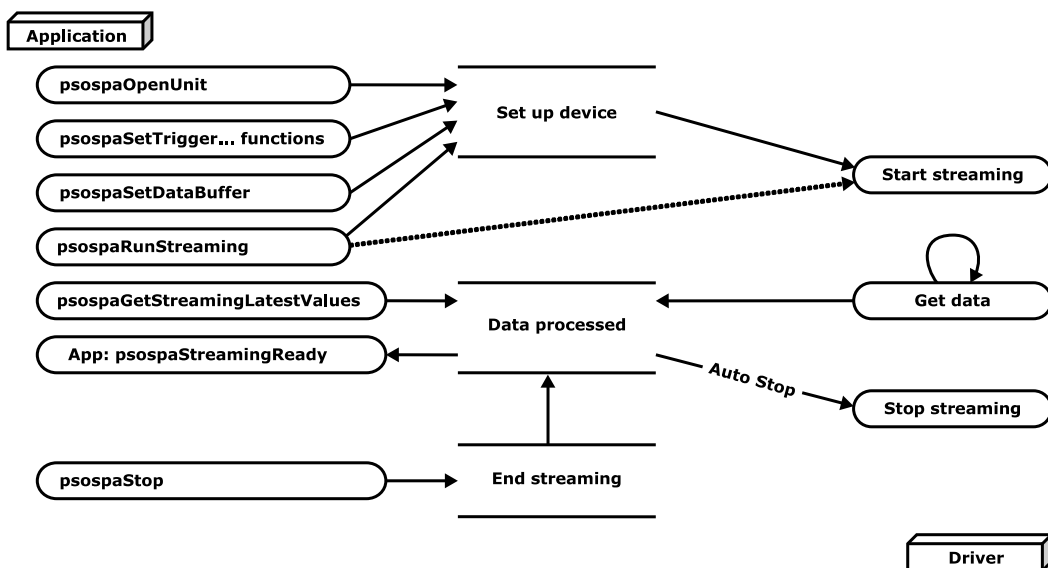
2.6.4   Retrieving stored data

You can retrieve data from the `psospa` driver with a different [downsampling](#) factor when [psospaRunBlock()](#)
or [psospaRunStreaming()](#) has already been called and has successfully captured all the data. Use
[psospaGetValuesAsync()](#).
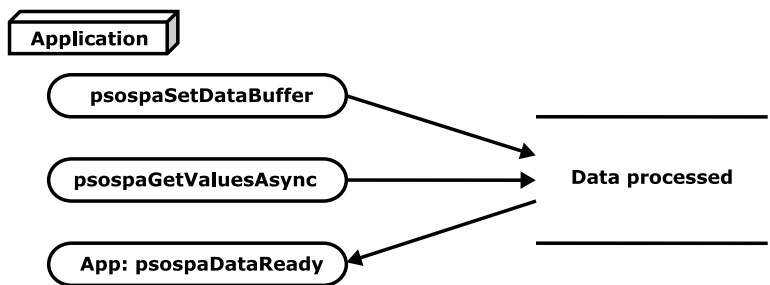
2.7   Timebases

The available sample intervals for the PicoScope 3000E oscilloscope are multiples of the minimum sampling
interval for the chosen resolution and number of enabled channels. Not all possible sample intervals are
supported, therefore you can use [psospaNearestSampleIntervalStateless](#) to determine the nearest
valid sample interval to your requirements.

In block mode, the timebase argument passed to [psospaRunBlock](#) directly represents the required sample
interval in picoseconds. In streaming mode, the sample interval is specified using the `sampleInterval` and
`sampleIntervalTimeUnits` arguments to [psospaRunStreaming](#).

Applicability	Calls to [psospaGetTimebase()](#)

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

2.8   Combining several oscilloscopes

It is possible to collect data using up to 64 PicoScope 3000E Series oscilloscopes at the same time, depending on the capabilities of the PC. Each oscilloscope must be connected to a separate USB port. The psospaOpenUnit() 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 psospaBlockReady(...)
// define callback function specific to application

handle1 = psospaOpenUnit()
handle2 = psospaOpenUnit()

psospaSetChannelOn(handle1)
// set up unit 1
psospaRunBlock(handle1)

psospaSetChannelOn(handle2)
// set up unit 2
psospaRunBlock(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: a trigger may be fed into the **Aux Trig (AUX I/O)** input to provide some degree of synchronization between multiple oscilloscopes.

3   API functions

The PicoScope 3000E Series API exports the following functions for you to use in your own applications. All functions are C functions using the standard calling convention (`__stdcall`). They are all exported with both decorated and undecorated names.

3.1　psospaCheckForUpdate - check if firmware update is available

```
PICO_STATUS psospaCheckForUpdate
(
  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. Firmware updates, when required, are distributed as part of the driver library and this function checks whether the currently-running driver contains more up-to-date firmware than that on the connected device.

Applicability

All modes

Arguments

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

`firmwareInfos,` a pointer to a buffer of `PICO_FIRMWARE_INFO` structs which, on exit, will be populated with detailed information about the available updates. Information about firmware which is already up to date will also be provided. You may pass NULL if you do not require the detailed information.

`nFirmwareInfos,` on entry, a pointer to a value which is the length of the `firmwareInfos` buffer, if `firmwareInfos` is not NULL. On exit, the number of populated entries in `firmwareInfos` (or the available number of `PICO_FIRMWARE_INFO`s if `firmwareInfos` is NULL). May be NULL if the caller does not need detailed firmware information (in which case `firmwareInfos` must also be NULL).

`updatesRequired,`　on entry, a pointer to a flag which will be set by the function to indicate if updates are required. On exit, 1 if updates are required and 0 otherwise.

Returns

`PICO_OK`

3.2   psospaCloseUnit - close a scope device

```
PICO_STATUS psospaCloseUnit
(
  int16_t    handle
)
```

This function shuts down a PicoScope 3000E Series oscilloscope. Closing the unit correctly after use returns it to a low-power state, turning off the fan and LED indicators, and leaves it in a known state ready to be re-opened when required.

Applicability
All modes

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

Returns
PICO_OK

3.3   psospaEnumerateUnits - get a list of unopened units

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

This function counts the number of PicoScope units, supported by the `psospa` driver, 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 [psospaGetUnitInfo()](#).

Applicability

All modes

Arguments

`count,` on exit, the number of PicoScope 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:

`10001/0001[3418E,01Jun24,769,2.0,1.7.16.0]`

`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_CONFIG_FAIL_AWG
PICO_INITIALISE_FPGA
```

3.4   psospaGetAdcLimits - get min and max sample values

```
PICO_STATUS psospaGetAdcLimits
(
  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. These values can be used to scale the returned sample values from the driver into voltages, using the full-scale voltage of the current input range.

Applicability

All modes

Arguments

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

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

`minValue`, the minimum sample value.

`maxValue`, the maximum sample value.

Returns

`PICO_OK`
`PICO_NULL_PARAMETER`, if both `maxValue` and `minValue` are `NULL`.

3.5   psospaGetAnalogueOffsetLimits - get analog offset information

```
PICO_STATUS psospaGetAnalogueOffsetLimits
(
  int16_t                   handle,
  int64_t                   rangeMin,
  int64_t                   rangeMax,
  PICO_PROBE_RANGE_INFO     rangeType,
  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 `psospaOpenUnit()`.

`rangeMin, rangeMax, rangeType,` the voltage range for which minimum and maximum voltages are required. See `psospaSetChannelOn()`.

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

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

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

Returns

`PICO_OK`
`PICO_INVALID_VOLTAGE_RANGE`
`PICO_NULL_PARAMETER,` if both `maximumVoltage` and `minimumVoltage` are `NULL`.
`PICO_INVALID_COUPLING`

3.6 psospaGetDeviceResolution – retrieve the device resolution

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

This function retrieves the currently selected vertical resolution of the oscilloscope.

Applicability
All modes

Arguments

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

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

Returns
`PICO_OK`
`PICO_NULL_PARAMETER`

3.7   psospaGetMaximumAvailableMemory - find max memory at a given resolution

```
PICO_STATUS psospaGetMaximumAvailableMemory
(
  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 `psospaOpenUnit()`.

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

`resolution`, the resolution as one of the `PICO_DEVICE_RESOLUTION` enum values.

Returns

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

3.8　psospaGetMinimumTimebaseStateless - find fastest available timebase

```
PICO_STATUS psospaGetMinimumTimebaseStateless
(
  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 [psospaOpenUnit()](#).

`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, in picoseconds.

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

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

Returns
`PICO_OK`
`PICO_NO_SAMPLES_AVAILABLE`
`PICO_NULL_PARAMETER`
`PICO_INVALID_PARAMETER`
`PICO_SEGMENT_OUT_OF_RANGE`
`PICO_TOO_MANY_SAMPLES`
`PICO_NO_CHANNELS_OR_PORTS_ENABLED`
`PICO_INVALID_DIGITAL_PORT`

3.9   psospaGetNoOfCaptures - query how many captures made

```
PICO_STATUS psospaGetNoOfCaptures
(
  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 psospaStop().

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

Applicability
All modes

Arguments
`handle`, the device identifier returned by psospaOpenUnit().

`nCaptures`, on output, the number of available captures that have been collected from calling psospaRunBlock().

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

3.10   psospaGetNoOfProcessedCaptures - query how many captures processed

```
PICO_STATUS psospaGetNoOfProcessedCaptures
(
  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 psospaRunBlock(). It is for use in rapid block mode, alongside the psospaGetValuesOverlapped() function, when the driver is set to transfer data from the device automatically as soon as the psospaRunBlock() function is called. You can call psospaGetNoOfProcessedCaptures() during device capture, after collection has completed or after interrupting waveform collection by calling psospaStop().

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

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

Applicability
Rapid block mode

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

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

Returns
PICO_OK
PICO_INVALID_PARAMETER

3.11 psospaGetScalingValues - get scaling factors used to adjust the gain and offset of oscilloscope data

```
PICO_STATUS psospaGetScalingValues
(
  int16_t                                      handle,
  PICO_SCALING_FACTORS_FOR_RANGE_TYPES_VALUES * scalingValues,
  int16_t                                      nChannels
)
```

This function is included for compatibility with other PicoScope series and will always return zero offset and unity scaling factor.

Applicability

All modes

Arguments

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

`scalingValues`, an array of `PICO_SCALING_FACTORS_FOR_RANGE_TYPES_VALUES` indicating the channels and ranges for which to retrieve scaling factors.

