# Contents

1 Welcome ................................................................................................................................. 1

2 Introduction ............................................................................................................................. 2
   1 License agreement .................................................................................................................. 2
   2 Trademarks .......................................................................................................................... 2
   3 System requirements ............................................................................................................. 3

3 Programming with the PicoScope 5000 Series (A API) ....................................................... 4
   1 Driver .................................................................................................................................... 4
   2 Voltage ranges ...................................................................................................................... 5
   3 MSO digital data ..................................................................................................................... 6
   4 Triggering .............................................................................................................................. 7
   5 Sampling modes .................................................................................................................... 8
      1 Block mode ....................................................................................................................... 9
      2 Rapid block mode ............................................................................................................. 12
      3 ETS (Equivalent Time Sampling) .................................................................................... 18
      4 Streaming mode ............................................................................................................... 20
   5 Retrieving stored data .......................................................................................................... 21
   6 Timebases ............................................................................................................................ 22
   7 Power options ...................................................................................................................... 24
   8 Combining several oscilloscopes .......................................................................................... 25

4 API functions .......................................................................................................................... 26
   1 ps5000aChangePowerSource – select USB or AC adaptor power ........................................ 26
   2 ps5000aChannelCombinationsStateless – find out which channels can be used .................. 27
      1 PS5000A_CHANNEL_FLAGS enumerated type ............................................................... 28
   3 ps5000aCloseUnit – close a scope device .......................................................................... 29
   4 ps5000aCurrentPowerSource – indicate the current power state of the device .................. 30
   5 ps5000aEnumerateUnits – find all connected oscilloscopes ............................................... 31
   6 ps5000aFlashLed – flash the front-panel LED ................................................................... 32
   7 ps5000aGetAnalogueOffset – query the permitted analog offset range ............................... 33
      1 PS5000A_RANGE enumerated type ................................................................................. 34
      2 PS5000A_COUPLING enumerated type ....................................................................... 34
   8 ps5000aGetChannelInformation – query which ranges are available on a device ............... 35
      1 PS5000A_CHANNEL_INFO enumerated type ............................................................... 35
   9 ps5000aGetDeviceResolution – retrieve the resolution the device will run ......................... 36
  10 ps5000aGetMaxDownSampleRatio – query the aggregation ratio for data ......................... 37
  11 ps5000aGetMaxSegments – query the maximum number of segments .............................. 38
  12 ps5000aGetMinimumTimebaseStateless – find fastest available timebase ......................... 39
  13 ps5000aGetNoOfCaptures – find out how many captures are available ............................. 40
  14 ps5000aGetNoOfProcessedCaptures – query number of captures processed .................. 41
  15 ps5000aGetStreamingLatestValues – get streaming data while scope is running ............... 42
  16 ps5000aGetTimebase – get properties of the selected timebase ........................................ 43
ps5000aGetTimebase – get properties of the selected timebase ........................................ 44
ps5000aGetTriggerInfoBulk – get trigger timestamps ......................................................... 45
  1 PS5000A_TRIGGER_INFO structure ............................................................................. 46
  2 Time stamping ............................................................................................................. 46
  3 PS5000A_TIME_UNITS enumerated type ...................................................................... 48
ps5000aGetTriggerTimeOffset – find out when trigger occurred (32-bit) .................................. 49
ps5000aGetTriggerTimeOffset64 – find out when trigger occurred (64-bit) .......................... 50
ps5000aGetUnitInfo – read information about scope device ............................................. 51
ps5000aGetValues – retrieve block-mode data with callback ........................................... 52
  1 PS5000A_RATIO_MODE enumerated type ...................................................................... 53
ps5000aGetValuesAsync – retrieve streaming data with callback ....................................... 54
ps5000aGetValuesBulk – retrieve data in rapid block mode ............................................. 55
ps5000aGetValuesOverlapped – set up data collection ahead of capture .............................. 56
  1 Using the GetValuesOverlapped functions .................................................................... 56
ps5000aGetValuesOverlappedBulk – set up data collection in rapid block mode .......... 58
ps5000aGetValuesTriggerTimeOffsetBulk – get rapid-block waveform timings (32-bit) ..... 59
ps5000aGetValuesTriggerTimeOffsetBulk64 – get rapid–block waveform timings (64-bit) ... 60
ps5000aIsLedFlashing – check LED status ....................................................................... 61
ps5000aOpenUnit – open a scope device .......................................................................... 62
ps5000aOpenUnitAsync – open a scope device without blocking ......................................... 70
ps5000aOpenUnitProgress – check progress of OpenUnit call ......................................... 71
ps5000aPingUnit – check communication with device ....................................................... 72
ps5000aQueryOutputEdgeDetect – check if output edge detection is enabled ................... 73
ps5000aRunBlock – start block mode ................................................................................. 74
ps5000aRunStreaming – start streaming mode .................................................................... 75
ps5000aSetAutotriggerMicroSeconds – set auto-trigger timeout ...................................... 76
ps5000aSetBandwidthFilter – specifies the bandwidth limit ................................................ 77
  1 PS5000A_BANDWIDTH_LIMITER enumerated type ....................................................... 77
ps5000aSetChannel – set up input channels ...................................................................... 78
  1 PS5000A_CHANNEL enumerated type ........................................................................... 79
ps5000aSetDataBuffer – register data buffer with driver .................................................... 80
ps5000aSetDataBuffers – register aggregated data buffers with driver ............................ 81
ps5000aSetDeviceResolution – set the hardware resolution ............................................ 82
  1 PS5000A_DEVICE_RESOLUTION enumerated type .......................................................... 82
ps5000aSetDigitalPort – set up digital inputs ..................................................................... 84
  1 MSO digital connector .................................................................................................. 84
ps5000aSetEt – set up equivalent-time sampling .............................................................. 85
1 PS5000A_ETS_MODE enumerated type .................................................. 86
52 ps5000aSetEtsTimeBuffer – set up buffer for ETS timings (64-bit) ......................... 87
53 ps5000aSetEtsTimeBuffers – set up buffer for ETS timings (32-bit) ....................... 88
54 ps5000aSetNoOfCaptures – set number of captures to collect in one run .................. 89
55 ps5000aSetOutputEdgeDetect – change triggering behavior .................................. 90
56 ps5000aSetPulseWidthDigitalPortProperties – set digital port pulse width ................. 91
57 ps5000aSetPulseWidthQualifier – set up pulse width triggering .............................. 92
1 PS5000A_PWQ_CONDITIONS structure ................................................... 93
58 ps5000aSetPulseWidthQualifierConditions – set up pulse width triggering ............... 94
1 PS5000A_CONDITIONS_INFO enumerated type .......................................... 94
59 ps5000aSetPulseWidthQualifierDirections – set up pulse width triggering ............... 96
60 ps5000aSetPulseWidthQualifierProperties – set up pulse width triggering ............. 97
1 PS5000A_PULSE_WIDTH_TYPE enumerated type .......................................... 97
61 ps5000aSetSigGenArbitrary – set up arbitrary waveform generator ......................... 98
1 PS5000A_INDEX_MODE enumerated type .................................................. 99
2 Calculating deltaPhase ............................................................................. 100
3 PS5000A_SWEEP_TYPE enumerated type .................................................. 101
4 PS5000A_EXTRA_OPERATIONS enumerated type ............................................ 101
62 ps5000aSetSigGenBuiltIn – set up standard signal generator ................................. 102
1 PS5000A_SIGGEN_TRIG_TYPE enumerated type ........................................... 103
2 PS5000A_SIGGEN_TRIG_SOURCE enumerated type ........................................ 104
3 PS5000A_WAVE_TYPE enumerated type ................................................... 104
63 ps5000aSetSigGenBuiltInV2 – high-precision signal generator setup ....................... 106
64 ps5000aSetSigGenPropertiesArbitrary – change AWG settings ............................. 107
65 ps5000aSetSigGenPropertiesBuiltIn – change function generator settings ............... 108
66 ps5000aSetSimpleTrigger – set edge or level trigger ....................................... 109
67 ps5000aSetTriggerChannelConditions – specify which channels to trigger on ........... 110
1 PS5000A_TRIGGER_CONDITIONS structure ............................................. 110
68 ps5000aSetTriggerChannelConditionsV2 – specify which channels to trigger on ...... 112
1 PS5000ACONDITION structure .............................................................. 112
2 PS5000A_TRIGGER_STATE enumerated type .............................................. 113
69 ps5000aSetTriggerChannelDirections – set up signal polarities for triggering .......... 114
1 PS5000A_THRESHOLD_DIRECTION enumerated type ..................................... 115
70 ps5000aSetTriggerChannelDirectionsV2 – set up signal polarities for triggering ...... 116
1 PS5000A_DIRECTION structure .............................................................. 116
2 PS5000A_THRESHOLD_MODE enumerated type ............................................ 117
71 ps5000aSetTriggerChannelProperties – set up trigger thresholds ......................... 118
1 PS5000A_TRIGGER_CHANNEL_PROPERTIES structure ................................. 119
72 ps5000aSetTriggerChannelPropertiesV2 – set up trigger thresholds ..................... 120
1 PS5000A_TRIGGER_CHANNEL_PROPERTIES_V2 structure ............................. 120
2 Hysteresis ............................................................................................... 121
73 ps5000aSetTriggerDelay – set up post-trigger delay ........................................... 122
74 ps5000aSetTriggerDigitalPortProperties – set up digital inputs for triggering .......... 123
1 PS5000A_DIGITAL_CHANNEL_DIRECTIONS structure .................................. 124
| 2 | PS5000A_DIGITAL_CHANNEL enumerated type ................................................................. 124 |
| 3 | PS5000A_DIGITAL_DIRECTION enumerated type ............................................................... 125 |
| 75 | ps5000aSigGenArbitraryMinMaxValues – get AWG parameters ........................................ 126 |
| 76 | ps5000aSigGenFrequencyToPhase – convert frequency to phase count ............................ 127 |
| 77 | ps5000aSigGenSoftwareControl – trigger the signal generator ....................................... 128 |
| 78 | ps5000aStop – stop data capture ................................................................................ 129 |
| 79 | ps5000aTriggerWithinPreTriggerSamples – change triggering behavior ......................... 130 |
| 80 | 1 | PS5000A_TRIGGER_WITHIN_PRE_TRIGGER enumerated type ............................................. 130 |
| 81 | Callback functions ..................................................................................................... 131 |
| 82 | 1 | ps5000aBlockReady – indicate when block-mode data ready ........................................... 132 |
| 83 | 2 | ps5000aDataReady – indicate when post-collection data ready ........................................ 133 |
| 84 | 3 | ps5000aStreamingReady – indicate when streaming-mode data ready ............................ 134 |
| 85 | Wrapper functions ..................................................................................................... 135 |
| 86 | 5 | Reference ..................................................................................................................... 137 |
| 87 | 1 | Driver status codes ...................................................................................................... 137 |
| 88 | 2 | Enumerated types and constants .................................................................................. 137 |
| 89 | 3 | Numeric data types ....................................................................................................... 137 |
| 90 | 4 | Glossary ....................................................................................................................... 138 |
| 91 | Index ................................................................................................................................ 141 |
1 Welcome

The PicoScope 5000A, 5000B and 5000D Series PC Oscilloscopes from Pico Technology are a range of high-specification, real-time measuring instruments that connect to the USB port of your computer. They offer various combinations of portability, deep memory, fast sampling rates and high bandwidth to suit a wide range of applications. The range includes hi-speed USB 2.0 and SuperSpeed USB 3.0 devices.

This manual explains how to use the API (application programming interface) functions, so that you can develop your own programs to collect and analyze data from the oscilloscope.

The information in this manual applies to the following oscilloscopes:

- **PicoScope 5000A Series**
  The A models are high-speed portable USB 2.0 oscilloscopes with a function generator.

- **PicoScope 5000B Series**
  The B models have all the features of the A models with the addition of an arbitrary waveform generator (AWG) and deeper memory.

- **PicoScope 5000D Series**
  The D models are USB 3.0-connected and include an AWG. The D MSO models have mixed-signal (analog and digital) inputs.

**Related products**

The PicoScope 5203 and 5204 oscilloscopes use the ps5000 API, which comes with its own *Programmer’s Guide*. For information on any PicoScope 5000 Series oscilloscope, refer to the documentation on [www.picotech.com](http://www.picotech.com).
Introduction

2.1 License agreement

Grant of license. The material contained in this release is licensed, not sold. Pico Technology Limited (‘Pico’) grants a license to the person who installs this software, subject to the conditions listed below.

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

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

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

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

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

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

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

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

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

2.2 Trademarks

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

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

Windows, Excel and Visual Basic for Applications are registered trademarks or trademarks of Microsoft Corporation in the USA and other countries. LabVIEW is a registered trademark of National Instruments Corporation. MATLAB is a registered trademark of The MathWorks, Inc.
2.3 System requirements

Using the ps5000a API
To ensure that your PicoScope 5000 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 multicore processor.