`nChannels`, the number of elements in the `scalingValues` array.

Returns

```
PICO_OK
PICO_NULL_PARAMETER
PICO_ARGUMENT_OUT_OF_RANGE
```

3.11.1 PICO_SCALING_FACTORS_FOR_RANGE_TYPES_VALUES structure

```
typedef struct tPicoScalingFactorsForRangeTypes
{
  PICO_CHANNEL            channel;
  int64_t                rangeMin;
  int64_t                rangeMax;
  PICO_PROBE_RANGE_INFO  rangeType;
  int16_t                offset;
  double                 scalingFactor
}
PICO_SCALING_FACTORS_FOR_RANGE_TYPES_VALUES;
```

This structure contains information needed to convert the source data to a measurement in volts.

Applicability

Calls to `psospaGetScalingValues()`.

Elements

`source`, the channel or port to which the information applies.

rangeMin, rangeMax, rangeType, the voltage range for which minimum and maximum voltages are required. See [psospaSetChannelOn()](#).

offset, the offset applied to the specified channel or port.

scalingFactor, the number that the specified channel or port has been multiplied by.

3.12   psospaGetStreamingLatestValues - read streaming data

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

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

`streamingDataInfo` should point to an array of structures, one per combination of enabled channel, downsampling mode and data type, to determine how many samples are available for that combination. For example, if you have enable two channels with both raw data and min-max aggregation, the array should contain four structures. The number of available samples at a given instant may not be the same for each channel due to the way samples are processed in blocks. If your application requires the same number of samples on each channel, process the minimum number of samples reported by any channel. The later samples remain in the buffer and can be processed on the next call.

Applicability
Streaming mode only

Arguments

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

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

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

`triggerInfo,`   a structure containing trigger information. See PICO_STREAMING_DATA_TRIGGER_INFO.

Returns	
	PICO_OK
	PICO_WAITING_FOR_DATA_BUFFERS - indicates that you need to call psospaSetDataBuffer() again as the previously supplied buffers have been filled. Note this return status does not mean the call has failed: if the `streamingDataInfo` structures indicate a non-zero number of samples (completing the previous buffer) then these are still valid data which should be read by the user.

3.12.1   PICO_STREAMING_DATA_INFO

A list of structures of this type is passed to <u>psospaGetStreamingLatestValues()</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

Set by the user:

`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.

Set by driver when the function returns:

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

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

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

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

3.12.2   PICO_STREAMING_DATA_TRIGGER_INFO

A structure of this type is returned by <u>psospaGetStreamingLatestValues()</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 has stopped capturing due to `autoStop` being set and the requested number of samples having been collected (1) or not (0).

3.13   psospaGetTimebase - get available timebases

```
PICO_STATUS psospaGetTimebase
(
  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 psospaSetChannelOn() or psospaSetChannelOff().

The easiest way to find a suitable timebase is to call psospaNearestSampleIntervalStateless(). Alternatively, you can estimate the timebase number that you require, representing a desired sample interval in picoseconds, then pass this timebase to psospaGetTimebase() 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 psospaOpenUnit().

`timebase`,   in picoseconds, 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_TOO_MANY_SAMPLES
PICO_INVALID_CHANNEL
PICO_INVALID_TIMEBASE
PICO_INVALID_PARAMETER
PICO_SEGMENT_OUT_OF_RANGE
PICO_NO_CHANNELS_OR_PORTS_ENABLED
```

3.14   psospaGetTriggerInfo - get trigger timing information

```
PICO_STATUS psospaGetTriggerInfo
(
  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 `psospaOpenUnit()`.

`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. Must be equal to (or less than) the number of structures in `*triggerInfo`.

Returns

```
PICO_OK
PICO_DEVICE_SAMPLING
PICO_SEGMENT_OUT_OF_RANGE
PICO_NULL_PARAMETER
PICO_NO_SAMPLES_AVAILABLE
```

3.14.1   PICO_TRIGGER_INFO - structure

A list of structures of this type containing trigger information is written by <u>psospaGetTriggerInfo()</u> 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. This value may be the logical `OR` of multiple status values:

PICO_DEVICE_TIME_STAMP_RESET, the time stamp per waveform segment has been reset.

PICO_TRIGGER_TIME_NOT_REQUESTED, when requesting the <u>psospaGetTriggerTimeOffset</u> the
trigger time has not been set as described under `triggerTime` below.

PICO_TRIGGER_TIME_BUFFER_NOT_SET, trigger time buffer not set (see below).

PICO_TRIGGER_TIME_FAILED_TO_CALCULATE, the trigger time failed to be calculated.

For example the value 0x03000001 is a combination of:

PICO_DEVICE_TIME_STAMP_RESET bit or'ed with PICO_TRIGGER_TIME_NOT_REQUESTED

`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. This is interpolated to greater precision than one sample
interval, using adjacent sample values. `triggerTime` is only available when trigger data has been requested
from the oscilloscope either by using PICO_RATIO_MODE_TRIGGER with <u>psospaGetValues()</u> or its
comparable bulk or async versions, or by using
PICO_RATIO_MODE_TRIGGER_DATA_FOR_TIME_CALCULATION with
<u>psospaGetValuesOverlapped()</u>. See <u>Downsampling modes</u> for more information. Otherwise, `status`
includes the value PICO_TRIGGER_TIME_NOT_REQUESTED and the `triggerTime` is zero.

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

`missedTriggers,` the number of trigger events, if any, detected since the trigger point of the previous segment. The trigger circuit is constantly counting events, regardless of whether the trigger is armed, so this includes events which may have occurred during the post-trigger time of the previous capture, or the pre-trigger time of the current capture, or in the "dead time" between captures (trigger re-arm time). By dividing `missedTriggers` by the `timeStampCounter` difference between the previous and current capture, you can calculate the input signal frequency even if this is faster than the scope's trigger re-arm time.

`timeStampCounter,` the time in samples from the first capture to the current capture. See Time stamping.

3.14.2   Time stamping

The `timeStampCounter` parameter in the `PICO_TRIGGER_INFO` structure allows you to determine the time interval between the trigger points of consecutive captures with the same settings, in block or rapid block mode. Only events causing the scope to trigger are timestamped. Additional trigger events occurring within a capture or in the trigger rearm time between captures cannot be timestamped.

To get the offset between the respective segment trigger points, in sample intervals at the current timebase, subtract the `timeStampCounter` for each segment from the previous segment's timestamp. The timestamps are accurate to one sample interval at the current timebase.

The timestamp of the first segment captured after changing any scope settings is arbitrary, and is only provided to allow you to calculate the offset of subsequent segments. The timestamp counter may either maintain or reset its value when scope settings are changed, and your code must not rely on particular behavior in this respect but should instead check the status code.

The status code returned for each segment indicates whether the timestamp is valid. For example, if you set up 10 segments in memory and then carry out two rapid block runs of 5 captures each, the status codes for segments 0 and 5 may have the bit-flag `PICO_DEVICE_TIME_STAMP_RESET` set, if you changed any settings since the previous run, indicating that the timestamp for that segment is arbitrary. The other segments will not have this flag set, indicating that the timestamp is valid and can be used to determine the time offset from the previous segment. `PICO_DEVICE_TIME_STAMP_RESET` is a bit-flag so may be masked with any other status flag that relates to that segment.

You can convert the intervals between segments from sample counts to time intervals if required. The current sample interval can be found by using the timebase that was passed to `psospaRunBlock` in conjunction with `psospaGetTimebase`.

`timeStampCounter` is a 56-bit unsigned value and will eventually wrap around. Your code must handle this correctly, for example by masking the results of any arithmetic to the lower 56 bits. If the timestamp wraps around more than once between two adjacent segments, this cannot be detected. This will only happen if the interval between two adjacent trigger events exceeds 100 days (at the fastest timebase, or longer for slower timebases), so is unlikely to be a concern in practical applications. Note that calculating the time offset between adjacent segments, rather than to the first segment, reduces the complexity of dealing with wraparounds.

3.15　psospaGetTriggerTimeOffset - get timing corrections

```
PICO_STATUS psospaGetTriggerTimeOffset
(
  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.

This is the same as the `triggerTime` value obtained from [psospaGetTriggerInfo()](#), which also provides additional information. There is no need to call both functions.

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

The trigger time offset is only available when trigger data has been requested from the oscilloscope either by using `PICO_RATIO_MODE_TRIGGER` with [psospaGetValues()](#) or its comparable bulk or async versions, or by using `PICO_RATIO_MODE_TRIGGER_DATA_FOR_TIME_CALCULATION` with [psospaGetValuesOverlapped()](#). See [Downsampling modes](#) for more information. Otherwise, `PICO_TRIGGER_TIME_NOT_REQUESTED` is returned.

Applicability
[Block mode](#), [rapid block mode](#)

Arguments

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

`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_DEVICE_SAMPLING`
`PICO_SEGMENT_OUT_OF_RANGE`
`PICO_NULL_PARAMETER`
`PICO_NO_SAMPLES_AVAILABLE`
`PICO_TRIGGER_TIME_NOT_REQUESTED`

3.16　　psospaGetUnitInfo - get information about device

```
PICO_STATUS psospaGetUnitInfo
(
  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 psospaEnumerateUnits().

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 – an initial call like this allows you to determine the required length of the string before allocating it.

`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_NULL_PARAMETER
PICO_INVALID_INFO
PICO_INFO_UNAVAILABLE
```
`PICO_STRING_BUFFER_TO_SMALL`,　`stringLength` is insufficient for the required data, but non-zero.

info		Example
0x00	`PICO_DRIVER_VERSION` - Version number of psospa DLL	`1,0,0,1`
0x01	`PICO_USB_VERSION` - Type of USB connection to device: 2.0 or 3.0	`3.0`
0x02	`PICO_HARDWARE_VERSION` - Hardware version of device	`1`
0x03	`PICO_VARIANT_INFO` - Model number of device	`3418E`
0x04	`PICO_BATCH_AND_SERIAL` - Batch and serial number of device	`10001/0001`
0x05	`PICO_CAL_DATE` - Calibration date of device	`30Sep24`
0x06	`PICO_KERNEL_VERSION` - Version of kernel driver	`1,1,2,4`
0x07	`PICO_DIGITAL_HARDWARE_VERSION` - Hardware version of the digital section	`1`
0x08	`PICO_ANALOGUE_HARDWARE_VERSION` - Hardware version of the analog section	`1`
0x09	`PICO_FIRMWARE_VERSION_1` - Version information of Firmware 1	`1,0,0,1`
0x0A	`PICO_FIRMWARE_VERSION_2` - Version information of Firmware 2	`1,0,0,1`
0x0F	`PICO_FIRMWARE_VERSION_3` - Version information of Firmware 3	`1,0,0,1`
0x10	`PICO_FRONT_PANEL_FIRMWARE_VERSION` - Version of front-panel microcontroller firmware	`1,0,0,1`

psospa-3   46

3.17   psospaGetValues - get data after a capture has completed

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

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

Applicability

All modes.