<table>
<thead>
<tr>
<th>Item</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating system</td>
<td>Windows 7, 8 or 10, 32-bit and 64-bit versions. Linux and macOS, 64-bit versions only: see picotech.com for supported versions.</td>
</tr>
<tr>
<td>Processor, memory, free disk space</td>
<td>As required by the operating system</td>
</tr>
<tr>
<td>Ports</td>
<td>USB 2.0 or USB 3.0 port</td>
</tr>
</tbody>
</table>

**USB**
The ps5000a API offers four different methods of recording data, all of which support USB 2.0 and USB 3.0 connections.

The 5000A and 5000B Series oscilloscopes are all hi-speed USB 2.0 devices. They are compatible with USB 3.0 but will run at USB 2.0 speeds when connected to a USB 3.0 port.

The 5000D Series oscilloscopes are SuperSpeed USB 3.0 devices. They are compatible with USB 2.0 but will run at USB 2.0 speeds when connected to a USB 2.0 port.
3 Programming with the PicoScope 5000 Series (A API)

PicoSDK allows you to program a PicoScope 5000 Series (A API) oscilloscope using standard function calls.

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

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

The 'picotech' pages on GitHub contain links to programming examples in various languages and development environments.

3.1 Driver

**Microsoft Windows**
Your application will communicate with a PicoScope 5000 Series library called ps5000a.dll, which is supplied in 32-bit and 64-bit versions. This DLL is compatible with the 5000A, 5000B and 5000D Series oscilloscopes. The DLL exports the ps5000a **function definitions** in stdcall format, which is compatible with a wide range of programming languages.

ps5000a.dll depends on another DLL, picoipp.dll (which is supplied in 32-bit and 64-bit versions) and a low-level driver called WinUsb.sys (or CyUsb3.sys on Windows 7). These are installed by the SDK and configured when you plug the oscilloscope into each USB port for the first time. Your application does not call these drivers directly.

**Linux and Apple macOS**
Please see the **Downloads** section of picotech.com for instructions on downloading the drivers for these operating systems. The drivers use the cdecl calling convention. Linux libraries and dependencies are distributed via our package repositories. macOS libraries and dependencies are distributed with PicoScope 6 for macOS.
3.2 Voltage ranges

You can set a device input channel to any voltage range from ±10 mV to ±20 V with the `ps5000aSetChannel` function. Each sample is scaled to 16 bits, and the minimum and maximum values returned to your application are given by `ps5000aMinimumValue` and `ps5000aMaximumValue` as follows:

<table>
<thead>
<tr>
<th>Function</th>
<th>Voltage</th>
<th>Value returned</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>decimal</td>
<td>hex</td>
</tr>
<tr>
<td>8-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>ps5000aMaximumValue</code></td>
<td>maximum</td>
<td>+32 512</td>
</tr>
<tr>
<td></td>
<td>zero</td>
<td>0</td>
</tr>
<tr>
<td><code>ps5000aMinimumValue</code></td>
<td>minimum</td>
<td>−32 512</td>
</tr>
<tr>
<td>12, 14, 15 and 16-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>ps5000aMaximumValue</code></td>
<td>maximum</td>
<td>+32 767</td>
</tr>
<tr>
<td></td>
<td>zero</td>
<td>0</td>
</tr>
<tr>
<td><code>ps5000aMinimumValue</code></td>
<td>minimum</td>
<td>−32 767</td>
</tr>
</tbody>
</table>

**Example at 8-bit resolution**

1. Call `ps5000aSetChannel` with range set to PS5000A_1V.
2. Apply a sine wave input of 500 mV amplitude to the oscilloscope.
3. Capture some data using the desired sampling mode.
4. The data will be encoded as shown opposite.

**External trigger input**

The external trigger input (marked Ext), where available, is scaled to a 16-bit value as follows:

<table>
<thead>
<tr>
<th>Voltage</th>
<th>Constant</th>
<th>Digital value</th>
</tr>
</thead>
<tbody>
<tr>
<td>−5 V</td>
<td>PS5000A_EXT_MIN_VALUE</td>
<td>−32 767</td>
</tr>
<tr>
<td>0 V</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>+5 V</td>
<td>PS5000A_EXT_MAX_VALUE</td>
<td>+32 767</td>
</tr>
</tbody>
</table>

**Digital inputs (MSO devices only)**

See `ps5000aSetDigitalPort`. 
3.3 MSO digital data

Applicability
Mixed-signal oscilloscope (MSO) devices only

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

Use the \texttt{ps5000aSetDataBuffer} and \texttt{ps5000aSetDataBuffers} 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 while the upper 8 bits of the word are undefined.

<table>
<thead>
<tr>
<th>Sample$_0$</th>
<th>PORT1 buffer</th>
<th>PORT0 buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[XXXXXXXXX,D15...D8]$_0$</td>
<td>[XXXXXXXXX,D7...D0]$_0$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Sample$_{n-1}$</td>
<td>[XXXXXXXXX,D15...D8]$_{n-1}$</td>
<td>[XXXXXXXXX,D7...D0]$_{n-1}$</td>
</tr>
</tbody>
</table>

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.

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

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

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

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

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

PicoScope 5000 Series oscilloscopes can either start collecting data immediately, or be programmed to wait for a trigger event to occur. In either case, call the function:

- `ps5000aSetSimpleTrigger`

For more complex trigger setups such as pulse width triggering, call the lower-level trigger functions:

- `ps5000aSetTriggerChannelPropertiesV2`
- `ps5000aSetTriggerChannelConditionsV2`
- `ps5000aSetTriggerChannelDirectionsV2`

To set up triggers on the digital inputs, use this additional function:

- `ps5000aSetTriggerDigitalPortProperties`

A trigger event can occur when one of the signal or trigger input channels crosses a threshold voltage on either a rising or a falling edge, or when a more complex time-qualified condition occurs. It is also possible to combine multiple analog and digital inputs and time-qualified conditions using the logic trigger function.

The driver supports these triggering methods:

- Simple edge (rising or falling with fixed hysteresis)
- Advanced edge (rising or falling with adjustable hysteresis)
- Windowed (entering or leaving a voltage range)
- Pulse width
- Logic (a Boolean function of multiple inputs)
- Delay (wait after trigger and then capture)
- Drop-out (no trigger within a specified time)
- Runt (pulse height between two thresholds)
- Digital (a function of digital inputs; MSO devices only)

The pulse width, delay and drop-out triggering methods additionally require the use of the pulse width qualifier functions:

- `ps5000aSetPulseWidthQualifierProperties`
- `ps5000aSetPulseWidthQualifierConditions`
- `ps5000aSetPulseWidthQualifierDirections`
### 3.5 Sampling modes

PicoScope 5000 Series oscilloscopes can capture data using various sampling modes:

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

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

- **Rapid block mode.** This is a variant of block mode that allows you to capture more than one waveform at a time with a minimum of delay between captures.

- **Streaming mode.** In this mode, data is passed directly to the PC without being limited by the size of the scope’s capture memory. This enables long periods of data collection. Streaming mode supports downsampling and triggering. Maximum data rates are listed in the data sheet for your oscilloscope.

In all sampling modes, the driver writes data to the application's buffers asynchronously and then notifies you using a callback. This is a call to one of the functions in your own application. When you request data from the scope, you pass to the driver a pointer to your callback function. The callback function then checks whether the capture completed successfully or resulted in an error.

The callback will be called asynchronously in its own thread and therefore you must ensure that it is thread-safe.

For compatibility with programming environments not supporting C callback functions, polling of the driver is available in block mode. We also supply a wrapper for streaming mode.

*Note: The oversampling feature of older PicoScope oscilloscopes has been replaced by PS5000A_RATIO_MODE_AVERAGE.*
3.5.1 Block mode

In **block mode**, the computer prompts a PicoScope 5000 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 [ps5000aMemorySegments](#)).

- **Sampling rate.** A PicoScope 5000 Series oscilloscope can sample at a number of different rates according to the selected [timebase](#) and resolution. In turn, the available timebases may depend on the combination of channels enabled. See the [PicoScope 5000 Series User’s Guide](#) for the specifications that apply to your scope model. You can call [ps5000aGetMinimumTimebaseStateless](#) to find the fastest available timebase.

- **Setup time.** The driver normally performs a number of setup operations, which can take up to 50 milliseconds, before collecting each block of data. If you need to collect data with the minimum time interval between blocks, use [rapid block mode](#) and avoid calling setup functions between calls to [ps5000aRunBlock](#), [ps5000aStop](#) and [ps5000aGetValues](#).

- **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.

- **Segmented memory.** The scope’s internal memory can be divided into segments so that you can capture several waveforms in succession. Configure this using [ps5000aMemorySegments](#).

- **Data retention.** The data is lost when a new run is started in the same segment, the settings are changed, the resolution is changed, or the scope is powered down or (for flexible power devices) the power source is changed.

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

You can use **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: see **rapid block mode example 2** for an example of this.

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

1. Open the oscilloscope using `ps5000aOpenUnit`.
2. Select channel ranges and AC/DC coupling using `ps5000aSetChannel`.
2a. Set the digital port using `ps5000aSetDigitalPort` (mixed-signal scopes only).
3. Using `ps5000aGetTimebase`, select timebases until the required nanoseconds per sample is located.
4. Use the trigger setup function `ps5000aSetSimpleTrigger` to set up the trigger if required.
4a. Use the trigger setup functions `ps5000aSetTriggerDigitalPortProperties` and `ps5000aSetTriggerChannelConditionsV2` to set up the digital trigger if required (mixed-signal scopes only).
5. Start the oscilloscope running using `ps5000aRunBlock`.
6. Wait until the oscilloscope is ready using the `ps5000aBlockReady` callback (or poll using `ps5000aIsReady`).
7. Use `ps5000aSetDataBuffer` to tell the driver where your memory buffer is. For greater efficiency when doing multiple captures, you can call this function outside the loop, after step 4.
8. Transfer the block of data from the oscilloscope using `ps5000aGetValues`.
9. Display the data.
10. Repeat steps 5 to 9.
11. Stop the oscilloscope using `ps5000aStop`.
12. Request new views of stored data using different downsampling parameters: see **Retrieving stored data**.
13. Close the device using `ps5000aCloseUnit`.

Note that if you use `ps5000aGetValues` or `ps5000aStop` before the oscilloscope is ready, no capture will be available. In this case `ps5000aGetValues` would return **PICO_NO_SAMPLES_AVAILABLE**.
3.5.1.2 Asynchronous data retrieval

The `ps5000aGetValues` function may take a long time to complete if a large amount of data is being collected. For example, it can take 14 seconds (or several minutes on USB 1.1) to retrieve the full 512 megasamples (in 8-bit mode) from the higher-capacity PicoScope 5000 Series models using a USB 2.0 connection. To avoid hanging the calling thread, it is possible to call `ps5000aGetValuesAsync` instead. This immediately returns control to the calling thread, which then has the option of waiting for the data or calling `ps5000aStop` to abort the operation.
3.5.2 Rapid block mode

In normal block mode, the PicoScope 5000 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 2 microseconds (on fastest timebase).

See Using rapid block mode for details.

3.5.2.1 Using rapid block mode

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

Without aggregation

1. Open the oscilloscope using `ps5000aOpenUnit`.
2. Select channel ranges and AC/DC coupling using `ps5000aSetChannel`.
2a. Set the digital port using `ps5000aSetDigitalPort` (mixed-signal scopes only).
3. Set the number of memory segments equal to or greater than the number of captures required using `ps5000aMemorySegments`. Use `ps5000aSetNoOfCaptures` before each run to specify the number of waveforms to capture.
4. Using `ps5000aGetTimebase`, select timebases until the required nanoseconds per sample is located. This will indicate the number of samples per channel available for each segment.
5. Use the trigger setup function `ps5000aSetSimpleTrigger` to set up the trigger if required.
6. Start the oscilloscope running using `ps5000aRunBlock`.
   THEN EITHER
7a. To obtain data before rapid block capture has finished, call `ps5000aStop` and then `ps5000aGetNoOfCaptures` to find out how many captures were completed.
    OR
7b. Wait until the oscilloscope is ready using `ps5000aIsReady`.
    OR
7c. Wait on the callback function.
8. Use `ps5000aSetDataBuffer` to tell the driver where your memory buffers are. Call the function once for each channel/segment combination for which you require data. For greater efficiency when doing multiple captures, you can call this function outside the loop, after step 5.
9. Transfer the blocks of data from the oscilloscope using `ps5000aGetValuesBulk` (or `ps5000aGetValues` to retrieve one buffer at a time). These functions stop the oscilloscope.
10. Retrieve the time offset for each data segment using `ps5000aGetValuesTriggerTimeOffsetBulk64`.
10a. Optionally retrieve trigger time stamps using `ps5000aGetTriggerInfoBulk`.
11. Display the data.
12. Repeat steps 6 to 11 if necessary.
13. Call `ps5000aStop` (usually unnecessary as the scope stops automatically in most cases, but recommended as a precaution).
14. Close the device using `ps5000aCloseUnit`.
**With aggregation**
To use rapid block mode with aggregation, follow steps 1 to 7 above, then proceed as follows:

8a. Call `ps5000aSetDataBuffer` or `ps5000aSetDataBuffers` to set up one pair of buffers for every waveform segment required.

9a. Call `ps5000aGetValuesBulk` for each pair of buffers.

10a. Retrieve the time offset for each data segment using `ps5000aGetValuesTriggerTimeOffsetBulk64`.

10b. Optionally retrieve trigger time stamps using `ps5000aGetTriggerInfoBulk`.

Continue from step 11 above.
3.5.2.2 Rapid block mode example 1: no aggregation

#define MAX_WAVEFORMS 100
#define MAX_SAMPLES 1000

Set up the device up as usual.

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

    // Set the number of waveforms to MAX_WAVEFORMS
    ps5000aSetNoOfCaptures (handle, MAX_WAVEFORMS);

    pParameter = false;
    ps5000aRunBlock
    (handle,
    0,         // noOfPreTriggerSamples
    10000,     // noOfPostTriggerSamples
    1,         // timebase to be used
    &timeIndisposedMs,
    0,         // segment index
    lpReady,
    &pParameter
    );

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

    while (!pParameter) Sleep (0);

    int16_t buffer[PS5000A_MAX_CHANNELS][MAX_WAVEFORMS][MAX_SAMPLES];

    for (int32_t i = 0; i < 20; i++)
    {
        for (int32_t c = PS5000A_CHANNEL_A; c <= PS5000A_CHANNEL_B; c++)
        {
            ps5000aSetDataBuffer
            (handle,
            c,
            buffer[c][i],
            MAX_SAMPLES,
            1
            PS5000A_RATIO_MODE_NONE
            );
        }
    }
Comments: buffer has been created as a three-dimensional 16-bit integer array, which will contain 1000 samples as defined by MAX_SAMPLES. There are only 20 buffers set, but it is possible to set up to the number of captures you have requested. PS5000A_RATIO_MODE_NONE can be substituted for PS5000A_RATIO_MODE_AGGREGATE, PS5000A_RATIO_MODE_DECIMATE, or PS5000A_RATIO_MODE_AVERAGE.

```c
int16_t overflow[MAX_WAVEFORMS];

ps5000aGetValuesBulk
(  handle,  ndOfSamples,           // set to MAX_SAMPLES on entering the function
  10,      // fromSegmentIndex
  19,      // toSegmentIndex
  1,       // downsampling ratio
  PS5000A_RATIO_MODE_NONE, // downsampling ratio mode
  overflow  // indices 10 to 19 will be populated
)
```

Comments: the number of samples could be up to noOfPreTriggerSamples + noOfPostTriggerSamples, the values set in ps5000aRunBlock. The samples are always returned from the first sample taken, unlike the ps5000aGetValues function which allows the sample index to be set. The above segments start at 10 and finish at 19 inclusive. It is possible for the fromSegmentIndex to wrap around to the toSegmentIndex, by setting the fromSegmentIndex to 98 and the toSegmentIndex to 7.

```c
int64_t times[MAX_WAVEFORMS];
PS5000A_TIME_UNITS timeUnits[MAX_WAVEFORMS];

ps5000aGetValuesTriggerTimeOffsetBulk64
(  handle,  times,           // indices 10 to 19 will be populated
  timeUnits,       // indices 10 to 19 will be populated
  10,             // fromSegmentIndex, inclusive
  19,             // toSegmentIndex, inclusive
)
```

Comments: the above segments start at 10 and finish at 19 inclusive. It is possible for the fromSegmentIndex to wrap around to the toSegmentIndex, if the fromSegmentIndex is set to 98 and the toSegmentIndex to 7.
3.5.2.3 Rapid block mode example 2: using aggregation

#define MAX_WAVEFORMS 100
#define MAX_SAMPLES 1000

Set up the device up as usual.

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

// Set the number of waveforms to MAX_WAVEFORMS
ps5000aSetNoOfCaptures (handle, MAX_WAVEFORMS);

pParameter = false;
ps5000aRunBlock
(handle, 0, noOfPreTriggerSamples, 1000000, noOfPostTriggerSamples, 1, timebase to be used, &timeIndisposedMs, lpReady, &pParameter);

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

for (int32_t segment = 10; segment < 20; segment++)
{
    for (int32_t c = PS5000A_CHANNEL_A; c <= PS5000A_CHANNEL_D; c++)
    {
        ps5000aSetDataBuffers
        (handle, c, bufferMax[c], bufferMin[c], MAX_SAMPLES 1, PS5000A_RATIO_MODE_AGGREGATE);
    }
}

ps5000aGetValues
(handle, 0, noOfSamples, // set to MAX_SAMPLES on entering 1000, downSampleRatioMode, // set to RATIO_MODE_AGGREGATE index,
overflow

ps5000aGetTriggerTimeOffset64
{
    handle,
    &time,
    &timeUnits,
    index
}

Each waveform is retrieved one at a time from the driver, with an aggregation of 1000. Since only one waveform is 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 greater efficiency you can use ps5000aGetValuesBulk to retrieve the values in one go.
3.5.3 ETS (Equivalent Time Sampling)

ETS is a way of increasing the effective sampling rate of the scope when capturing repetitive signals. It is a modified form of block mode, and is controlled by the trigger functions and the ps5000aSetEts function.

- **Overview.** ETS works by capturing several cycles of a repetitive waveform, then combining them to produce a composite waveform that has a higher effective sampling rate than the individual captures. The scope hardware accurately measures the delay, which is a small fraction of a single sampling interval, between each trigger event and the subsequent sample. The driver then shifts each capture slightly in time and overlays them so that the trigger points are exactly lined up. The result is a larger set of samples spaced by a small fraction of the original sampling interval. The maximum effective sampling rates that can be achieved with this method are listed in the User’s Guide for the scope device.

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

- **Callback.** ETS mode calls the ps5000aBlockReady callback function when a new waveform is ready for collection. Call ps5000aGetValues to retrieve the waveform.

<table>
<thead>
<tr>
<th>Applicability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available in block mode only.</td>
</tr>
<tr>
<td>Not suitable for one-shot (non-repetitive) signals.</td>
</tr>
<tr>
<td>Aggregation is not supported.</td>
</tr>
<tr>
<td>Edge-triggering only.</td>
</tr>
<tr>
<td>Auto trigger delay (autoTriggerMilliseconds) is ignored.</td>
</tr>
<tr>
<td>Cannot be used when MSO digital ports are enabled.</td>
</tr>
<tr>
<td>Available in 8-bit resolution mode only.</td>
</tr>
<tr>
<td>On PicoScope 5000D Series scopes, available on channel A only.</td>
</tr>
</tbody>
</table>
3.5.3.1 Using ETS mode

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

1. Open the oscilloscope using `ps5000aOpenUnit`.
2. Select analog channel ranges and AC/DC coupling using `ps5000aSetChannel`.
3. Use `ps5000aGetTimebase` to verify the number of samples to be collected.
4. Set up ETS using `ps5000aSetEts`.
5. Use the trigger setup function `ps5000aSetSimpleTrigger` to set up the trigger.
6. Start the oscilloscope running using `ps5000aRunBlock`.
7. Wait until the oscilloscope is ready using the `ps5000aBlockReady` callback (or poll using `ps5000aIsReady`).
8. Use `ps5000aSetDataBuffer` to tell the driver where to store sampled data.
8a. Use `ps5000aSetEtsTimeBuffer` or `ps5000aSetEtsTimeBuffers` to tell the driver where to store sample times.
9. Transfer the block of data from the oscilloscope using `ps5000aGetValues`.
10. Display the data.
11. While you want to collect updated captures, repeat steps 7 to 10.
12. Repeat steps 6 to 11.
13. Stop the oscilloscope using `ps5000aStop`.
14. Close the device using `ps5000aCloseUnit`. 
3.5.4 Streaming mode

**Streaming mode** can capture data without the gaps that occur between blocks when using block mode. Streaming mode supports downsampling and triggering, while providing fast streaming at up to 125 MS/s (8 ns per sample) when one channel is active, depending on the computer's performance. This makes it suitable for **high-speed data acquisition**, allowing you to capture long data sets limited only by the computer's memory.

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

- **Memory segmentation.** The memory can be divided into **segments** to reduce the latency of data transfers to the PC. However, this increases the risk of losing data if the PC cannot keep up with the device's sampling rate. See **Using streaming mode** for programming details.

### 3.5.4.1 Using streaming mode

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

1. Open the oscilloscope using `ps5000aOpenUnit`.
2. Select channels, ranges and AC/DC coupling using `ps5000aSetChannel`.
2a. Set the digital port using `ps5000aSetDigitalPort` (mixed-signal scopes only).
3. Use the trigger setup function `ps5000aSetSimpleTrigger` to set up the trigger if required.
3a. Use the trigger setup functions `ps5000aSetTriggerDigitalPortProperties` and `ps5000aSetTriggerChannelConditions` to set up the digital trigger if required (mixed-signal scopes only).
4. Call `ps5000aSetDataBuffer` (or `ps5000aSetDataBuffers` if you will be using aggregation) to tell the driver where your data buffer is.
5. Start the oscilloscope running (with aggregation if required) using `ps5000aRunStreaming`. In this example we set `autostop = 1` to stop the oscilloscope collecting data when it has retrieved the requested number of samples.
6. Call `ps5000aGetStreamingLatestValues` to get data. Repeat until enough data is collected.
7. Process data returned to your application's function. This example is using `autoStop = 1`, so after the driver has received all the data points requested by the application, it stops the device streaming.
8. Call `ps5000aStop`. This is necessary even when `autoStop = 1`.
9. Optionally, request new views of stored data using different downsampling parameters: see **Retrieving stored data**.
10. Close the device using `ps5000aCloseUnit`.
3.5.5 Retrieving stored data

You can collect data from the ps5000a driver with a different downsampling factor when `ps5000aRunBlock` or `ps5000aRunStreaming` has already been called and has successfully captured all the data. Use `ps5000aGetValuesAsync`.
### 3.6 Timebases

The timebase is an integer that encodes the sampling interval of the oscilloscope. The API allows you to select any available* timebase down to the minimum sampling interval of your oscilloscope. The available timebases allow slow enough sampling in block mode to overlap the streaming sample intervals, so that you can make a smooth transition between block mode and streaming mode.

Convert a given timebase to a sampling interval using `ps5000aGetTimebase`. Find the fastest available timebase in a given mode using `ps5000aGetMinimumTimebaseStateless`.

Accepted timebases for each resolution mode are as follows:

#### 8-bit mode

<table>
<thead>
<tr>
<th>Timebase (n)</th>
<th>Sampling interval formula</th>
<th>Sampling interval</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>1 ns</td>
<td>Only channel enabled</td>
</tr>
<tr>
<td>1</td>
<td>$2^n / 1,000,000,000$</td>
<td>2 ns</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>4 ns</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$(n-2) / 125,000,000$</td>
<td>8 ns</td>
<td>~ 34.36 s</td>
</tr>
</tbody>
</table>

#### 12-bit mode

<table>
<thead>
<tr>
<th>Timebase (n)**</th>
<th>Sampling interval formula</th>
<th>Sampling interval</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$2^{(n-1)} / 500,000,000$</td>
<td>2 ns</td>
<td>Only channel enabled</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>4 ns</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>8 ns</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$(n-3) / 62,500,000$</td>
<td>16 ns</td>
<td>~ 68.72 s</td>
</tr>
</tbody>
</table>

#### 14-bit mode

<table>
<thead>
<tr>
<th>Timebase (n)†</th>
<th>Sampling interval formula</th>
<th>Sampling interval</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>$1 / 125,000,000$</td>
<td>8 ns</td>
<td>5000A/B Series: only one analog channel enabled. 5000D Series: up to 4 analog channels or digital ports enabled.</td>
</tr>
<tr>
<td>4</td>
<td>$(n-2) / 125,000,000$</td>
<td>16 ns</td>
<td>~ 34.36 s</td>
</tr>
</tbody>
</table>

#### 15-bit mode

*PicoScope 5000D MSO Series: any number of digital ports can be enabled without affecting the timebase.*

<table>
<thead>
<tr>
<th>Timebase (n)†</th>
<th>Sampling interval formula</th>
<th>Sampling interval</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>$1 / 125,000,000$</td>
<td>8 ns</td>
<td>Up to two analog channels enabled.</td>
</tr>
<tr>
<td>4</td>
<td>$(n-2) / 125,000,000$</td>
<td>16 ns</td>
<td>~ 34.36 s</td>
</tr>
</tbody>
</table>

#### 16-bit mode

*PicoScope 5000D MSO Series: any number of digital ports can be enabled without affecting the timebase.*

<table>
<thead>
<tr>
<th>Timebase (n)‡</th>
<th>Sampling interval formula</th>
<th>Sampling interval</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>$1 / 62,500,000$</td>
<td>16 ns</td>
<td>Only one analog channel enabled.</td>
</tr>
<tr>
<td>5</td>
<td>$(n-3) / 62,500,000$</td>
<td>32 ns</td>
<td>~ 68.72 s</td>
</tr>
</tbody>
</table>
The fastest available sampling rate depends on the combination of channels and ports enabled, the sampling mode, the ETS mode and the power supply mode. Please refer to the oscilloscope data sheet for sampling rate specifications. In streaming mode, the speed of the USB port may affect the rate of data transfer.