Arguments

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

`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_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_MEMORY
PICO_RATIO_MODE_NOT_SUPPORTED
```

3.17.1   Downsampling modes

Various methods of data reduction, or **downsampling**, are possible with the PicoScope 3000E 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 psospaGetValues(). 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.
	When using trigger delay, this is the original event causing the trigger and not the delayed point. This data is available even when the original trigger point falls outside the main preTrigger and postTrigger data.
	Trigger data must be retrieved before attempting to get the trigger time using psospaGetTriggerInfo(), psospaGetTriggerTimeOffset() or psospaGetValuesTriggerTimeOffsetBulk()
`PICO_RATIO_MODE_RAW`	No downsampling. Returns raw data values.
`PICO_RATIO_MODE_TRIGGER_DATA_FOR_TIME_CALCULATION`	In overlapped mode only, causes trigger data to be retrieved from the scope to calculate the trigger time for psospaGetTriggerInfo(), psospaGetTriggerTimeOffset() or psospaGetValuesTriggerTimeOffsetBulk(), without requiring a user buffer to be set for this data.

See [psospaGetValuesOverlapped()](#).

3.18   psospaGetValuesAsync - read data without blocking

```
PICO_STATUS psospaGetValuesAsync
(
  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 psospaGetValues()
```

`lpDataReady`,   a pointer to the user-supplied `psospaDataReady()` callback function that will be called when the data is ready.

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

Returns

```
PICO_OK
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
```

3.19   psospaGetValuesBulk - read multiple segments

```
PICO_STATUS psospaGetValuesBulk
(
  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 psospaGetValues()

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

Returns

```
PICO_OK
PICO_INVALID_PARAMETER
PICO_SEGMENT_OUT_OF_RANGE
PICO_NO_SAMPLES_AVAILABLE
PICO_STARTINDEX_INVALID
PICO_INVALID_SAMPLERATIO
```

3.20　psospaGetValuesBulkAsync - read multiple segments without blocking

```
PICO_STATUS psospaGetValuesBulkAsync
(
  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 psospaGetValues()

fromSegmentIndex,
toSegmentIndex: see psospaGetValuesBulk()

lpDataReady,
pParameter: see psospaGetValuesAsync()
```

Returns

```
PICO_OK
PICO_INVALID_PARAMETER
PICO_SEGMENT_OUT_OF_RANGE
PICO_NO_SAMPLES_AVAILABLE
PICO_STARTINDEX_INVALID
```

3.21   psospaGetValuesOverlapped - make a deferred request for data before running the scope

```
PICO_STATUS psospaGetValuesOverlapped
(
  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 block or rapid block mode. The request will be executed, and the arguments validated, when you call psospaRunBlock(). The advantage of this method is that the driver makes contact with the scope only once, when you call psospaRunBlock(), compared with the two contacts that occur when you use the conventional psospaRunBlock(), psospaGetValues() calling sequence. This slightly reduces the dead time between successive captures.

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

To stop collecting data, call psospaStopUsingGetValuesOverlapped().

Applicability

Rapid block mode

Arguments

```
handle,
startIndex,
noOfSamples,
downSampleRatio,
downSampleRatioMode,
overflow,   see psospaGetValues()

fromSegmentIndex,
toSegmentIndex, see psospaGetValuesBulk().
```

Returns

```
PICO_OK
PICO_INVALID_PARAMETER
```

3.21.1   Using GetValuesOverlapped()

1.   Open the oscilloscope using `psospaOpenUnit()`.
2.   Select channel ranges and AC/DC coupling using `psospaSetChannelOn()`.
3.   Use `psospaNearestSampleIntervalStateless()`, to find a valid sampling interval to use
4.   Use the trigger setup functions `psospaSetTriggerChannelConditions()`, `psospaSetTriggerChannelDirections()` and `psospaSetTriggerChannelProperties()` to set up the trigger if required.
5.   Use `psospaSetDataBuffer()` to tell the driver where your memory buffer is.
6.   Set up the transfer of the block of data from the oscilloscope using `psospaGetValuesOverlapped()`.
7.   Start the oscilloscope running using `psospaRunBlock()`.
8.   Wait until the oscilloscope is ready using the `psospaBlockReady()` callback (or poll using `psospaIsReady()`).
9.   Display or process the data.
10. Repeat steps 7 to 9 if needed.
11. Stop the oscilloscope by calling `psospaStop()`.

A similar procedure can be used with rapid block mode.

3.22   psospaGetValuesTriggerTimeOffsetBulk - get trigger time offsets for multiple segments

```
PICO_STATUS psospaGetValuesTriggerTimeOffsetBulk
(
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 [psospaGetTriggerTimeOffset()](#) once for each waveform required. See [psospaGetTriggerTimeOffset()](#) for an explanation of trigger time offsets.

Applicability

[Rapid block mode](#)

Arguments

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

`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 [psospaGetTriggerTimeOffset()](#).

`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_NULL_PARAMETER
PICO_DEVICE_SAMPLING
PICO_SEGMENT_OUT_OF_RANGE
PICO_NO_SAMPLES_AVAILABLE
PICO_TRIGGER_TIME_NOT_REQUESTED
```

3.23   psospaGetVariantDetails - get specification details in JSON format

```
PICO_STATUS psospaGetVariantDetails
(
  const int8_t*    variantName,
  int16_t          variantNameLength,
  int8_t*          outputString,
  int32_t*         outputStringLength
  PICO_TEXT_FORMAT textFormat
)
```

This function returns a string (in the requested `textFormat` format) containing specification details of the `psospa` variant requested by `variantName`. The data is copied into the location provided by `outputString` only if a non-null location is provided and `outputStringLength` is sufficient. If making an initial call to find the required buffer size, call with `outputStringLength` set to zero. `outputString` may be `nullptr` in this case and PICO_OK will still be returned.

Applicability

All modes

Arguments

`variantName,`   the string variant name for which data is being requested, for example "3418E". The variant name for a currently connected device can be found using `psospaGetUnitInfo()` if the unit is open, or `psospaEnumerateUnits()` before opening. Passing a `variantName` of `"all-series"` returns a list of all supported variant names for each series supported by the driver.

`variantNameLength,`   the size of the variantName array.

`outputString,`   the location to copy the 'textFormat' formatted object string to.

`outputStringLength,`   the size of the outputString array. On return, the size of the buffer that has been copied if successful, or otherwise the size of the buffer that would be required to contain the requested data.

`textFormat,`   the text format type to request. Supplying PICO_JSON_DATA will return the given device's capabilities structured in json. Supplying PICO_JSON_SCHEMA will return the given device's JSON schema.

Returns

PICO_OK
PICO_INVALID_PARAMETER
PICO_INVALID_VARIANT
PICO_NULL_PARAMETER,   a required parameter is `nullptr`
PICO_STRING_BUFFER_TO_SMALL, `outputStringLength` is insufficient for the required data, but non-zero.

3.23.1   PICO_TEXT_FORMAT textFormat

```
typedef enum enPicoTextFormat
{
  PICO_JSON_DATA,
  PICO_JSON_SCHEMA,
} PICO_TEXT_FORMAT;
```

This is used to identify the format of the text data requested.

Applicability

Calls to <u>psospaGetVariantDetails</u>

Values

`PICO_JSON_DATA,` is used as an identifier for JSON data

`PICO_JSON_SCHEMA,` is used as an identifier for JSON schema

3.24   psospaIsReady - get status of block capture

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

This function may be used instead of a callback function to receive data from **psospaRunBlock()**. To use this method, pass a NULL pointer as the `lpReady` argument to **psospaRunBlock()**. 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 **psospaOpenUnit()**.

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

Returns

PICO_OK
PICO_NULL_PARAMETER

3.25   psospaMemorySegments - set number of memory segments

```
PICO_STATUS psospaMemorySegments
(
  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 can use up to 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.

See also psospaMemorySegmentsBySamples() which sets up the memory segments to each fit a required number of samples.

Applicability

All modes

Arguments

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

`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_TOO_MANY_SEGMENTS
PICO_MEMORY
```

3.26   psospaMemorySegmentsBySamples - set size of memory segments

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

This function sets the number of samples per memory segment. Like psospaMemorySegments() 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 psospaOpenUnit().

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_TOO_MANY_SEGMENTS
PICO_MEMORY

3.27 psospaNearestSampleIntervalStateless - get nearest sampling interval

```
PICO_STATUS psospaNearestSampleIntervalStateless
(
  int16_t                handle,
  PICO_CHANNEL_FLAGS     enabledChannelFlags,
  double                 timeIntervalRequested,
  uint8_t                roundFaster,
  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 `psospaOpenUnit()`.

`enabledChannelFlags,` see `psospaGetMinimumTimebaseStateless()`.

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

`roundFaster,` if the requested time interval cannot be exactly achieved then, when `0`, the timebase slower than the requested is returned, when 1 the timebase faster than the requested is returned.

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

`timebase,` on exit, the number of the nearest available timebase, in picoseconds.

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

Returns

```
PICO_OK
PICO_NO_SAMPLES_AVAILABLE
PICO_NULL_PARAMETER
PICO_INVALID_PARAMETER
PICO_SEGMENT_OUT_OF_RANGE
PICO_TOO_MANY_SAMPLES
PICO_NO_CHANNELS_OR_PORTS_ENABLED
PICO_INVALID_DIGITAL_PORT
```

3.28   psospaNoOfStreamingValues - get number of captured samples

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

This function returns the number of samples available after data collection in <u>streaming mode</u>. Call it after calling <u>psospaStop()</u>.

Applicability

<u>Streaming mode</u>

Arguments

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

noOfValues, on exit, the number of samples of raw data, per enabled channel, available for retrieval after the end of the capture.