**Timebase 0 is not available in 12-bit resolution mode.**

† Timebases 0, 1 and 2 are not available in 14 and 15-bit resolution modes.

‡ Timebases 0, 1, 2 and 3 are not available in 16-bit resolution mode.

**ETS mode**

In ETS mode the sample time is not set according to the above tables, but is instead calculated and returned by `ps5000aSetEts`. 
3.7 Power options

The 5000A/B Series oscilloscopes allow you to choose from two different methods of powering your device: using a standard Pico USB cable and the power supply provided, or using a double-headed Pico USB cable (available separately) to obtain power from two powered USB ports. For 4-channel devices, the second method is available only in 2-channel mode.

The 5000D Series 4-channel scopes also have a choice of two power sources. When one or two channels are enabled, you can power the scope from a USB port that supplies at least 1200 mA or you can use the AC adaptor supplied. If you do not have a USB port capable of supplying 1200 mA, you can instead use a double-headed USB cable, available separately, connected to two USB ports. When three or four channels are enabled, you must use the AC adaptor.

If the power source is changed (i.e. AC adaptor connected or disconnected) while the oscilloscope is in operation, the device will restart automatically and any unsaved data will be lost.

For further information on these options, refer to the documentation included with your device.

Power options functions
The following functions control the power options:

- `ps5000aChangePowerSource`
- `ps5000aCurrentPowerSource`

If you call `ps5000aOpenUnit` without the power supply connected, the function returns `PICO_POWER_SUPPLY_NOT_CONNECTED` and passes back a valid handle argument. If you want the device to run on USB power only, you can then instruct the driver by passing this handle to `ps5000aChangePowerSource`.

If the power supply is connected or disconnected during use, the driver will return the relevant status code and you must then call `ps5000aChangePowerSource` to continue running the scope.

The 2-channel 5000D and 5000D MSO scopes return `PICO_USB3_0_DEVICE_NON_USB3_0_PORT` when connected to a non-USB 3.0 port.
3.8 Combining several oscilloscopes

It is possible to collect data using up to 64 PicoScope 5000 Series oscilloscopes at the same time, depending on the capabilities of the PC. Each oscilloscope must be connected to a separate USB port. The `ps5000aOpenUnit` function assigns a handle (a device identifier) to an oscilloscope. Almost all the other functions require this handle for oscilloscope identification. For example, to collect data from two oscilloscopes at the same time:

```c
CALLBACK ps5000aBlockReady(...)
    // Define callback function specific to application

ps5000aOpenUnit(&handle1)
ps5000aOpenUnit(&handle2)

ps5000aSetChannel(handle1)  // Set up device 1
ps5000aRunBlock(handle1)

ps5000aSetChannel(handle2)  // Set up device 2
ps5000aRunBlock(handle2)

// Data will be stored in buffers and application notified using a callback.
// The callback arguments include the device handle, which identifies the
// device that generated the data.

ready = false
while not ready
    ready = handle1_ready
    ready &= handle2_ready
```
4 API functions

The ps5000a API exports the following functions for you to use in your own applications. They are all exported with both decorated and undecorated names.

4.1 ps5000aChangePowerSource – select USB or AC adaptor power

```c
PICO_STATUS ps5000aChangePowerSource
(
    int16_t        handle,
    PICO_STATUS    powerstate
)
```

This function selects the power supply mode. If USB power is required, you must explicitly allow it by calling this function. You must also call this function if the AC power adapter is connected or disconnected during use. If you change the power source to `PICO_POWER_SUPPLY_NOT_CONNECTED` and either of channels C and D is currently enabled, they will be switched off. If a trigger is set using channel C or D, the trigger settings for those channels will also be removed.

**Applicability**
All modes.

**Arguments**

- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `powerstate`, the required state of the unit; one of the following:
  - `PICO_POWER_SUPPLY_CONNECTED` – to run the device on AC adaptor power
  - `PICO_POWER_SUPPLY_NOT_CONNECTED` – to run the device on USB power
  - `PICO_USB3_0_DEVICE_NON_USB3_0_PORT` – for 2-channel 5000D and 5000D MSO devices

**Returns**

- `PICO_OK` or other code from `PicoStatus.h`
4.2 ps5000aChannelCombinationsStateless – find out which channels can be used

```c
PICO_STATUS ps5000aChannelCombinationsStateless
(
    int16_t handle,
    PS5000A_CHANNEL_FLAGS * channelOrPortFlagsCombinations,
    uint32_t * nChannelCombinations,
    PS5000A_DEVICE_RESOLUTION resolution,
    uint32_t timebase,
    int16_t hasDcPowerSupplyConnected
)
```

This function accepts a proposed device configuration and returns a list of available channel combinations that can be used under that configuration. It does not write the configuration to the device.

The function is designed to be called twice. First, call with `channelOrPortFlagsCombinations = NULL` and note the value of `nChannelCombinations` that the function returns. Then create an array with space for this number of `PS5000A_CHANNEL_FLAGS` values and call the function again with `channelOrPortFlagsCombinations` pointing to the array. On the second call, the function will populate the array with `PS5000A_CHANNEL_FLAGS` values.

**Applicability**

All modes

**Arguments**

- `handle`, the device identifier returned by `ps5000aOpenUnit`.

- `channelOrPortFlagsCombinations`, on exit, an array of possible channel and port combinations – see `PS5000A_CHANNEL_FLAGS`. Set to `NULL` to query the number of combinations without returning a list of values.

- `nChannelCombinations`, on entry and exit, the length of the `channelOrPortFlagCombinations` array.

- `resolution`, the proposed hardware resolution – see `PS5000A_DEVICE_RESOLUTION`.

- `timebase`, the proposed timebase number, as passed to `ps5000aGetTimebase`.

- `hasDcPowerSupplyConnected`, whether the proposed configuration uses the external AC adaptor or not:
  - `0` = not using AC adaptor
  - `1` = using AC adaptor

**Returns**

`PICO_OK` or other code from `PicoStatus.h`
4.2.1 PS5000A_CHANNEL_FLAGS enumerated type

```c
typedef enum enPS5000AChannelFlags {
    PS5000A_CHANNEL_A_FLAGS = 1,
    PS5000A_CHANNEL_B_FLAGS = 2,
    PS5000A_CHANNEL_C_FLAGS = 4,
    PS5000A_CHANNEL_D_FLAGS = 8,
    PS5000A_PORT0_FLAGS     = 65536,
    PS5000A_PORT1_FLAGS     = 131072,
    PS5000A_PORT2_FLAGS     = 262144,
    PS5000A_PORT3_FLAGS     = 524288
} PS5000A_CHANNEL_FLAGS;
```

These single-bit values identify channels. They can be ORed together to indicate channel and port combinations.

**Applicability**

Calls to `ps5000aChannelCombinationsStateless`

**Values**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS5000A_CHANNEL_A_FLAGS</td>
<td>analog channel A</td>
</tr>
<tr>
<td>PS5000A_CHANNEL_B_FLAGS</td>
<td>analog channel B</td>
</tr>
<tr>
<td>PS5000A_CHANNEL_C_FLAGS</td>
<td>analog channel C (4-channel models only)</td>
</tr>
<tr>
<td>PS5000A_CHANNEL_D_FLAGS</td>
<td>analog channel D (4-channel models only)</td>
</tr>
<tr>
<td>PS5000A_PORT0_FLAGS</td>
<td>digital port 0 (inputs D0–D7; MSO models only)</td>
</tr>
<tr>
<td>PS5000A_PORT1_FLAGS</td>
<td>digital port 1 (inputs D8–D15; MSO models only)</td>
</tr>
</tbody>
</table>
4.3 ps5000aCloseUnit – close a scope device

```c
PICO_STATUS ps5000aCloseUnit
(int16_t handle)
```

This function shuts down the PicoScope 5000 Series oscilloscope.

<table>
<thead>
<tr>
<th>Applicability</th>
</tr>
</thead>
<tbody>
<tr>
<td>All modes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle, the device identifier that was returned by ps5000aOpenUnit. When ps5000aCloseUnit returns, this value of handle will no longer be valid.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>PICO_OK or other code from PicoStatus.h</td>
</tr>
</tbody>
</table>
4.4 ps5000aCurrentPowerSource – indicate the current power state of the device

```
PICO_STATUS ps5000aCurrentPowerSource
(
    int16_t handle
)
```

This function returns the current power state of the device.

**Applicability**

All modes.

**Arguments**

handle, the device identifier returned by `ps5000aOpenUnit`.

**Returns**

PICO_INVALID_HANDLE – handle of the device is not recognized.
PICO_POWER_SUPPLY_CONNECTED – device is powered by the AC adaptor.
PICO_POWER_SUPPLY_NOT_CONNECTED – device is powered by the USB cable.
PICO_USB3_0_DEVICE_NON_USB3_0_PORT – a 2-channel 5000D or 5000D MSO model is connected to a USB 2.0 port.
PICO_OK – the device has two channels and PICO_USB3_0DEVICE_NON_USB3_0_PORT does not apply.
4.5 ps5000aEnumerateUnits – find all connected oscilloscopes

```c
PICO_STATUS ps5000aEnumerateUnits
(
    int16_t * count,
    int8_t * serials,
    int16_t * serialLth
)
```

This function counts the number of PicoScope 5000 Series units connected to the computer, and returns a list of serial numbers as a string. Note that this function will only detect devices that are not yet being controlled by an application.

**Applicability**
All modes

**Arguments**
- * count, on exit, the number of PicoScope 5000 Series units found.
- * serials, on exit, a list of serial numbers separated by commas and terminated by a final null. Example: AQ005/139, VDR61/356, ZOR14/107. Can be NULL on entry if serial numbers are not required.
- * serialLth, on entry, the length of the buffer pointed to by serials; on exit, the length of the string written to serials. Includes the terminating null character.

**Returns**
PICO_OK or other code from PicoStatus.h
4.6  ps5000aFlashLed – flash the front-panel LED

```c
PICO_STATUS ps5000aFlashLed
(
    int16_t        handle,
    int16_t        start
)
```

This function flashes the LED on the front of the scope without blocking the calling thread. Calls to `ps5000aRunStreaming` and `ps5000aRunBlock` cancel any flashing started by this function. It is not possible to set the LED to be constantly illuminated, as this state is used to indicate that the scope has not been initialized.

**Applicability**

All modes

**Arguments**

- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `start`, the action required:
  - `< 0`: flash the LED indefinitely.
  - `0`: stop the LED flashing.
  - `> 0`: flash the LED `start` times. If the LED is already flashing on entry to this function, the flash count will be reset to `start`.

**Returns**

`PICO_OK` or other code from `PicoStatus.h`
4.7 ps5000aGetAnalogueOffset – query the permitted analog offset range

```c
PICO_STATUS ps5000aGetAnalogueOffset
(
    int16_t handle,
    PS5000A_RANGE range,
    PS5000A_COUPLING coupling,
    float * maximumVoltage,
    float * minimumVoltage
)
```

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

<table>
<thead>
<tr>
<th>Applicability</th>
</tr>
</thead>
<tbody>
<tr>
<td>All models</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle, the device identifier returned by ps5000aOpenUnit.</td>
</tr>
</tbody>
</table>

| range, the voltage range to be used when gathering the min and max information. See PS5000A_RANGE. |

| coupling, the type of AC/DC coupling used. See PS5000A_COUPLING. |

| maximumVoltage, a pointer to a float, an out parameter set to the maximum voltage allowed for the range, may be NULL. |

| minimumVoltage, a pointer to a float, an out parameter set to the minimum voltage allowed for the range, may be NULL. |

If both maximumVoltage and minimumVoltage are set to NULL, the driver returns PICO_NULL_PARAMETER.

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>PICO_OK or other code from PicoStatus.h</td>
</tr>
</tbody>
</table>
4.7.1 PS5000A_RANGE enumerated type

typedef enum enPS5000ARange
{
    PS5000A_10MV,
    PS5000A_20MV,
    PS5000A_50MV,
    PS5000A_100MV,
    PS5000A_200MV,
    PS5000A_500MV,
    PS5000A_1V,
    PS5000A_2V,
    PS5000A_5V,
    PS5000A_10V,
    PS5000A_20V,
    PS5000A_50V,
    PS5000A_MAX_RANGES
} PS5000A_RANGE;

These values specify all the possible voltage ranges to which an analog input channel can be set. Each range is bipolar, so for example the PS5000A_10MV range spans from –10 mV to +10 mV.

**Applicability**
Calls to `ps5000aGetAnalogueOffset` etc.

**Values**
- PS5000A_10MV ±10 mV range
- ...
- PS5000A_20V ±20 V range
- PS5000A_50V not available

4.7.2 PS5000A_COUPLING enumerated type

typedef enum enPS5000ACoupling
{
    PS5000A_AC,
    PS5000A_DC
} PS5000A_COUPLING;

These values specify the two possible input coupling modes for each analog channel.

**Applicability**
Calls to `ps5000aGetAnalogueOffset` etc.

**Arguments**
- PS5000A_AC – 1 megohm impedance, AC coupling. The channel accepts input frequencies from about 1 hertz up to its maximum –3 dB analog bandwidth.
- PS5000A_DC – 1 megohm impedance, DC coupling. The scope accepts all input frequencies from zero (DC) up to its maximum –3 dB analog bandwidth.
4.8 ps5000aGetChannelInformation – query which ranges are available on a device

```c
PICO_STATUS ps5000aGetChannelInformation
(
    int16_t handle,
    PS5000A_CHANNEL_INFO info,
    int32_t probe,
    int32_t ranges,
    int32_t length,
    int32_t channels
)
```

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

**Applicability**

All modes

**Arguments**

- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `info`, the type of information required: see `PS5000A_CHANNEL_INFO`. Only one type is available:
  - `PS5000A_CI_RANGES` – input voltage ranges
- `probe`, not used, must be set to 0.
- `ranges`, an array that will be populated with available `PS5000A_RANGE` values for the given `info`. If NULL, `length` is set to the number of `ranges` available.
- `length`, on input: the length of the ranges array; on output: the number of elements written to the ranges array.
- `channels`, the channel for which the information is required.

**Returns**

PICO_OK or other code from `PicoStatus.h`

### 4.8.1 PS5000A_CHANNEL_INFO enumerated type

```c
typedef enum enPS5000AChannelInfo
{
    PS5000A_CI_RANGES,
} PS5000A_CHANNEL_INFO;
```

Only one type of channel information—ranges—is available for the 5000 Series oscilloscopes.

**Applicability**

Calls to `ps5000aGetChannelInformation`.

**Values**

- `PS5000A_CI_RANGES` – obtain range information.
4.9 ps5000aGetDeviceResolution – retrieve the resolution the device will run in

```c
PICO_STATUS ps5000aGetDeviceResolution
(int16_t handle,
 PS5000A_DEVICE_RESOLUTION * resolution)
```

This function retrieves the resolution the specified device will run in.

**Applicability**
All modes

**Arguments**
handle, the device identifier returned by `ps5000aOpenUnit`.

* resolution, returns the resolution of the device.

**Returns**
PICO_OK or other code from PicoStatus.h
4.10 ps5000aGetMaxDownSampleRatio – query the aggregation ratio for data

```
PICO_STATUS ps5000aGetMaxDownSampleRatio(
    int16_t handle,
    uint32_t noOfUnaggregatedSamples,
    uint32_t * maxDownSampleRatio,
    PS5000A_RATIO_MODE downSampleRatioMode,
    uint32_t segmentIndex
)
```

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

**Applicability**
All modes

**Arguments**
- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `noOfUnaggregatedSamples`, the number of unprocessed samples to be downsampled
- `* maxDownSampleRatio`, the maximum possible downsampling ratio output
- `downSampleRatioMode`, the downsampling mode. See `ps5000aGetValues`
- `segmentIndex`, the memory segment where the data is stored

**Returns**
PICO_OK or other code from PicoStatus.h
4.11 ps5000aGetMaxSegments – query the maximum number of segments

```c
PICO_STATUS ps5000aGetMaxSegments
(
    int16_t handle,
    uint32_t * maxsegments
)
```

This function returns the maximum number of segments allowed for the opened device. Refer to `ps5000aMemorySegments` for specific figures.

**Applicability**
All modes

**Arguments**
handle, the device identifier returned by `ps5000aOpenUnit`.

maxsegments, on exit, the maximum number of segments allowed.

**Returns**
PICO_OK or other code from `PicoStatus.h`
4.12 ps5000aGetMinimumTimebaseStateless – find fastest available timebase

```c
PICO_STATUS ps5000aGetMinimumTimebaseStateless
(
    int16_t handle,
    PS5000A_CHANNEL_FLAGS enabledChannelOrPortFlags,
    uint32_t * timebase,
    double * timeInterval,
    PS5000A_DEVICE_RESOLUTION resolution
)
```

This function returns the fastest available timebase for the proposed device configuration. It does not write the proposed configuration to the device.

**Applicability**
All modes

**Arguments**
- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `enabledChannelOrPortFlags`, the proposed combination of enabled channels and ports. To specify multiple channels and ports, use the bitwise-OR of the relevant `PS5000A_CHANNEL_FLAGS` values.
- `timebase`, on exit, the shortest timebase available.
- `timeInterval`, on exit, the sampling interval, in seconds, corresponding to the stated timebase.
- `resolution`, the resolution mode in which you propose to operate the oscilloscope.

**Returns**
PICO_OK or other code from PicoStatus.h
4.13 ps5000aGetNoOfCaptures – find out how many captures are available

```
PICO_STATUS ps5000aGetNoOfCaptures(
    int16_t handle,
    uint32_t * nCaptures
)
```

This function returns the number of captures the device has made in rapid block mode, since you called ps5000aRunBlock. You can call ps5000aGetNoOfCaptures during device capture, after collection has completed or after interrupting waveform collection by calling ps5000aStop. The returned value (nCaptures) can then be used to iterate through the number of segments using ps5000aGetValues, or in a single call to ps5000aGetValuesBulk, where it is used to calculate the toSegmentIndex parameter.

**Applicability**
Rapid block mode

**Arguments**
handle, the device identifier returned by ps5000aOpenUnit.

* nCaptures, output: the number of available captures that has been collected from calling ps5000aRunBlock.

**Returns**
PICO_OK or other code from PicoStatus.h
4.14 ps5000aGetNoOfProcessedCaptures – query number of captures processed

```
PICO_STATUS ps5000aGetNoOfProcessedCaptures
(
  int16_t handle,
  uint32_t * nProcessedCaptures
)
```

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

The function returns the number of captures the driver has processed since you called `ps5000aRunBlock`. It is for use in rapid block mode, alongside the `ps5000aGetValuesOverlappedBulk` function, when the driver is set to transfer data from the device automatically as soon as the `ps5000aRunBlock` function is called. You can call `ps5000aGetNoOfProcessedCaptures` during device capture, after collection has completed or after interrupting waveform collection by calling `ps5000aStop`.

The returned value (`nProcessedCaptures`) can then be used to iterate through the number of segments using `ps5000aGetValues`, or in a single call to `ps5000aGetValuesBulk`, where it is used to calculate the `toSegmentIndex` parameter.

**When capture is stopped**

If `nProcessedCaptures` = 0, you will also need to call `ps5000aGetNoOfCaptures`, in order to determine how many waveform segments were captured, before calling `ps5000aGetValues` or `ps5000aGetValuesBulk`.

**Applicability**

Rapid block mode, using `ps5000aGetValuesOverlapped`

**Arguments**

- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `* nProcessedCaptures`, output: the number of available captures that has been collected from calling `ps5000aRunBlock`.

**Returns**

PICO_OK or other code from `PicoStatus.h`
4.15 ps5000aGetStreamingLatestValues – get streaming data while scope is running

```c
PICO_STATUS ps5000aGetStreamingLatestValues
(
    int16_t handle,
    ps5000aStreamingReady lpPs5000aReady,
    void * pParameter
)
```

This function instructs the driver to return the next block of values to your `ps5000aStreamingReady` callback function. You must have previously called `ps5000aRunStreaming` beforehand to set up streaming.

In most cases the block of values returned will not be enough to fill the data buffer, so you will need to call `ps5000aGetStreamingLatestValues` repeatedly until you have obtained the required number of samples. The timing between calls to the function depends on your application – it should be fast enough to avoid running out data but not so fast that it wastes processor time.

**Applicability**

**Streaming** mode only

**Arguments**

- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `lpPs5000aReady`, a pointer to your `ps5000aStreamingReady` callback function.
- `pParameter`, a void pointer that will be passed to the `ps5000aStreamingReady` callback function. The callback function may optionally use this pointer to return information to the application.

**Returns**

- `PICO_OK` or other code from `PicoStatus.h`
4.16 ps5000aGetTimebase – get properties of the selected timebase

PICO_STATUS ps5000aGetTimebase(
    int16_t handle,
    uint32_t timebase,
    int32_t noSamples,
    int32_t * timeIntervalNanoseconds,
    int32_t * maxSamples,
    uint32_t segmentIndex
)

This function calculates the sampling rate and maximum number of samples for a given timebase under the specified conditions. The result will depend on the number of channels enabled by the last call to ps5000aSetChannel.

This function is provided for use with programming languages that do not support the float data type. The value returned in the timeIntervalNanoseconds argument is restricted to integers. If your programming language supports the float type, then we recommend that you use ps5000aGetTimebase2 instead.

To use ps5000aGetTimebase or ps5000aGetTimebase2, first estimate the timebase number that you require using the information in the timebase guide. Next, call one of these functions with the timebase that you have just chosen and verify that the timeIntervalNanoseconds argument that the function returns is the value that you require. You may need to iterate this process until you obtain the time interval that you need.

<table>
<thead>
<tr>
<th>Applicability</th>
</tr>
</thead>
<tbody>
<tr>
<td>All modes</td>
</tr>
</tbody>
</table>

**Arguments**

- **handle**, the device identifier returned by ps5000aOpenUnit.
- **timebase**, see timebase guide
- **noSamples**, the number of samples required.

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

  * **maxSamples**, on exit, the maximum number of samples available. The scope reserves some 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 or other code from PicoStatus.h
4.17 ps5000aGetTimebase2 – get properties of the selected timebase

`PICO_STATUS` ps5000aGetTimebase2
(`
  int16_t handle,
  uint32_t timebase,
  int32_t noSamples,
  float * timeIntervalNanoseconds,
  int32_t * maxSamples,
  uint32_t segmentIndex
`)

This function is an upgraded version of `ps5000aGetTimebase`, and returns the time interval as a `float` rather than an `int32_t`. This allows it to return sub-nanosecond time intervals. See `ps5000aGetTimebase` for a full description.

<table>
<thead>
<tr>
<th>Applicability</th>
</tr>
</thead>
<tbody>
<tr>
<td>All modes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle, timebase, noSamples, see <code>ps5000aGetTimebase</code>.</td>
</tr>
<tr>
<td>* timeIntervalNanoseconds, a pointer to the time interval between readings at the selected timebase. If a null pointer is passed, nothing will be written here.</td>
</tr>
<tr>
<td>* maxSamples, segmentIndex, see <code>ps5000aGetTimebase</code>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PICO_OK</code> or other code from <code>PicoStatus.h</code></td>
</tr>
</tbody>
</table>
4.18 ps5000aGetTriggerInfoBulk – get trigger timestamps

```c
PICO_STATUS ps5000aGetTriggerInfoBulk(
    int16_t handle,
    PS5000A_TRIGGER_INFO * triggerInfo,
    uint32_t fromSegmentIndex,
    uint32_t toSegmentIndex
)
```

**ps5000aGetTriggerInfoBulk** is used to retrieve information about the trigger point in one or more segments of captured data, in the form of a **PS5000A_TRIGGER_INFO** structure or array of structures. This function can be used with **ps5000aGetValues**, **ps5000aGetValuesBulk**, **ps5000aGetValuesAsync**, **ps5000aGetValuesOverlapped** and **ps5000aGetValuesOverlappedBulk**. Although it is primarily intended for use with **ps5000aTriggerWithinPreTriggerSamples**, it can be used with any block mode capture when ETS is off and trigger delay is 0.

This function can retrieve trigger information for more than one segment at once by using **fromSegmentIndex** and **toSegmentIndex**. These values are both inclusive so, to collect details for a single segment, set **fromSegmentIndex** equal to **toSegmentIndex**.

**Applicability**

Block mode, rapid block mode.

**Arguments**

* handle*, the device identifier returned by **ps5000aOpenUnit**.

* triggerInfo*, a pointer to one or more **PS5000A_TRIGGER_INFO** objects. When collecting details for a single segment, this parameter should be a pointer to a single object. When collecting details for more than one segment the parameter should be a pointer to an array of objects, of length greater than or equal to the number of **PS5000A_TRIGGER_INFO** elements requested.

* fromSegmentIndex*, the zero-based number of the first segment of interest.