Returns

```
PICO_OK
PICO_NULL_PARAMETER
PICO_NO_SAMPLES_AVAILABLE
PICO_NOT_USED
PICO_BUSY
```

3.29   psospaOpenUnit - open a scope device

```
PICO_STATUS psospaOpenUnit
(
  int16_t               * handle,
  int8_t                * serial,
  PICO_DEVICE_RESOLUTION   resolution
  PICO_USB_POWER_DETAILS * powerDetails
)
```

This function opens a PicoScope 3000E 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.

If the function returns PICO_FIRMWARE_UPDATE_REQUIRED_TO_USE_DEVICE_WITH_THIS_DRIVER, all other API calls that perform operations with the same device will fail with the same return value until psospaStartFirmwareUpdate() is called. Users should avoid unplugging the device during this operation, otherwise there is a small chance that the firmware could be corrupted.

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).

powerDetails, returns details about the unit power setup, can be null if not required.

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_CONFIG_FAIL_AWG
PICO_INITIALISE_FPGA
PICO_FIRMWARE_UPDATE_REQUIRED_TO_USE_DEVICE_WITH_THIS_DRIVER - call psospaCheckForUpdate() and then psospaStartFirmwareUpdate()
PICO_POWER_MANAGER (power setup is invalid)
PICO_USB_VERSION_NOT_SUPPORTED (USB 1.x connections not supported)

3.29.1   PICO_USB_POWER_DETAILS

```
typedef struct tPicoUsbPowerDetails
{
  uint8_t powerErrorLikely_;
  PICO_USB_POWER_DELIVERY dataPort_;
  PICO_USB_POWER_DELIVERY powerPort_;
} PICO_USB_POWER_DETAILS;
```

Applicability

Set by psospaOpenUnit() if a non-null pointer is passed for powerDetails

Elements

powerErrorLikely_, If psospaOpenUnit fails, this indicates whether it was likely that the failure was due to an issue with the power supply. Non-zero indicates that a power failure was likely, zero indicates that it was unlikely

dataPort_, USB power details for the data port. See PICO_USB_POWER_DELIVERY

powerPort_, USB power details for the power port. See PICO_USB_POWER_DELIVERY

3.29.2   PICO_USB_POWER_DELIVERY

```
typedef struct tPicoUsbPowerDelivery
{
  uint8_t valid_;
  uint32_t busVoltagemV_;
  uint32_t rpCurrentLimitmA_;
  uint8_t partnerConnected_;
  uint8_t ccPolarity_;
  PICO_USB_POWER_DELIVERY_DEVICE_TYPE attachedDevice_;
  uint8_t contractExists_;
  uint32_t currentPdo_;
  uint32_t currentRdo_;
} PICO_USB_POWER_DELIVERY;
```

Applicability

Forms part of the PICO_USB_POWER_DETAILS set by psospaOpenUnit()

Elements

valid_, non-zero, indicates that the following data is valid. Zero indicates that they are not (likely no USB connection on this port)

busVoltagemV_, the USB voltage in mV

rpCurrentLimitmA_, the current limit set by the USB CC lines in mA

partnerConnected_, non-zero indicates that there is a compatible USB-C power connected.

`ccPolarity_,` indicates which way up the USB-C connector is

`attachedDevice_,` indicates the type of device connected to the unit via this port

`contractExists_,` non-zero indicates that an explicit PD contract exists between the unit and its partner

`currentPdo_,` current power data object, refer to chapter 6.4 of the PD 2.0 or 3.0 specifications

`currentRdo_,` current request data object, refer to chapter 6.4 of the PD 2.0 or 3.0 specifications

`ccPolarity_,` indicates which way up the USB-C connector is

3.30   psospaPingUnit - check if device is still connected

```
PICO_STATUS psospaPingUnit
(
  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 psospaOpenUnit().

Returns

`PICO_OK`
`PICO_BUSY`
`PICO_NOT_RESPONDING`

3.31   psospaQueryMaxSegmentsBySamples - get number of segments

```
PICO_STATUS psospaQueryMaxSegmentsBySamples
(
  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. It does not change the current segment configuration of the scope.

Applicability

All modes

Arguments

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

`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_NO_SAMPLES_AVAILABLE
PICO_NULL_PARAMETER
PICO_INVALID_PARAMETER
PICO_SEGMENT_OUT_OF_RANGE
PICO_TOO_MANY_SAMPLES
```

3.32   psospaQueryOutputEdgeDetect – check if output edge detection is enabled

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

This function queries the state of the trigger function edge detection, set with psospaSetOutputEdgeDetect().

Applicability
All modes

Arguments

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

`state,` on exit, the current state of trigger function edge detection.
   `0` = off
   `1` = on

Returns
`PICO_OK`

3.33   psospaResetChannelsAndReportAllChannelsOvervoltageTripStatus - reset 50 Ω input protection

```
PICO_STATUS psospaResetChannelsAndReportAllChannelsOvervoltageTripStatus
(
  int16_t                             handle,
  PICO_CHANNEL_OVERVOLTAGE_TRIPPED    * allChannelsTrippedStatus,
  uint8_t                             nChannelTrippedStatus
)
```

In 50Ω coupling mode, the oscilloscope hardware includes an overvoltage protection circuit which disconnects the input to prevent damage.

This function resets all oscilloscope channels and then reports the overvoltage trip status for all channels. Use this to reset after an overvoltage trip event, and check that the channels haven't immediately tripped again due to a continuing overvoltage.

Applicability
All modes

Arguments

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

`allChannelsTrippedStatus,` a pointer to an array of `PICO_CHANNEL_OVERVOLTAGE_TRIPPED` structs. On exit, the overvoltage trip status of each channel will be written to this array.

`nChannelTrippedStatus,` the number of `PICO_CHANNEL_OVERVOLTAGE_TRIPPED` structs in the above array.

Returns
`PICO_OK`
`PICO_HARDWARE_CAPTURING_CALL_STOP`
`PICO_NULL_PARAMETER`
`PICO_INVALID_PARAMETER`
`PICO_NOT_SUPPORTED_BY_THIS_DEVICE`

3.34   psospaReportAllChannelsOvervoltageTripStatus-check if 50 Ω input protection has tripped

```
PICO_STATUS psospaReportAllChannelsOvervoltageTripStatus
(
  int16_t                           handle,
  PICO_CHANNEL_OVERVOLTAGE_TRIPPED  * allChannelsTrippedStatus,
  uint8_t                           nChannelTrippedStatus
)
```

This function reports the overvoltage trip status for all channels without resetting their status. Use it to find out which channels caused an overvoltage trip event.

Applicability

All modes

Arguments

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

`allChannelsTrippedStatus,` a pointer to an array of `PICO_CHANNEL_OVERVOLTAGE_TRIPPED` channel status flags. On exit, the overvoltage trip status of each channel will be written to this array.

`nChannelTrippedStatus,` the number of `PICO_CHANNEL_OVERVOLTAGE_TRIPPED` structs in the above array.

Returns

`PICO_OK`

`PICO_NULL_PARAMETER`
`PICO_INVALID_PARAMETER`
`PICO_NOT_SUPPORTED_BY_THIS_DEVICE`

3.34.1   PICO_CHANNEL_OVERVOLTAGE_TRIPPED structure

```
typedef struct tPicoChannelOvervoltageTripped
{
  PICO_CHANNEL   channel_;
  uint8_t        tripped_;
} PICO_CHANNEL_OVERVOLTAGE_TRIPPED;
```

This structure contains information about the overvoltage trip on a given channel. An overvoltage trip occurs when an oscilloscope channel in 50 Ω coupling mode detects an excessive voltage on its input, and disconnects the scope input to prevent damage.

Applicability

Analog input channels

Elements

`channel_,` the oscilloscope channel to which the information applies.

`tripped_,` a flag indicating whether the overvoltage trip occurred (non-zero) or did not occur (zero).

3.35 psospaRunBlock - start block mode capture

```
PICO_STATUS psospaRunBlock
(
  int16_t              handle,
  uint64_t             noOfPreTriggerSamples,
  uint64_t             noOfPostTriggerSamples,
  uint32_t             timebase,
  double             * timeIndisposedMs,
  uint64_t             segmentIndex,
  psospaBlockReady     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 `psospaOpenUnit()`.

`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, indicating the sample interval, in picoseconds. See the guide to calculating timebase values. If the value you pass does not correspond to a valid sampling rate for the current scope setup, the scope will select the nearest available sampling rate. It is recommended to call `psospaNearestSampleIntervalStateless()` to determine a valid sample interval before calling this function.

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

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

Returns

PICO_OK

PICO_SEGMENT_OUT_OF_RANGE

PICO_INVALID_CHANNEL

PICO_INVALID_TRIGGER_CHANNEL

PICO_INVALID_CONDITION_CHANNEL

PICO_TOO_MANY_SAMPLES

PICO_INVALID_TIMEBASE

PICO_CONFIG_FAIL

PICO_INVALID_PARAMETER

PICO_TRIGGER_ERROR

PICO_FW_FAIL

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   psospaRunStreaming - start streaming mode capture

```
PICO_STATUS psospaRunStreaming
(
  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 <u>streaming mode</u>. The device can return either raw or <u>downsampled</u> data to your application while streaming is in progress. Call <u>psospaGetStreamingLatestValues()</u> to retrieve the data. See <u>Using streaming mode</u> 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 <u>psospaOpenUnit()</u>.

`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:
  <u>PICO_FS</u>
  <u>PICO_PS</u>
  <u>PICO_NS</u>
  <u>PICO_US</u>
  <u>PICO_MS</u>
  <u>PICO_S</u>

`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 <u>psospaGetValues()</u>.

Returns

`PICO_OK`

`PICO_NULL_PARAMETER`

`PICO_INVALID_PARAMETER`

`PICO_STREAMING_FAILED`

`PICO_TRIGGER_ERROR`

`PICO_INVALID_SAMPLE_INTERVAL`

`PICO_INVALID_BUFFER`

`PICO_FW_FAIL`

`PICO_MEMORY`

`PICO_SIGGEN_GATING_AUXIO_ENABLED` (signal generator is set to trigger on AUX input with incompatible trigger type)

`PICO_HARDWARE_CAPTURING_CALL_STOP`

`PICO_STREAMING_COMBINATION_OF_RAW_DATA_AND_ONE_AGGREGATION_DATA_TYPE_ALLOWED`

`PICO_TIME_UNITS_OUT_OF_RANGE`

`PICO_NO_SAMPLES_REQUESTED`

`PICO_TOO_FEW_REQUESTED_STREAMING_SAMPLES`

`PICO_TOO_MANY_SAMPLES`

`PICO_NOT_USED_IN_THIS_CAPTURE_MODE`

`PICO_INVALID_RATIO_MODE`

`PICO_STREAMING_DOES_NOT_SUPPORT_TRIGGER_RATIO_MODES`

`PICO_RATIO_MODE_NOT_SUPPORTED`

`PICO_THRESHOLD_UPPER_LOWER_MISMATCH`

`PICO_TRIGGER_CHANNEL_NOT_ENABLED`

`PICO_CONDITION_HAS_NO_TRIGGER_PROPERTY`

`PICO_INVALID_DIGITAL_PORT`

`PICO_TRIGGER_PORT_NOT_ENABLED`

`PICO_DIGITAL_DIRECTION_NOT_SET`

`PICO_WARNING_AUX_OUTPUT_CONFLICT`

`PICO_READ_SELECTION_OUT_OF_RANGE`

`PICO_RATIO_MODE_TRIGGER_MASKING_INVALID`

`PICO_USE_THE_TRIGGER_READ`

`PICO_USE_A_DATA_READ`

`PICO_RATIO_MODE_REQUIRES_NUMBER_OF_SAMPLES_TO_BE_SET`

`PICO_OPERATION_FAILED`

`PICO_ARGUMENT_OUT_OF_RANGE`

`PICO_BUFFERS_NOT_SET`

`PICO_STREAMING_ONLY_SUPPORTS_ONE_READ`

`PICO_NOT_RESPONDING_OVERHEATED`

`PICO_ENDPOINT_MISSING`

`PICO_UNKNOWN_ENDPOINT_REQUEST`

3.37   psospaSetAuxIoMode - configure the AUX IO connector

```
PICO_STATUS psospaSetAuxIoMode
(
    int16_t              handle
    PICO_AUXIO_MODE      auxIoMode
)
```

Configures the `AuxIO` mode/function using the `PICO_AUXIO_MODE` enum values.

Applicability

All modes

Arguments

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

`auxIoMode,` required AuxIO mode:
   `PICO_AUXIO_INPUT,` high-impedance input for use triggering the scope or AWG if configured.
   `PICO_AUXIO_HIGH_OUT,` constant logic high output.
   `PICO_AUXIO_LOW_OUT,` constant logic low output.
   `PICO_AUXIO_TRIGGER_OUT,` logic high pulse during the post-trigger acquisition time.

Returns

`PICO_OK`

`PICO_OPERATION_FAILED,` failed to change AuxIO mode

`PICO_WARNING_AUX_OUTPUT_CONFLICT,` the AuxIO mode has been set to an output mode while the scope or AWG is set to trigger on it. This is allowed but will result in the scope or AWG triggering on the output value of the AuxIO.

3.38   psospaSetChannelOff - disable one channel

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

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

Applicability

All modes

Arguments

handle,   the device identifier returned by psospaOpenUnit().

channel, see psospaSetChannelOn().

Returns

PICO_OK
PICO_INVALID_CHANNEL

3.39 psospaSetChannelOn - enable and set options for one channel

```
PICO_STATUS psospaSetChannelOn
(
    int16_t                    handle,
    PICO_CHANNEL               channel,
    PICO_COUPLING              coupling,
    int64_t                    rangeMin,
    int64_t                    rangeMax,
    PICO_PROBE_RANGE_INFO      rangeType,
    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.

To switch off, use psospaSetChannelOff().

For digital ports, see psospaSetDigitalPortOn().

Applicability
All modes

Arguments

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

`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

`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.

`rangeMin`, minimum value of the input range, expressed in the units selected by `rangeType` below.
`rangeMax`, maximum value of the range , expressed in the units selected by `rangeType` below.

`rangeType`, defines the units of `rangeMin`/`rangeMax` and any multiplication factor of the probe:

   PICO_PROBE_NONE_NV, range is defined in nanovolts (nV) with no probe scaling
   PICO_X1_PROBE_NV, range is defined in nanovolts (nV) with scaling for a 1:1 probe (same as no probe scaling)
   PICO_X10_PROBE_NV, range is defined in nanovolts (nV) with scaling for a 10:1 probe

`rangeMin` and `rangeMax` must correspond to one of the available input voltage ranges of the oscilloscope as described in the datasheet or returned by [psospaGetVariantDetails()](). For example, the 5 mV range would be represented as `rangeMin −5 000 000`, `rangeMax 5 000 000`, `rangeType PICO_PROBE_NONE_NV`.

`analogueOffset`, a voltage to add to the input channel before digitization. The allowable analog offset for a given input voltage range can be read from [psospaGetAnalogueOffsetLimits()](), or the values for all ranges are included in the JSON returned by [psospaGetVariantDetails()]().

`bandwidth`, the bandwidth limiter setting:
    `PICO_BW_FULL`, the scope's full specified bandwidth. This may vary depending on selected resolution and voltage range as described in the device datasheet, in which case the driver will automatically select the highest available bandwidth for the channel based on those settings.
    `PICO_BW_20MHZ:` −3 dB bandwidth limited to 20 MHz
    `PICO_BW_50MHZ:` −3 dB bandwidth limited to 50 MHz
    `PICO_BW_100MHZ:` −3 dB bandwidth limited to 100 MHz
    `PICO_BW_200MHZ:` −3 dB bandwidth limited to 200 MHz
    `PICO_BW_350MHZ:` −3 dB bandwidth limited to 350 MHz
    `PICO_BW_500MHZ:` −3 dB bandwidth limited to 500 MHz
    Available values vary by device model as described in the data sheet or reported by [psospaGetVariantDetails()]().

Returns

`PICO_OK`
`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_WARNING_PROBE_CHANNEL_OUT_OF_SYNC`

3.40　psospaSetDataBuffer - provide location of data buffer

```
PICO_STATUS psospaSetDataBuffer
(
  int16_t              handle,
  PICO_CHANNEL         channel,
  PICO_POINTER         buffer,
  uint64_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 `psospaSetDataBuffers()` 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 `psospaGetValues()`.

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 `psospaOpenUnit()`.

`channel,` the channel you want to use with the buffer. You must call this function for each channel for which you want to retrieve data.

`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
Note: valid data types vary by resolution and downsample ratio mode.

`waveform,` the segment index.

`downSampleRatioMode,` the downsampling mode. See `psospaGetValues()` for the available modes, but note that a single call to `psospaSetDataBuffer()` can only associate one buffer with one downsampling mode. If you intend to call `psospaGetValues()` with more than one downsampling mode activated, then you must call `psospaSetDataBuffer()` 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_CHANNEL`
`PICO_RATIO_MODE_NOT_SUPPORTED`
`PICO_INVALID_PARAMETER`

3.41   psospaSetDataBuffers - provide locations of both data buffers

```
PICO_STATUS psospaSetDataBuffers
(
  int16_t              handle,
  PICO_CHANNEL         channel,
  PICO_POINTER         bufferMax,
  PICO_POINTER         bufferMin,
  uint64_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 `psospaSetDataBuffer()` instead.

Applicability
Block and streaming modes with aggregation.

Arguments
`handle`, the device identifier returned by `psospaOpenUnit()`.

`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 `psospaSetDataBuffer()`.

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

`action`, see `psospaSetDataBuffer()`.

Returns
```
PICO_OK
PICO_INVALID_CHANNEL
PICO_RATIO_MODE_NOT_SUPPORTED
PICO_INVALID_PARAMETER
```

3.42   psospaSetDeviceResolution – set the hardware resolution

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

This function sets the sampling resolution of the device. At 10-bit resolution, the maximum capture buffer length and sample rate is half that of 8-bit mode.

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

Applicability
All modes.

Arguments
`handle,`   the device identifier returned by `psospaOpenUnit()`.

`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.

3.42.1   PICO_DEVICE_RESOLUTION enumerated type

```
typedef enum enPicoDeviceResolution
{
  PICO_DR_8BIT  = 0,
  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 `psospaSetDeviceResolution()` etc.

Values
`PICO_DR_8BIT`   – 8-bit resolution (256 levels)
`PICO_DR_10BIT`   – 10-bit resolution (1024 levels)

3.43   psospaSetDigitalPortOff – switch off a digital port

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

This function switches off a given digital port.

Applicability

All modes.
Any model with MSO channels fitted.

Arguments

handle, the device identifier returned by psospaOpenUnit().

port, see psospaSetDigitalPortOn().

Returns

PICO_OK
PICO_INVALID_DIGITAL_PORT
PICO_NOT_SUPPORTED_BY_THIS_DEVICE

3.44　psospaSetDigitalPortOn – set up and enable a digital port

```
PICO_STATUS psospaSetDigitalPortOn
(
  int16_t                    handle,
  PICO_CHANNEL               port,
  double                     logicThresholdLevelVolts
)
```

This function switches on a digital port and sets its corresponding logic threshold voltage.

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
All modes.
Any model with MSO channels fitted.

Arguments
`handle,`　the device identifier returned by `psospaOpenUnit()`.

`port,`　identifies the digital port on the oscilloscope:
　　`PICO_PORT0,`　channels D0–D7
　　`PICO_PORT1,`　channels D8–D15

`logicThresholdLevelVolts,`　the threshold voltage for the port in volts, used to distinguish the digital 0 and 1 states.

Returns
`PICO_OK,`　the PicoScope is functioning correctly.

`PICO_INVALID_DIGITAL_PORT,`　the requested digital port number is out of range.

`PICO_INVALID_PARAMETER,`　the `logicThresholdLevelVolts` parameter is not valid.

`PICO_NOT_SUPPORTED_BY_THIS_DEVICE,`　the PicoScope does not have digital ports.

3.45　psospaSetLedBrightness - set brightness of LEDs

```
PICO_STATUS psospaSetLedBrightness
(
  int16_t                              handle,
  uint8_t                              brightness
)
```

Sets the brightness of all configurable LEDs. The new brightness will only take effect the next time that LED is applied by calling psospaRunBlock(), psospaRunStreaming(), psospaSetAuxIoMode() or psospaSigGenApply() as appropriate.

Applicability
All modes

Arguments
`handle,` the device identifier returned by psospaOpenUnit().

`brightness,` brightness of the LEDs. 0-100 inclusive (default 79).

Returns
`PICO_OK`
`PICO_INVALID_PARAMETER` (brightness is out of range)

3.46 psospaSetLedColours - set the colors of specified LEDs