* toSegmentIndex*, the zero-based number of the last segment of interest. If **fromSegmentIndex** > **toSegmentIndex**, the segment index will wrap from the last segment back to 0.

**Returns**

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

If the function return status is **PICO_OK**, all the **triggerInfo** status codes will be **PICO_OK** or **PICO_DEVICE_TIME_STAMP_RESET**.

If the return status is any other status code, check the individual element status codes as some of the elements could be **PICO_OK** and others could show an error (for example, if you request trigger information for a range of segments but have not captured data to some of them).
4.18.1 PS5000A_TRIGGER_INFO structure

```c
typedef struct tPS500ATriggerInfo {
    PICO_STATUS status;
    uint32_t segmentIndex;
    uint32_t triggerIndex;
    int64_t triggerTime;
    int16_t timeUnits;
    int16_t reserved0;
    uint64_t timeStampCounter;
} PS5000A_TRIGGER_INFO;
```

This structure contains the trigger timestamp information for the specified buffer segment.

**Applicability**

Calls to `ps5000aGetTriggerInfoBulk`. **Rapid block mode** only.

**Elements**

- **status**, a status code indicating success or failure for the segment.
- **segmentIndex**, a zero-based index identifying the segment.
- **triggerIndex**, the index of the trigger point measured in samples within the captured data, with the first sample being index 0. In ordinary triggering this is equal to the number of pre-trigger samples requested. When using `ps5000aTriggerWithinPreTriggerSamples` this element is used to find the location of the trigger point, which may fall anywhere within the pre-trigger samples.
- **triggerTime**, the trigger offset time as returned by `ps5000aGetTriggerTimeoffset` or `ps5000aGetTriggerTimeoffset64`. These elements are included in this structure to avoid the need to call those functions separately.
- **timeUnits**, the unit of time in which **triggerTime** is expressed. See `PS5000A_TIME_UNITS`.
- **reserved0**, not used.
- **timeStampCounter**, the number of sample intervals between the trigger point of this segment and the previous segment. This allows you to determine the time interval between the trigger points of captures within a single rapid block run: see Timestamping.

4.18.2 Time stamping

The **timeStampCounter** parameter in the `PS5000A_TRIGGER_INFO` structure allows you to determine the time interval between the trigger points of captures within a single **rapid block** run. 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 in any run 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 between runs, 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, 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.

In normal block mode (one segment per run, i.e. not rapid block mode) all segments may have PICO_DEVICE_TIME_STAMP_RESET set, and no timing information can be inferred. 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 ps5000aRunBlock in conjunction with ps5000aGetTimebase.

timeStampCounter is a 48-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 48 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 3 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.
4.18.3  PS5000A_TIME_UNITS enumerated type

typedef enum enPS5000ATimeUnits
{
    PS5000A_FS,
    PS5000A_PS,
    PS5000A_NS,
    PS5000A_US,
    PS5000A_MS,
    PS5000A_S,
    PS5000A_MAX_TIME_UNITS,
} PS5000A_TIME_UNITS;

<table>
<thead>
<tr>
<th>Applicability</th>
<th>Any function that requires time units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Values</td>
<td>PS5000A_FS, femtoseconds ($10^{-15}$ s)</td>
</tr>
<tr>
<td></td>
<td>PS5000A_PS, picoseconds ($10^{-12}$ s)</td>
</tr>
<tr>
<td></td>
<td>PS5000A_NS, nanoseconds ($10^{-9}$ s)</td>
</tr>
<tr>
<td></td>
<td>PS5000A_US, microseconds ($10^{-6}$ s)</td>
</tr>
<tr>
<td></td>
<td>PS5000A_MS, milliseconds ($10^{-3}$ s)</td>
</tr>
<tr>
<td></td>
<td>PS5000A_S, seconds (s)</td>
</tr>
</tbody>
</table>
4.19 ps5000aGetTriggerTimeOffset – find out when trigger occurred (32-bit)

```c
PICO_STATUS ps5000aGetTriggerTimeOffset
(
    int16_t handle,
    uint32_t * timeUpper,
    uint32_t * timeLower,
    PS5000A_TIME_UNITS * timeUnits,
    uint32_t segmentIndex
)
```

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

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

This function is provided for use in programming environments that do not support 64-bit integers. Another version of this function, `ps5000aGetTriggerTimeOffset64`, is available that returns the time as a single 64-bit value.

**Applicability**

Block mode, rapid block mode

**Arguments**

- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `* timeUpper`, on exit, the upper 32 bits of the time at which the trigger point occurred
- `* timeLower`, on exit, the lower 32 bits of the time at which the trigger point occurred
- `* timeUnits`, returns the time units in which `timeUpper` and `timeLower` are measured. See `PS5000A_TIME_UNITS`.
- `segmentIndex`, the number of the memory segment for which the information is required.

**Returns**

`PICO_OK` or other code from `PicoStatus.h`
4.20 ps5000aGetTriggerTimeOffset64 – find out when trigger occurred (64-bit)

```c
PICO_STATUS ps5000aGetTriggerTimeOffset64
(  
    int16_t                     handle,  
    int64_t                     * time,  
    PS5000A_TIME_UNITS          * timeUnits,  
    uint32_t                    segmentIndex
)
```

This function gets the trigger time offset for a waveform. It is equivalent to `ps5000aGetTriggerTimeOffset` except that the time offset is returned as a single 64-bit value instead of two 32-bit values.

**Applicability**

Block mode, rapid block mode

**Arguments**

- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `* time`, on exit, the time at which the trigger point occurred
- `* timeUnits`, on exit, the time units in which time is measured. See `PS5000A_TIME_UNITS`.
- `segmentIndex`, the number of the memory segment for which the information is required

**Returns**

`PICO_OK` or other code from `PicoStatus.h`
4.21 ps5000aGetUnitInfo – read information about scope device

```
PICO_STATUS ps5000aGetUnitInfo
(
    int16_t handle,
    int8_t * string,
    int16_t stringLength,
    int16_t * requiredSize,
    PICO_INFO info
)
```

This function retrieves information about the specified oscilloscope or driver software. If the device fails to open or no device is opened, it is still possible to read the driver version.

**Applicability**
All modes

**Arguments**

- **handle**, identifies the device from which information is required. If an invalid handle is passed, only the driver versions can be read.

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

- **stringLength**, the maximum number of 8-bit integers (int8_t) 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 as follows:

<table>
<thead>
<tr>
<th>info</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 PICO_DRIVER_VERSION</td>
<td>Version number of ps5000a.dll</td>
</tr>
<tr>
<td>1 PICO_USB_VERSION</td>
<td>Type of USB connection to device: 1.1, 2.0 or 3.0</td>
</tr>
<tr>
<td>2 PICO_HARDWARE_VERSION</td>
<td>Hardware version of device</td>
</tr>
<tr>
<td>3 PICO_VARIANT_INFO</td>
<td>Variant number of device</td>
</tr>
<tr>
<td>4 PICO_BATCH_AND_SERIAL</td>
<td>Batch and serial number of device</td>
</tr>
<tr>
<td>5 PICO_CAL_DATE</td>
<td>Calibration date of device</td>
</tr>
<tr>
<td>6 PICO_KERNEL_VERSION</td>
<td>Version of kernel driver</td>
</tr>
<tr>
<td>7 PICO_DIGITAL_HARDWARE_VERSION</td>
<td>Hardware version of the digital section</td>
</tr>
<tr>
<td>8 PICO_ANALOGUE_HARDWARE_VERSION</td>
<td>Hardware version of the analog section</td>
</tr>
<tr>
<td>9 PICO_FIRMWARE_VERSION_1</td>
<td>Primary firmware (FPGA code) version</td>
</tr>
<tr>
<td>10 PICO_FIRMWARE_VERSION_2</td>
<td>Secondary firmware (FPGA code) version</td>
</tr>
</tbody>
</table>

**Returns**
PICO_OK or other code from PicoStatus.h
4.22 ps5000aGetValues – retrieve block-mode data with callback

```c
PICO_STATUS ps5000aGetValues
(
    int16_t handle,
    uint32_t startIndex,
    uint32_t * noOfSamples,
    uint32_t downSampleRatio,
    PS5000A_RATIO_MODE downSampleRatioMode,
    uint32_t segmentIndex,
    int16_t * overflow
)
```

This function returns block-mode data from the oscilloscope's buffer memory, with or without down sampling, starting at the specified sample number. It is used to get the stored data after data collection has stopped. It blocks the calling function while retrieving data.

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

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

### Applicability

Block mode, rapid block mode

### Arguments

handle, the device identifier returned by ps5000aOpenUnit.

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

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

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

downSampleRatioMode, which downsampling mode to use. See PS5000A_RATIO_MODE. These values are single-bit constants that can be ORed to apply multiple downsampling modes to the data.

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

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

### Returns

PICO_OK or other code from PicoStatus.h
4.22.1 PS5000A_RATIO_MODE enumerated type

typedef enum enPS5000ARatioMode
{
    PS5000A_RATIO_MODE_NONE         = 0,
    PS5000A_RATIO_MODE_AGGREGATE    = 1,
    PS5000A_RATIO_MODE_DECIMATE     = 2,
    PS5000A_RATIO_MODE_AVERAGE      = 4,
    PS5000A_RATIO_MODE_DISTRIBUTION = 8
} PS5000A_RATIO_MODE;

Various methods of data reduction, or downsampling, are possible with the PicoScope 5000 Series oscilloscopes. The downsampling is done at high speed by the driver, making your application faster and more responsive than if it had to do its own data processing.

You specify the downsampling mode when you call one of the data collection functions such as ps5000aGetValues. The following modes are available:

PS5000A_RATIO_MODE_NONE – No downsampling. Returns raw data values.

PS5000A_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.

PS5000A_RATIO_MODE_AVERAGE – Reduces every block of \( n \) values to a single value representing the average (arithmetic mean) of all the values.

PS5000A_RATIO_MODE_DECIMATE – Reduces every block of \( n \) values to just the first value in the block, discarding all the other values.

PS5000A_RATIO_MODE_DISTRIBUTION – Not used.

Retrieving multiple types of downsampled data
You can optionally retrieve data using more than one downsampling mode with a single call to ps5000aGetValues. Set up a buffer for each downsampling mode by calling ps5000aSetDataBuffer. Then, when calling ps5000aGetValues, set downSampleRatioMode to the bitwise OR of the required downsampling modes.

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

1. Call ps5000aStop.
2. Set up a data buffer for each channel using ps5000aSetDataBuffer with the ratio mode set to PS5000A_RATIO_MODE_NONE.
3. Call ps5000aGetValues to retrieve the data.
4.23 ps5000aGetValuesAsync – retrieve streaming data with callback

```c
PICO_STATUS ps5000aGetValuesAsync
(
    int16_t handle,
    uint32_t startIndex,
    uint32_t noOfSamples,
    uint32_t downSampleRatio,
    PS5000A_RATIO_MODE downSampleRatioMode,
    uint32_t segmentIndex,
    void * lpDataReady,
    void * pParameter
)
```

This function returns data either with or without downsampling, starting at the specified sample number. It is used to get the stored data from the driver after data collection has stopped. It returns the data using a callback.

**Applicability**

Streaming mode and block mode

**Arguments**

- `handle`, `startIndex`, `noOfSamples`, `downSampleRatio`, `downSampleRatioMode`, `segmentIndex`, see ps5000aGetValues.
- `lpDataReady`, a pointer to the user-supplied function that will be called when the data is ready. This will be a ps5000aDataReady function for block-mode data or a ps5000aStreamingReady function for streaming-mode data.
- `pParameter`, a void pointer that will be passed to the callback function. The data type is determined by the application.