```
PICO_STATUS psospaSetLedColours
(
  int16_t                          handle,
  PICO_LED_COLOUR_PROPERTIES*      colourProperties,
  uint32_t                         nColourProperties
)
```

Sets the colors of the specified LEDs. The new color will only take effect the next time that LED is applied by calling psospaRunBlock(), psospaRunStreaming(), psospaSetAuxIoMode() or psospaSigGenApply() as appropriate.

Applicability

All modes

Arguments

handle, the device identifier returned by psospaOpenUnit().

colourProperties, array of the LEDs to set and the colour to set them to. Duplicate LEDs will take the colour of the last in the list.

nColourProperties, number of elements in the colourProperties array

Returns

PICO_OK

PICO_NULL_PARAMETER (the colourProperties pointer is null)

PICO_INVALID_PARAMETER (array length is invalid, one or more of the specified LEDs is not present on the device, one or more of the hue or saturation are out of range)

3.46.1 PICO_LED_COLOUR_PROPERTIES structure

```
typedef struct tPicoLedColourProperties
{
  PICO_LED_SELECT led_;
  uint16_t hue_;
  uint8_t saturation_;
} PICO_LED_COLOUR_PROPERTIES
```

This structure is used with psospaSetLedColours() to define the color for one LED using hue and saturation (HSV) values for the color.

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

Elements

led_, LED to set the colour of

hue_, hue of the LED, 0-359 inclusive

saturation_, saturation of the LED, 0-100 inclusive

3.46.2   PICO_LED_SELECT enumerated type

```
typedef enum enPicoLedSelect
{
  PICO_LED_CHANNEL_A = 0x00000000,
  PICO_LED_CHANNEL_B = 0x00000001,
  PICO_LED_CHANNEL_C = 0x00000002,
  PICO_LED_CHANNEL_D = 0x00000003,
  PICO_LED_AWG = 0x00010000,
  PICO_LED_AUX = 0x00020000
} PICO_LED_SELECT;
```

These values specify the LED channel within `PICO_LED_COLOUR_PROPERTIES` and `PICO_LED_STATE_PROPERTIES` structures.

3.47   psospaSetLedStates - set the states of specified LEDs

```
PICO_STATUS psospaSetLedStates
(
  int16_t                      handle,
  PICO_LED_STATE_PROPERTIES*   stateProperties,
  uint32_t                     nStateProperties
)
```

Sets the states of the specified LEDs.

Applicability

All modes

Arguments

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

`stateProperties`: array of the LEDs to set and the state to set them to. Duplicate LEDs will take the state of the last in the list.
`nStateProperties`: number of elements in the `stateProperties` array.

Returns

`PICO_OK`

`PICO_NULL_PARAMETER` (the `stateProperties` pointer is null)

`PICO_INVALID_PARAMETER` (array length is invalid, one or more of the specified LEDs is not present on the device)

3.47.1   PICO_LED_STATE_PROPERTIES structure

```
typedef struct tPicoLedStateProperties
{
  PICO_LED_SELECT              led_
  PICO_LED_STATE               state_;
} PICO_LED_STATE_PROPERTIES
```

This structure is used with <u>psospaSetLedStates()</u> to define the state for one LED using `PICO_LED_STATE` values below.

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

Elements

`led_`, LED to be configured

`state_`, new state of the LED:

    `PICO_LED_AUTO`, LED is controlled automatically to match the enabled state of its associated connector. This is the default behavior for all LEDs unless changed by the user.

    `PICO_LED_OFF`, LED is set off

    `PICO_LED_ON`, LED is set on

3.48   psospaSetNoOfCaptures - configure rapid block mode

```
PICO_STATUS psospaSetNoOfCaptures
(
  int16_t     handle,
  uint64_t    nCaptures
)
```

This function sets the number of captures to be collected in one run of <u>rapid block mode</u>. 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 <u>psospaOpenUnit()</u>.

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

Returns
`PICO_OK`
`PICO_INVALID_PARAMETER`

3.49　psospaSetOutputEdgeDetect – change triggering behavior

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

This function enables or disables edge detection on the output of the trigger function. In the default mode (edge detection on) the trigger will only fire when there is a change in the state of the trigger function. For example, if you set up trigger conditions representing "A OR B above threshold", the scope will only trigger at the moment the output of the logic function "A OR B" transitions from false to true. If either or both of the channels A or B are constantly high, the scope will not trigger. With edge detection off, the scope will trigger immediately when set running at any time that channel A or B is high.

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

Applicability

All modes

Arguments

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

`state`,　enable (1) or disable (0) the trigger function output edge detection.

Returns

`PICO_OK`

3.50   psospaSetPulseWidthDigitalPortProperties – set the digital port pulse-width trigger settings

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

This function sets the individual digital channels' pulse-width trigger directions. Each digital channel's direction consists of a channel name and a direction. If the channel is not included in the array of `PICO_DIGITAL_CHANNEL_DIRECTIONS,` the driver assumes the digital channel's pulse-width trigger direction is `PICO_DIGITAL_DONT_CARE.`

Applicability

All modes.
Any model with MSO channels fitted.

Arguments

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

`port,` identifies the digital port on the oscilloscope:

  `PICO_PORT0,` channels D0–D7
  `PICO_PORT1,` channels D8–D15

`directions,` a pointer to an array of `PICO_DIGITAL_CHANNEL_DIRECTIONS` structures describing the requested properties. The array can contain a single element describing the properties of one channel, or a number of elements describing several digital channels. If directions is `NULL,` all the digital channels' directions for the given port are set to `PICO_DIGITAL_DONT_CARE.` A digital channel that is not included in the array is also set to `PICO_DIGITAL_DONT_CARE.`

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

Returns

`PICO_OK`
`PICO_INVALID_DIGITAL_PORT,` the requested digital port number is out of range.
`PICO_INVALID_DIGITAL_CHANNEL,` the digital channel is not a value within the supported `PICO_PORT_DIGITAL_CHANNEL` range.
`PICO_INVALID_DIGITAL_TRIGGER_DIRECTION,` the digital trigger direction is not a valid trigger direction within the supported `PICO_DIGITAL_DIRECTION` range.

3.51   psospaSetPulseWidthQualifierConditions - specify how to combine channels

```
PICO_STATUS psospaSetPulseWidthQualifierConditions
(
  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.

Multiple conditions can be combined as described in `psospaSetTriggerChannelConditions()` . When the pulse width condition is met, the pulse width timer is reset and this signifies the start of a "pulse". The main trigger condition signifies the end of the "pulse" and if the pulse width qualifier is enabled and the time between these events meets the time condition set with `psospaSetPulseWidthQualifierProperties()`, the scope will trigger.

Applicability

All modes

Arguments

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

`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 `psospaSetTriggerChannelConditions()` for the list of actions.

Returns

`PICO_OK`
`PICO_NULL_PARAMETER`
`PICO_INVALID_ACTION`

3.52　psospaSetPulseWidthQualifierDirections - specify threshold directions

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

This function is used to set the trigger direction for each channel used in the pulse width time qualifier, which is an optional input to the triggering condition.

This function works in the same way as psospaSetTriggerChannelDirections() . Each channel has two trigger threshold comparators, so when using simple level triggers you can use one for the pulse width direction (for example, RISING), and the other for the main trigger direction (for example, FALLING_LOWER) signifying a positive pulse.

Applicability
All modes

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

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
PICO_NULL_PARAMETER

3.53　psospaSetPulseWidthQualifierProperties - specify threshold logic

```
PICO_STATUS psospaSetPulseWidthQualifierProperties
(
    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 time qualifier, which is an optional input to the triggering condition.

The pulse width timer is reset when an event occurs matching the user's conditions set using psospaSetPulseWidthQualifierConditions(), this represents the start of a "pulse". The qualifier is true when the time since the most recent start-of-pulse event meets the conditions set by this function (for example, less than 100 sample intervals ago).

Applicability

All modes

Arguments

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

`lower`, the lower pulse width threshold in sample intervals.

`upper`, the upper pulse width threshold in sample intervals

`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
PICO_NULL_PARAMETER

3.54   psospaSetSimpleTrigger - set up basic triggering

```
PICO_STATUS psospaSetSimpleTrigger
(
   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 `psospaOpenUnit()`.

`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 `psospaSetChannelOn()`.

`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 post-trigger sample being taken, in sample intervals.

`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

3.55   psospaSetTriggerChannelConditions - set triggering logic

```
PICO_STATUS psospaSetTriggerChannelConditions
(
  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 ANDed together. By calling the function multiple times, additional sets of trigger conditions can be defined which are then ORed together. This AND-OR logic allows you to create any possible Boolean function of the scope's inputs.

If complex triggering is not required, use psospaSetSimpleTrigger().

Applicability
All modes

Arguments

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

`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 AND 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, resets any previous conditions
  PICO_ADD, adds this condition to any previous conditions

To apply only the conditions passed in the current call, specify both PICO_CLEAR_ALL | PICO_ADD together.

Returns

PICO_OK
PICO_NO_TRIGGER_CONDITIONS_SET, returned when no trigger conditions are set while the action is PICO_ADD.
PICO_INVALID_ACTION, returned when the action contains invalid flags.
PICO_CLEAR_DATA_BUFFER_INVALID, returned when the action contains invalid buffer clear flags.
PICO_INVALID_CHANNEL, returned when a specified channel is invalid.
PICO_INVALID_TRIGGER_STATES, returned when a specified trigger state is invalid.
PICO_DUPLICATE_CONDITION_SOURCE, returned when a duplicate condition source is detected.

3.55.1 PICO_CONDITION structure

A structure of this type is passed to <u>psospaSetTriggerChannelConditions()</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 specifies a condition for just one of the scope's inputs. The <u>psospaSetTriggerChannelConditions()</u> function can AND together a number of these structures to produce the final trigger condition.

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,` one of the analog input channels

    `PICO_PORT0,` channels D0–D7, the first 8 digital inputs on MSO models

    `PICO_PORT1,` channels D8–D15, the next 8 digital inputs on MSO models

    `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.

3.56   psospaSetTriggerChannelDirections - set trigger directions

```
PICO_STATUS psospaSetTriggerChannelDirections
(
  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 `psospaOpenUnit()`.

`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_PARAMETER`

3.56.1   PICO_DIRECTION structure

A structure of this type is passed to psospaSetTriggerChannelDirections() 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. Each channel has two threshold comparators, designated as "upper" and "lower". These can be used independently, for example to set different directions or thresholds for the start and end of a time-qualified trigger using psospaSetPulseWidthQualifierDirections(), or used both together to set up a window or runt trigger as described below:

`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_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

3.57   psospaSetTriggerChannelProperties - set up triggering

```
PICO_STATUS psospaSetTriggerChannelProperties
(
  int16_t                          handle,
  PICO_TRIGGER_CHANNEL_PROPERTIES  * channelProperties
  int16_t                          nChannelProperties
  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 [psospaOpenUnit()](#).

`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.

`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_NULL_CHANNEL_PROPERTIES`, returned if `channelProperties` is `nullptr` and `nChannelProperties` is greater than `0`.
`PICO_INVALID_CHANNEL`, returned if a channel is not valid.
`PICO_DUPLICATED_CHANNEL`, returned if a channel appears more than once in `channelProperties`.
`PICO_THRESHOLD_OUT_OF_RANGE`, returned if the upper or lower threshold of a channel is out of the valid range.

3.57.1   TRIGGER_CHANNEL_PROPERTIES structure

A structure of this type is passed to psospaSetTriggerChannelProperties() 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 psospaSetTriggerChannelDirections(). 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
psospaSetTriggerChannelDirections().

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
psospaSetTriggerChannelDirections().

hysteresisLower, lower threshold hysteresis (see hysteresisUpper).

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

3.58   psospaSetTriggerDelay - set post-trigger delay

```
PICO_STATUS psospaSetTriggerDelay
(
  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 [psospaOpenUnit()].

`delay,`   the time between the trigger event occurring and the first post-trigger sample being captured. For example, if `delay=100`, the post-trigger samples will be counted starting 100 sample periods after the trigger event. At a [timebase] of 5 GS/s, or 200 ps per sample, the delay would be 100 x 200 ps = 20 ns. If pre-trigger samples are requested, these are immediately preceding the post-trigger samples, i.e. overlapping with the trigger delay time.

Returns

`PICO_OK`

3.59   psospaSetTriggerDigitalPortProperties - set digital port trigger directions

```
PICO_STATUS psospaSetTriggerDigitalPortProperties
(
  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
Any model with MSO channels fitted.

Arguments

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

`port`,   identifies the digital port on the oscilloscope:
  `PICO_PORT0`, channels D0–D7
  `PICO_PORT1`, channels D8–D15

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

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

Returns

`PICO_OK`
`PICO_INVALID_DIGITAL_PORT`, indicates that the provided port is not valid.
`PICO_INVALID_DIGITAL_CHANNEL`, indicates that one of the provided digital channels is not in range.
`PICO_INVALID_DIGITAL_TRIGGER_DIRECTION`, indicates that one of the provided digital trigger directions is not in range.

3.59.1   PICO_DIGITAL_CHANNEL_DIRECTIONS structure

A list of structures of this type is passed to [psospaSetTriggerDigitalPortProperties()](#) 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 within the selected port from `PICO_PORT_DIGITAL_CHANNEL0` up to `PICO_PORT_DIGITAL_CHANNEL7`. For example, if you have selected `PICO_PORT_1` then `PICO_PORT_DIGITAL_CHANNEL0` represents `D8` and `PICO_PORT_DIGITAL_CHANNEL7` represents `D15`.

`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:`	channel must transition (in either direction) to trigger

3.60   psospaSetTriggerHoldoffCounterBySamples - set the trigger holdoff time in sample intervals

```
PICO_STATUS psospaSetTriggerHoldoffCounterBySamples
(
    int16_t        handle,
    uint64_t       holdoffSamples,
)
```

This function sets the trigger holdoff time in sample intervals. Trigger holdoff allows you to set a period when the scope won't look for further trigger events after each triggered acquisition.

Applicability

All modes

Arguments

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

`holdoffSamples,` the time in sample intervals to disable looking for further triggers after the trigger event of each acquisition.

Returns

`PICO_OK`
`PICO_ARGUMENT_OUT_OF_RANGE`

3.61   psospaSigGenApply - set the signal generator running

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

This function sets the signal generator running using parameters previously configured by the other psospaSigGen... functions.

Applicability
All modes

Arguments

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

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

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

`triggerEnabled,` switches triggering of the signal generator on (1) or off (0).

`frequency,` on exit, the actual achieved signal generator frequency (or start frequency in sweep mode).

`stopFrequency,` on exit, the actual achieved signal generator frequency at the end of the sweep.

`frequencyIncrement,` on exit, the actual achieved frequency step size in sweep mode.

`dwellTime,` on exit, the actual achieved time in seconds between frequency steps in sweep mode.

Returns
`PICO_OK`
`PICO_WARNING_AUX_OUTPUT_CONFLICT,` indicates a conflict with the auxiliary output.
`PICO_BUSY,` indicates that the device is busy.

3.62   psospaSigGenFrequency - set output frequency

```
PICO_STATUS psospaSigGenFrequency
(
  int16_t                 handle,
  double                  frequencyHz
)
```

This function sets the frequency of the signal generator.

After configuring all required signal generator settings, call <u>psospaSigGenApply()</u> to apply them to the device.

Applicability
All modes

Arguments

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

`frequencyHz,` the desired frequency in hertz.

Returns
`PICO_OK`
`PICO_SIGGEN_FREQUENCY_OUT_OF_RANGE`

3.63   psospaSigGenFrequencyLimits - get signal generator limit values

```
PICO_STATUS psospaSigGenFrequencyLimits
(
  int16_t                  handle,
  PICO_WAVE_TYPE           waveType,
  uint64_t               * numSamples,
  double                 * minFrequencyOut,
  double                 * maxFrequencyOut,
  double                 * minFrequencyStepOut,
  double                 * maxFrequencyStepOut,
  double                 * minDwellTimeOut,
  double                 * maxDwellTimeOut
)
```

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

Applicability
All models

Arguments

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

`waveType,` the waveform that you intend to use.

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

`minFrequencyOut,` on exit, the lowest possible output frequency that can be achieved.

`maxFrequencyOut,` on exit, the highest possible output frequency that can be achieved.

`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`
`PICO_SIGGEN_WAVETYPE_NOT_SUPPORTED,` returned if the `waveType` is not supported.
`PICO_SIGGEN_NULL_PARAMETER,` returned if `numSamples` is `nullptr` when `waveType` is
`PICO_ARBITRARY.`
`PICO_SIG_GEN_PARAM,` returned if `numSamples` is `0` or greater than the maximum buffer size when
`waveType` is `PICO_ARBITRARY.`

3.64   psospaSigGenFrequencySweep - set signal generator to frequency sweep mode

```
PICO_STATUS psospaSigGenFrequencySweep
(
  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 psospaSigGenFrequency() to set the start frequency.

After configuring all required signal generator settings, call psospaSigGenApply() to apply them to the device.

Applicability
Signal generator.

Arguments
`handle,` the device identifier returned by psospaOpenUnit().

`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`
`PICO_SIGGEN_FREQUENCY_OUT_OF_RANGE,` returned if `stopFrequencyHz` or `frequencyIncrement` is out of the valid range.
`PICO_SIGGEN_SWEEPTYPE_INVALID,` returned if `sweepType` is not in the list of supported sweep types.
`PICO_SIGGEN_INVALID_SWEEP_PARAMETERS,` returned if the dwell count calculated from dwell time is outside the valid range.

3.65   psospaSigGenLimits - get signal generator parameters

```
PICO_STATUS psospaSigGenLimits
(
  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 [psospaOpenUnit()](#).

`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`
`PICO_NULL_PARAMETER`, returned if all pointers are `nullptr`.
`PICO_SIG_GEN_PARAM`, indicates that an invalid `PICO_SIGGEN_PARAMETER` was passed to the function.

3.66   psospaSigGenPause - stop the signal generator

```
PICO_STATUS psospaSigGenPause
(
  int16_t                    handle
)
```

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

Applicability

All modes

Arguments

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

Returns

PICO_OK
PICO_SIGGEN_SETTINGS_CHANGED_CALL_APPLY

3.67 psospaSigGenPhase - set signal generator using delta-phase value instead of a frequency

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

This function sets the signal generator output frequency (or the starting frequency, in the case of a frequency sweep) using a delta-phase value instead of a frequency. See Calculating deltaPhase for more information on how to calculate this value.

After configuring all required signal generator settings, call psospaSigGenApply() to apply them to the device.

Applicability

All modes

Arguments

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

`deltaPhase,` the desired delta phase.

Returns

`PICO_OK`
`PICO_SIGGEN_FREQUENCY_OUT_OF_RANGE`

3.67.1 Calculating deltaPhase

The signal generator uses direct digital synthesis (DDS) with a 48-bit phase accumulator that indicates the present location in the waveform. The top bits of the phase accumulator are used as an index into a buffer containing the arbitrary waveform. The remaining bits act as the fractional part of the index, enabling high-resolution control of output frequency and allowing the generation of lower frequencies.

The signal generator steps through the waveform by adding a *deltaPhase* value between 1 and *phaseAccumulatorSize*-1 to the phase accumulator every *dacPeriod* (= 1/*dacFrequency*). The generator produces a waveform at a frequency that can be calculated as follows:

$$outputFrequency = \frac{dacFrequency}{arbitraryWaveformSize} \times \frac{deltaPhase}{2^{(phaseAccumulatorSize - bufferAddressWidth)}}$$

where:

outputFrequency	= repetition rate of the complete arbitrary waveform
dacFrequency	= update rate of AWG DAC (see table below)
deltaPhase	= delta-phase value supplied to this function
phaseAccumulatorSize	= width in bits of phase accumulator (see table below)
bufferAddressWidth	= width in bits of AWG buffer address (see table below)
arbitraryWaveformSize	= length in samples of the user-defined waveform

Parameter	Value
dacFrequency	200 MHz
dacPeriod	1/*dacFrequency*. 5 ns.
phaseAccumulatorSize	48
bufferAddressWidth	15

3.68   psospaSigGenPhaseSweep - set signal generator to sweep using delta-phase values instead of frequency values

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

This function sets frequency sweep parameters for the signal generator using delta-phase values instead of frequency values. It assumes that you have previously called psospaSigGenPhase() to set the starting delta-phase.

After configuring all required signal generator settings, call psospaSigGenApply() to apply them to the device.

Applicability
All modes

Arguments

handle,   the device identifier returned by psospaOpenUnit().

stopDeltaPhase, the delta-phase at which the sweep should stop. You must set the starting delta-phase, deltaPhase, beforehand by calling psospaSigGenPhase().

deltaPhaseIncrement, the amount by which the delta-phase should change at each step of the sweep.

dwellCount, the number of samples for which the generator should wait between sweep 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
PICO_SIGGEN_FREQUENCY_OUT_OF_RANGE
PICO_SIGGEN_INVALID_SWEEP_PARAMETERS
PICO_SIGGEN_SWEEPTYPE_INVALID

3.69   psospaSigGenRange - set signal generator output voltages

```
PICO_STATUS psospaSigGenRange
(
  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.