**Returns**

PICO_OK or other code from PicoStatus.h
4.24 ps5000aGetValuesBulk – retrieve data in rapid block mode

```c
PICO_STATUS ps5000aGetValuesBulk
(
    int16_t handle,
    uint32_t * noOfSamples,
    uint32_t fromSegmentIndex,
    uint32_t toSegmentIndex,
    uint32_t downSampleRatio, PS5000A_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**, the device identifier returned by `ps5000aOpenUnit`.
- **noOfSamples**, on entry, the number of samples required; on exit, the actual number retrieved. The number of samples retrieved will not be more than the number requested. The data retrieved always starts with the first sample captured.
- **fromSegmentIndex**, the first segment from which the waveform should be retrieved
- **toSegmentIndex**, the last segment from which the waveform should be retrieved
- **downSampleRatio**, **downSampleRatioMode**: see `ps5000aGetValues`.
- **overflow**, an array of integers equal to or larger than the number of waveforms to be retrieved. Each segment index has a corresponding entry in the `overflow` array, with `overflow[0]` containing the flags for the segment numbered `fromSegmentIndex` and the last element in the array containing the flags for the segment numbered `toSegmentIndex`. Each element in the array is a bit field as described under `ps5000aGetValues`.

**Returns**

PICO_OK or other code from `PicoStatus.h`
4.25 ps5000aGetValuesOverlapped – set up data collection ahead of capture

```c
PICO_STATUS ps5000aGetValuesOverlapped(
    int16_t handle,
    uint32_t startIndex,
    uint32_t * noOfSamples,
    uint32_t downSampleRatio,
    PS5000A_RATIO_MODE downSampleRatioMode,
    uint32_t segmentIndex,
    int16_t * overflow
)
```

This function allows you to make a deferred data-collection request in block mode. The request will be executed, and the arguments validated, when you call `ps5000aRunBlock`. The advantage of this function is that the driver makes contact with the scope only once, when you call `ps5000aRunBlock`, compared with the two contacts that occur when you use the conventional `ps5000aRunBlock, ps5000aGetValues` calling sequence. This slightly reduces the dead time between successive captures in block mode.

After calling `ps5000aRunBlock`, you can optionally use `ps5000aGetValues` to request further copies of the data. This might be required if you wish to display the data with different data reduction settings.

For more information, see Using the GetValuesOverlapped functions.

### Applicability

**Block mode**

### Arguments

- `handle`, `startIndex`, `<noOfSamples>†`, `downSampleRatio`, `downSampleRatioMode`, `segmentIndex`, see `ps5000aGetValues`.
- `<overflow>†`, see `ps5000aGetValuesBulk`.

† The driver retains a pointer to `noOfSamples` and `overflow` to report back once the capture has completed. In C# you must pin these arguments.

### Returns

PICO_OK or other code from PicoStatus.h

### 4.25.1 Using the GetValuesOverlapped functions

1. Open the oscilloscope using `ps5000aOpenUnit`.
2. Select channel ranges and AC/DC coupling using `ps5000aSetChannel`.
3. Optionally set up digital inputs using `ps5000aSetDigitalPort` (mixed-signal scopes only).
4. Using `ps5000aGetTimebase`, select timebases until the required nanoseconds per sample is located.
5. Use the trigger setup functions `ps5000aSetSimpleTrigger` to set up the trigger if required.
6. Set up the transfer of the block of data from the oscilloscope using `ps5000aGetValuesOverlapped`.
7. Start the oscilloscope running using `ps5000aRunBlock`.
8. Wait until the oscilloscope is ready using the `ps5000aBlockReady` callback (or poll using `ps5000aIsReady`).
9. Display the data.
10. Repeat steps 7 to 9 if needed.
11. Stop the oscilloscope by calling `ps5000aStop`.

A similar procedure can be used with rapid block mode using the `ps5000aGetValuesOverlappedBulk` function.
4.26 ps5000aGetValuesOverlappedBulk – set up data collection in rapid block mode

```c
PICO_STATUS ps5000aGetValuesOverlappedBulk(
    int16_t handle,
    uint32_t startIndex,
    uint32_t * noOfSamples,
    uint32_t downSampleRatio,
    PS5000A_RATIO_MODE downSampleRatioMode,
    uint32_t fromSegmentIndex,
    uint32_t toSegmentIndex,
    int16_t * overflow
)
```

This function allows you to make a deferred data-collection request in rapid block mode. The request will be executed, and the arguments validated, when you call `ps5000aRunBlock`. The advantage of this method is that the driver makes contact with the scope only once, when you call `ps5000aRunBlock`, compared with the two contacts that occur when you use the conventional `ps5000aRunBlock, ps5000aGetValuesBulk` calling sequence. This slightly reduces the dead time between successive captures in rapid block mode.

After calling `ps5000aRunBlock`, you can optionally use `ps5000aGetValues` to request further copies of the data. This might be required if you wish to display the data with different data reduction settings.

For more information, see Using the GetValuesOverlapped functions.

### Applicability

**Rapid block mode**

### Arguments

- `handle`, `startIndex`, `* noOfSamples†`, `downSampleRatio`, `downSampleRatioMode`, see `ps5000aGetValues`.
- `fromSegmentIndex`, `toSegmentIndex`, `* overflow†`, see `ps5000aGetValuesBulk`.

† The driver retains a pointer to `noOfSamples` and `overflow` to report back once the capture has completed. In C# you must pin these arguments.

### Returns

- `PICO_OK` or other code from `PicoStatus.h`
4.27 ps5000aGetValuesTriggerTimeOffsetBulk – get rapid-block waveform timings (32-bit)

**PICO_STATUS** ps5000aGetValuesTriggerTimeOffsetBulk

(int16_t handle,
uint32_t * timesUpper,
uint32_t * timesLower,
PS5000A_TIME_UNITS * timeUnits,
uint32_t fromSegmentIndex,
uint32_t toSegmentIndex)

This function retrieves the trigger time offset for multiple waveforms obtained in **block mode** or **rapid block mode**. It is a more efficient alternative to calling `ps5000aGetTriggerTimeOffset` once for each waveform required. See `ps5000aGetTriggerTimeOffset` for an explanation of trigger time offsets. There is another version of this function, `ps5000aGetValuesTriggerTimeOffsetBulk64`, that returns trigger time offsets as 64-bit values instead of pairs of 32-bit values.

**Applicability**

**Rapid block mode**

**Arguments**

* handle, the device identifier returned by `ps5000aOpenUnit`.

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

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

* timeUnits, an array of integers. The array must be long enough to hold the number of requested times. On exit, `timeUnits[0]` will contain the time unit for fromSegmentIndex and the last element will contain the time unit for toSegmentIndex. Refer to `ps5000aGetTriggerTimeOffset` for specific figures.

* 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 or other code from PicoStatus.h
4.28 ps5000aGetValuesTriggerTimeOffsetBulk64 – get rapid-block waveform timings (64-bit)

```c
PICO_STATUS ps5000aGetValuesTriggerTimeOffsetBulk64
(
    int16_t handle,
    int64_t *times,
    PS5000A_TIME_UNITS *timeUnits,
    uint32_t fromSegmentIndex,
    uint32_t toSegmentIndex
)
```

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

A 32-bit version of this function, `ps5000aGetValuesTriggerTimeOffsetBulk`, is available for use with programming languages that do not support 64-bit integers. See that function for an explanation of waveform time offsets.

### Applicability
**Rapid block mode**

### Arguments
- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `*times`, an array of integers. On exit, this will hold the time offset for each requested segment index. `times[0]` will hold the time offset for `fromSegmentIndex`, and the last `times` index will hold the time offset for `toSegmentIndex`. The array must be long enough to hold the number of times requested.

- `*timeUnits`, an array of integers long enough to hold the number of requested times. `timeUnits[0]` will contain the time unit for `fromSegmentIndex`, and the last element will contain the `toSegmentIndex`. Refer to `ps5000aGetTriggerTimeOffset64` for specific figures.

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

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

### Returns
- `PICO_OK` or other code from `PicoStatus.h`
4.29 ps5000aIsLedFlashing – check LED status

```
PICO_STATUS ps5000aIsLedFlashing
(
    int16_t handle,
    int16_t * status
)
```

This function reads the status of the front-panel LED.

**Applicability**

*Block mode*

**Arguments**

*handle*, the device identifier returned by `ps5000aOpenUnit`.

*status*, output: indicates the status of the LED:

- 0 = not flashing
- 1 = flashing

**Returns**

PICO_OK or other code from `PicoStatus.h`
4.30 ps5000aIsReady – poll driver in block mode

```c
PICO_STATUS ps5000aIsReady
{
    int16_t handle,
    int16_t * ready
}
```

This function may be used instead of a callback function to receive data from `ps5000aRunBlock`. To use this method, pass a NULL pointer as the lpReady argument to `ps5000aRunBlock`. 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 `ps5000aOpenUnit`.
- `* ready`, output: indicates the state of the collection. If zero, the device is still collecting. If non-zero, the device has finished collecting and `ps5000aGetValues` can be used to retrieve the data.

**Returns**

`PICO_OK` or other code from `PicoStatus.h`
4.31 ps5000aIsTriggerOrPulseWidthQualifierEnabled – find out whether trigger is enabled

```c
PICO_STATUS ps5000aIsTriggerOrPulseWidthQualifierEnabled
    (int16_t handle,
     int16_t * triggerEnabled,
     int16_t * pulseWidthQualifierEnabled)
```

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

<table>
<thead>
<tr>
<th>Applicability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call after setting up the trigger, and just before calling either <code>ps5000aRunBlock</code> or <code>ps5000aRunStreaming</code>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>handle</code>, the device identifier returned by <code>ps5000aOpenUnit</code>.</td>
</tr>
</tbody>
</table>

* `triggerEnabled`, on exit, indicates whether the trigger will successfully be set when `ps5000aRunBlock` or `ps5000aRunStreaming` is called. A non-zero value indicates that the trigger is set, zero that the trigger is not set.

* `pulseWidthQualifierEnabled`, on exit, indicates whether the pulse width qualifier will successfully be set when `ps5000aRunBlock` or `ps5000aRunStreaming` is called. A non-zero value indicates that the pulse width qualifier is set, zero that the pulse width qualifier is not set.

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PICO_OK</code> or other code from <code>PicoStatus.h</code></td>
</tr>
</tbody>
</table>
4.32 ps5000aMaximumValue – get the maximum ADC count

```c
PICO_STATUS ps5000aMaximumValue
(
    int16_t handle,
    int16_t * value
)
```

This function returns a status code and outputs the maximum ADC count value to a parameter. The output value depends on the currently selected resolution.

<table>
<thead>
<tr>
<th>Applicability</th>
<th>All modes</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle, the device identifier returned by <code>ps5000aOpenUnit</code>.</td>
</tr>
</tbody>
</table>

| * value, output: set to the maximum ADC value. |

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>PICO_OK or other code from PicoStatus.h</td>
</tr>
</tbody>
</table>

4.33 ps5000aMemorySegments – divide scope memory into segments

```c
PICO_STATUS ps5000aMemorySegments
(
    int16_t handle,
    uint32_t nSegments,
    int32_t * nMaxSamples
)
```

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

When the scope is opened, the number of segments defaults to 1, meaning that each capture fills the scope's available memory. This function allows you to divide the memory into a number of segments so that the scope can store several waveforms sequentially. After capturing multiple segments, you can query their relative timings by calling `ps5000aGetTriggerInfoBulk`.

**Applicability**

All modes

**Arguments**

- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `nSegments`, the number of segments required. To find the maximum number of memory segments allowed, which may depend on the resolution setting, call `ps5000aGetMaxSegments`.
- `* nMaxSamples`, on exit, the number of samples available in each segment. This is the total number over all channels, so if two channels or 8-bit digital ports are in use, the number of samples available to each channel is `nMaxSamples` divided by 2; for 3 or 4 channels or digital ports divide by 4; and for 5 to 6 channels or digital ports divide by 8.

**Returns**

- `PICO_OK` or other code from `PicoStatus.h`
4.34 ps5000aMinimumValue – get the minimum ADC count

```c
PICO_STATUS ps5000aMinimumValue
(
    int16_t handle,
    int16_t * value
)
```

This function returns a status code and outputs the minimum ADC count value to a parameter. The output value depends on the currently selected resolution.

**Applicability**
All modes

**Arguments**
- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `* value`, output: set to the minimum ADC value.

**Returns**
PICO_OK or other code from PicoStatus.h
4.35 ps5000aNearestSampleIntervalStateless – find nearest available sampling interval

```
    PICO_STATUS ps5000aNearestSampleIntervalStateless
    (
        int16_t handle,
        PS5000A_CHANNEL_FLAGS enabledChannelOrPortFlags,
        double timeIntervalRequested,
        PS5000A_DEVICE_RESOLUTION resolution,
        uint16_t useEts,
        uint32_t * timebase,
        double * timeIntervalAvailable
    )
```

This function accepts a desired sampling interval and a proposed device configuration, and returns the nearest available sampling interval for that configuration. It does not write the proposed configuration to the device.

**Applicability**
All modes

**Arguments**
- **handle**, the device identifier returned by `ps5000aOpenUnit`.
- **enabledChannelOrPortFlags**, the proposed combination of enabled channels and ports. Use the bitwise-OR of the relevant `PS5000A_CHANNEL_FLAGS` values.
- **timeIntervalRequested**, the proposed sampling interval, in seconds.
- **resolution**, the proposed resolution mode.
- **useEts**, the proposed state of ETS:
  - \(0\) = ETS off
  - \(1\) = ETS on

* **timebase**, on exit, the timebase that will result in a sampling interval as close as possible to **timeIntervalRequested**.

* **timeIntervalAvailable**, on exit, the sampling interval corresponding to **timebase**.

**Returns**
- `PICO_OK` or other code from `PicoStatus.h`
4.36 ps5000aNoOfStreamingValues – get number of samples in streaming mode