After configuring all required signal generator settings, call psospaSigGenApply() to apply them to the device.

Applicability

All modes

Arguments

handle, the device identifier returned by psospaOpenUnit().

peakToPeakVolts, the signal generator's peak-to-peak output range in volts.

offsetVolts, the signal generator's output offset in volts.

The total output voltage accounting for both peak-to-peak and offset must fall within the signal generator's output voltage range described in the PicoScope 3000E data sheet or returned by psospaSigGenLimits().

Returns

PICO_OK
PICO_SIGGEN_PK_TO_PK
PICO_SIGGEN_OFFSET_VOLTAGE
PICO_SIGGEN_OUTPUT_OVER_VOLTAGE, if peakToPeak and offset are within their individual ranges but the combination is out of range.

3.70   psospaSigGenRestart - continue after pause

```
PICO_STATUS psospaSigGenRestart
(
  int16_t                    handle
)
```

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

Applicability

All modes

Arguments

handle,   the device identifier returned by psospaOpenUnit().

Returns

PICO_OK

PICO_SIGGEN_SETTINGS_CHANGED_CALL_APPLY,   the signal generator has been partially reconfigured and the new settings must be applied before it can be paused or restarted.

3.71   psospaSigGenSoftwareTriggerControl - set software triggering

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

This function causes the signal generator trigger to fire, if a software trigger has been set up using psospaSigGenTrigger() and the signal generator is waiting for a trigger event.

If the trigger type set using psospaSigGenTrigger() is PICO_SIGGEN_RISING or PICO_SIGGEN_FALLING, calling this function will trigger the defined number of waveform cycles or sweeps and the `triggerState` parameter is not used.

If the trigger type set using psospaSigGenTrigger() is PICO_SIGGEN_GATE_HIGH or PICO_SIGGEN_GATE_LOW, calling this function will start the signal generator running when `triggerState = PICO_SIGGEN_GATE_HIGH`, or pause it when any other value.

Applicability
All modes

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

triggerState,   when the trigger type is set to `gating`, runs the signal generator if `triggerState = PICO_SIGGEN_GATE_HIGH` or pauses it otherwise.

Returns
PICO_OK

PICO_SIGGEN_TRIGGER_SOURCE

3.72　psospaSigGenTrigger - choose the trigger event

```
PICO_STATUS psospaSigGenTrigger
(
  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.

After configuring all required signal generator settings, call psospaSigGenApply() to apply them to the device.

Applicability

All modes

Arguments

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

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

```
PICO_SIGGEN_RISING = 0,
PICO_SIGGEN_FALLING = 1,
PICO_SIGGEN_GATE_HIGH = 2,
PICO_SIGGEN_GATE_LOW = 3
```

`triggerSource`, the signal used as a trigger:

```
PICO_SIGGEN_NONE = 0,
PICO_SIGGEN_SCOPE_TRIG = 1,
PICO_SIGGEN_AUX_IN = 2,
PICO_SIGGEN_SOFT_TRIG = 4,
```

`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`, reserved for future use, set to zero.

Returns

`PICO_OK`
`PICO_SIGGEN_TRIGGERTYPE_NOT_SUPPORTED`, this indicates that the provided trigger type is not supported.
`PICO_SIGGEN_TRIGGERSOURCE_NOT_SUPPORTED`, indicates that the provided trigger source is not supported.
`PICO_SIGGEN_CYCLES_OUT_OF_RANGE`, this indicates that the provided cycles value exceeds the maximum allowed value.

3.73 psospaSigGenWaveform - choose signal generator waveform

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

This function specifies which waveform the signal generator will produce. After configuring all required signal generator settings, call psospaSigGenApply() to apply them to the device.

Applicability

All modes

Arguments

handle, the device identifier returned by psospaOpenUnit().

waveType, specifies the type of waveform to generate, for example PICO_SINE

buffer, an array of sample values to be used by the arbitrary waveform generator (AWG). Used only when waveType = PICO_ARBITRARY. Each sample value should be in the range (−32767 to 32767) as returned by psospaSigGenLimits(), representing the full output voltage span of the waveform generator.

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

Returns

PICO_OK
PICO_SIGGEN_WAVETYPE_NOT_SUPPORTED, indicates that the provided wave type is not supported.
PICO_SIGGEN_BUFFER_NOT_SUPPLIED, indicates that the buffer is nullptr when the wave type is PICO_ARBITRARY.
PICO_SIGGEN_EMPTY_BUFFER_SUPPLIED, indicates that the buffer length is zero when the wave type is PICO_ARBITRARY.
PICO_SIGGEN_TOO_MANY_SAMPLES, indicates that the buffer length exceeds the maximum allowed samples.

3.74   psospaSigGenWaveformDutyCycle - set duty cycle

```
PICO_STATUS psospaSigGenWaveformDutyCycle
(
  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. The default duty cycle is 50% (representing a square wave with equal high and low times, or a triangle wave with equal rise and fall times) and it is only necessary to call this function if a different duty cycle is required.

After configuring all required signal generator settings, call psospaSigGenApply() to apply them to the device.

Applicability

Square wave and triangle wave outputs only.

Arguments

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

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

Returns

`PICO_OK`
`PICO_SIGGEN_DUTYCYCLE_OUT_OF_RANGE`, indicates that the provided duty cycle percentage is out of the valid range (`0.0` to `100.0`)

3.75   psospaStartFirmwareUpdate - update the device firmware

```
PICO_STATUS psospaStartFirmwareUpdate
(
  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.

The function applies any firmware update to the device which is included in the current driver. It does not check online for updates or require internet access.

Applicability

All modes

Arguments

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

`progress,` a user-supplied function that receives callbacks when the status of the update changes. See `PicoUpdateFirmwareProgress()`. May be NULL if not required.

Returns

`PICO_FIRMWARE_UP_TO_DATE,` the firmware update was performed successfully or firmware was already up to date

3.76   psospaStop - stop sampling

```
PICO_STATUS psospaStop
(
  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 `psospaOpenUnit()`.

Returns
`PICO_OK`

3.77   psospaStopUsingGetValuesOverlapped - complements psospaGetValuesOverlapped

<u>PICO_STATUS</u> psospaStopUsingGetValuesOverlapped
(
  int16_t   handle
)

This function stops deferred data-collection that was started by calling <u>psospaGetValuesOverlapped()</u>.

Applicability
Block and rapid block mode

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

Returns
PICO_OK

3.78   psospaTriggerWithinPreTriggerSamples - switch feature on or off

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

When this feature is enabled, the scope will trigger if a trigger event is detected during the pre-trigger samples. Effectively, the user-specified pre-trigger count becomes a maximum pre-trigger count and the actual number of pre-trigger samples returned will be between zero and that number depending on when the trigger occurs. You can find the actual trigger point by calling psospaGetTriggerInfo() after the capture has completed.

Applicability

All modes

Arguments

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

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

Returns

`PICO_OK`
`PICO_INVALID_TRIGGER_WITHIN_PRE_TRIGGER_STATE,` indicates that the provided state is invalid.

4   Callbacks

4.1   psospaBlockReady - indicate when block-mode data ready

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

This callback function is part of your application. You register it with the PicoScope 3000E Series driver using psospaRunBlock() and the driver calls it back when block-mode data is ready. You can then download the data using the psospaGetValues() function.

Applicability

Block mode only

Arguments

handle,   the device identifier returned by psospaOpenUnit().

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

pParameter,   a pointer passed from psospaRunBlock(). 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   psospaDataReady - indicate when post-collection data is ready

```
typedef void(*psospaDataReady)
(
  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 psospaGetValuesAsync() and the driver calls your function back when the data is ready.

Applicability
All modes

Arguments

handle, the device identifier returned by psospaOpenUnit().

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 psospaGetValuesAsync(). 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 (* PicoUpdateFirmwareProgress)
(
  int16_t        handle,
  uint16_t       progress
)
```

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

Applicability
All modes

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

`progress,` a progress indicator.

Returns
nothing

5   Reference

5.1   Numeric data types

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

Type	Bits	Signed or unsigned?
`int8_t`	8	signed
`uint8_t`	8	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 3000E Series API driver are defined in header files 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 `psospa` 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 3000E Series.

In addition to the function-specific error codes described in this guide, functions may also return a generic error code such as one of the following:

`PICO_INVALID_HANDLE,` the handle passed does not refer to an open PicoScope unit.

`PICO_MEMORY_FAIL,` could not allocate sufficient memory on the host PC to complete the operation.

`PICO_NOT_RESPONDING,` the PicoScope did not respond to a command, for example if it has been disconnected.

`PICO_INTERNAL_ERROR,` an unexpected error has occurred in the driver. Contact [Pico technical support](#) for assistance.

`PICO_DRIVER_FUNCTION,` a driver function has already been called and not yet finished. Only one call to the driver can be made at any one time.

5.4 Glossary

Callback. A mechanism that the PicoScope 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 3000E Series oscilloscopes is supplied in the form of 64-bit Windows DLLs called `psospa.dll` and macOS and Linux libraries called `libpsospa.` These are used by your application to control the oscilloscope.

PicoScope 3000E Series. A range of PC Oscilloscopes from Pico Technology, with bandwidths up to 500 MHz, maximum sampling rate of up to 5 GS/s, sampling resolutions of 8/10 bits and a capture memory size of 2 GS.

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.

UK headquarters:

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

Tel: +44 (0) 1480 396 395

sales@picotech.com
support@picotech.com

US regional office:

Pico Technology
320 N Glenwood Blvd
Tyler
TX 75702
USA

Tel: +1 800 591 2796

sales@picotech.com
support@picotech.com

Asia-Pacific regional office:

pico.asia-pacific@picotech.com

Germany regional office and EU Authorised Representative:

Pico Technology GmbH
Emmericher Str. 60
47533 Kleve
Germany

Tel: +49 (0) 5131 907 62 90

info.de@picotech.com

www.picotech.com