```c
PICO_STATUS ps5000aNoOfStreamingValues
(
    int16_t handle,
    uint32_t * noOfValues
)
```

This function returns the number of samples available after data collection in streaming mode. Call it after calling `ps5000aStop`.

**Applicability**

Streaming mode

**Arguments**

- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `* noOfValues`, on exit, the number of samples

**Returns**

PICO_OK or other code from PicoStatus.h
4.37  ps5000aOpenUnit – open a scope device

PICO_STATUS  ps5000aOpenUnit
(  
int16_t   * handle,  
int8_t    * serial  
PS5000A_DEVICE_RESOLUTION resolution  
)

This function opens a PicoScope 5000A, 5000B or 5000D Series scope attached to the computer. The maximum number of units that can be opened depends on the operating system, the kernel driver and the computer.

Applicability
All modes

Arguments
* handle,  on exit, the result of the attempt to open a scope:
   -1:   if the scope fails to open (see return value for further information)
    0:   if no scope is found (return value will be PICO_NOT_FOUND)
   > 0:   a number that uniquely identifies the scope until you close the device with ps5000aCloseUnit; use this 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, the function opens the first scope found; otherwise it tries to open the scope that matches the string.

resolution,  determines the resolution of the device when opened. If resolution is out of range, the function returns PICO_INVALID_DEVICE_RESOLUTION.

Returns
PICO_OK or other code from PicoStatus.h

PICO_POWER_SUPPLY_NOT_CONNECTED:
- For a USB 2.0 device, call ps5000aChangePowerSource to complete the two-stage power-up sequence for connection to a USB 2.0 port.
- For a PicoScope 5000D device, this indicates that the device has 4 channels but no PSU is connected. The device will operate but only channels A and B (and digital ports on MSO devices) will be available.
- Note: The device ID passed back in handle is valid and can be passed to ps5000aChangePowerSource.

PICO_USB3_0_DEVICE_NON_USB3_0_PORT:
- Call ps5000aChangePowerSource to complete the two-stage power-up sequence for a USB 2.0 port.
- Note: The device ID passed back in handle is valid and can be passed to ps5000aChangePowerSource.
4.38 ps5000aOpenUnitAsync – open a scope device without blocking

```c
PICO_STATUS ps5000aOpenUnitAsync
(
    int16_t * status,
    int8_t * serial,
    PS5000A_DEVICE_RESOLUTION resolution
)
```

This function opens a scope without blocking the calling thread. You can find out when it has finished by periodically calling `ps5000aOpenUnitProgress` until that function returns a non-zero value.

**Applicability**
All modes

**Arguments**

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

* `serial`, see `ps5000aOpenUnit`.

* `resolution`, determines the resolution of the device when opened, the available values are one of the `PS5000A_DEVICE_RESOLUTION`. If resolution is out of range, the function will return `PICO_INVALID_DEVICE_RESOLUTION`.

**Returns**
PICO_OK or other code from `PicoStatus.h`
See `ps5000aOpenUnit` for more details.
4.39 ps5000aOpenUnitProgress – check progress of OpenUnit call

```c
PICO_STATUS ps5000aOpenUnitProgress
(
    int16_t * handle,
    int16_t * progressPercent,
    int16_t * complete
);
```

This function checks on the progress of a request made to `ps5000aOpenUnitAsync` to open a scope.

If the function returns `PICO_POWER_SUPPLY_NOT_CONNECTED` or `PICO_USB3_0_DEVICE_NON_USB3_0_PORT`, call `ps5000aChangePowerSource` to select a new power source.

**Applicability**

Use after `ps5000aOpenUnitAsync`

**Arguments**

* handle, see `ps5000aOpenUnit`. This handle is valid only if the function returns `PICO_OK`.

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

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

**Returns**

`PICO_OK` or other code from `PicoStatus.h`
4.40  ps5000aPingUnit – check communication with device

```c
PICO_STATUS ps5000aPingUnit
(
    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.

<table>
<thead>
<tr>
<th>Applicability</th>
</tr>
</thead>
<tbody>
<tr>
<td>All modes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle, the device identifier returned by <code>ps5000aOpenUnit</code>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>PICO_OK or other code from <code>PicoStatus.h</code>.</td>
</tr>
</tbody>
</table>
4.41 ps5000aQueryOutputEdgeDetect – check if output edge detection is enabled

```c
PICO_STATUS ps5000aQueryOutputEdgeDetect
(
    int16_t handle,
    int16_t * state
)
```

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

To switch output edge detection mode on or off, use `ps5000aSetOutputEdgeDetect`. See that function description for more details.

**Applicability**
All modes

**Arguments**
- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `state`, on exit, the state of output edge detection:
  - `0` = off
  - `1` = on

**Returns**
PICO_OK or other code from PicoStatus.h
4.42 ps5000aRunBlock – start block mode

```c
PICO_STATUS ps5000aRunBlock(
    int16_t handle,
    int32_t noOfPreTriggerSamples,
    int32_t noOfPostTriggerSamples,
    uint32_t timebase,
    int32_t * timeIndisposedMs,
    int32_t segmentIndex,
    ps5000aBlockReady lpReady,
    void * pParameter
)
```

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

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

**Applicability**

Block mode, rapid block mode

**Arguments**

- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `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:
  
  \[ \text{noOfPreTriggerSamples} + \text{noOfPostTriggerSamples} \]

- `timebase`, a number in the range 0 to \(2^{32} - 1\). See the [guide to calculating timebase values](#).
- `* timeIndisposedMs`, on exit, the time, in milliseconds, that the scope will spend collecting samples. This does not include any auto trigger timeout. If this pointer is null, nothing will be written here.
- `segmentIndex`, zero-based, specifies which memory segment to use.
- `lpReady`, a pointer to the `ps5000aBlockReady` callback function that the driver will call when the data has been collected. To use the `ps5000aIsReady` polling method instead of a callback function, set this pointer to NULL.
- `* pParameter`, a void pointer that is passed to the `ps5000aBlockReady` callback function. The callback can use this pointer to return arbitrary data to the application.

**Returns**

PICO_OK or other code from PicoStatus.h
4.43 ps5000aRunStreaming – start streaming mode

```c
PICO_STATUS ps5000aRunStreaming(
    int16_t handle,
    uint32_t * sampleInterval, PS5000A_TIME_UNITS sampleIntervalTimeUnits,
    uint32_t maxPreTriggerSamples,
    uint32_t maxPostTriggerSamples,
    int16_t autoStop,
    uint32_t downSampleRatio, PS5000A_RATIO_MODE downSampleRatioMode,
    uint32_t overviewBufferSize
)
```

This function tells the oscilloscope to start collecting data in streaming mode. When data has been collected from the device it is downsampeld if necessary and then delivered to the application. Call `ps5000aGetStreamingLatestValues` to retrieve the data. See Using streaming mode for a step-by-step guide to this process.

The function always starts collecting data immediately, regardless of the trigger settings. Whether a trigger is set or not, the total number of samples stored in the driver is always `maxPreTriggerSamples + maxPostTriggerSamples`.

**Applicability**

**Streaming mode**

**Arguments**

- `handle`, the device identifier returned by `ps5000aOpenUnit`.

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

- `sampleIntervalTimeUnits`, the unit of time used for `sampleInterval`. See `PS5000A_TIME_UNITS`.

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

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

- `autoStop`, a flag that specifies if the streaming should stop when all of `maxSamples = maxPreTriggerSamples + maxPostTriggerSamples` have been captured and a trigger event has occurred. If no trigger event occurs or no trigger is set, streaming will continue until stopped by `ps5000aStop`. If `autoStop` is false, the scope will collect data continuously using the buffer as a first-in-first-out (FIFO) memory.

- `downSampleRatio, downSampleRatioMode`: see `ps5000aGetValues`.

- `overviewBufferSize`, the length of the overview buffers. These are temporary buffers used for storing the data before returning it to the application. The length is the same as the `bufferLth` value passed to `ps5000aSetDataBuffer`.

**Returns**

PICO_OK or other code from `PicoStatus.h`
4.44 ps5000aSetAutoTriggerMicroSeconds – set auto-trigger timeout

```c
PICO_STATUS ps5000aSetAutoTriggerMicroSeconds
    (int16_t handle,
     uint64_t autoTriggerMicroseconds)
```

This function sets up the auto-trigger function, which starts a capture if no trigger event occurs within a specified time after a Run command has been issued.

**Applicability**

All modes

**Arguments**

- `handle`, the device identifier returned by `ps5000aOpenUnit`
- `autoTriggerMicroseconds`, the number of microseconds for which the scope device will wait for a trigger before timing out. If this argument is zero, the scope device will wait indefinitely for a trigger. Otherwise, its behavior depends on the sampling mode:
  - In **block mode**, the capture cannot finish until a trigger event or auto-trigger timeout has occurred.
  - In **streaming mode** the device always starts collecting data as soon as `ps5000aRunStreaming` is called but does not start counting post-trigger samples until it detects a trigger event or auto-trigger timeout.

**Returns**

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

Copyright © 2013–2018 Pico Technology Ltd. All rights reserved.
4.45 ps5000aSetBandwidthFilter – specifies the bandwidth limit

```c
PICO_STATUS ps5000aSetBandwidthFilter(
    int16_t handle,
    PS5000A_CHANNEL channel,
    PS5000A_BANDWIDTH_LIMITER bandwidth
);
```

This function controls the hardware bandwidth limiter fitted to each analog input channel. It does not apply to digital input channels on mixed-signal scopes.

<table>
<thead>
<tr>
<th>Applicability</th>
</tr>
</thead>
<tbody>
<tr>
<td>All modes and models.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle, the device identifier returned by <code>ps5000aOpenUnit</code>.</td>
</tr>
<tr>
<td>channel, the channel to be configured (analog channel A, B, C or D only). See <code>PS5000A_CHANNEL</code>.</td>
</tr>
<tr>
<td>bandwidth, the required bandwidth (full or limited to 20 MHz). See <code>PS5000A_BANDWIDTH_LIMITER</code>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>PICO_OK or other code from <code>PicoStatus.h</code>.</td>
</tr>
</tbody>
</table>

### 4.45.1 PS5000A_BANDWIDTH_LIMITER enumerated type

```c
typedef enum enPS5000ABandwidthLimiter{
    PS5000A_BW_FULL,
    PS5000A_BW_20MHZ,
} PS5000A_BANDWIDTH_LIMITER;
```

<table>
<thead>
<tr>
<th>Applicability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calls to <code>ps5000aSetBandwidthFilter</code>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS5000A_BW_FULL – use the scope's full specified bandwidth</td>
</tr>
<tr>
<td>PS5000A_BW_20MHZ – enable the hardware 20 MHz bandwidth limiter</td>
</tr>
</tbody>
</table>
ps5000aSetChannel – set up input channels

```c
PICO_STATUS ps5000aSetChannel(
    int16_t handle,
    PS5000A_CHANNEL channel,
    int16_t enabled,
    PS5000A_COUPLING type,
    PS5000A_RANGE range,
    float analogueOffset
)
```

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

**Applicability**
All modes.
Analog channels only. For digital channels, use `ps5000aSetDigitalPort`.

**Arguments**
- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `channel`, the channel to be configured. See `PS5000A_CHANNEL` (only the CHANNEL_A to CHANNEL_D values apply).
- `enabled`, whether or not to enable the channel. The values are:
  - 0: disable
  - 1: enable

  **Note 1:** When you open a device, all channels are enabled by default.
- `type`, the impedance and coupling type. See `PS5000A_COUPLING`.
- `range`, the input voltage range. See `PS5000A_RANGE`.
- `analogueOffset`, a voltage to add to the input channel before digitization. The allowable range of offsets depends on the input range selected for the channel, as obtained from `ps5000aGetAnalogueOffset`.

**Returns**
- `PICO_OK` or other code from `PicoStatus.h`
4.46.1 PS5000A_CHANNEL enumerated type

```c
typedef enum enPS5000AChannel {
    PS5000A_CHANNEL_A,
    PS5000A_CHANNEL_B,
    PS5000A_CHANNEL_C,
    PS5000A_CHANNEL_D,
    PS5000A_EXTERNAL,
    PS5000A_MAX_CHANNELS = PS5000A_EXTERNAL,
    PS5000A_TRIGGER_AUX,
    PS5000A_MAX_TRIGGER_SOURCES,
    PS5000A_DIGITAL_PORT0 = 0x80,
    PS5000A_DIGITAL_PORT1,
    PS5000A_DIGITAL_PORT2,
    PS5000A_DIGITAL_PORT3,
    PS5000A_PULSE_WIDTH_SOURCE = 0x10000000
} PS5000A_CHANNEL;
```

These values allow you to specify an input channel, 8-bit digital port or other input.

**Applicability**

All devices.
Not all values apply to all functions - see the description of the calling function for details.

**Values**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS5000A_CHANNEL_A</td>
<td>analog channel A</td>
</tr>
<tr>
<td>PS5000A_CHANNEL_B</td>
<td>analog channel B</td>
</tr>
<tr>
<td>PS5000A_CHANNEL_C</td>
<td>analog channel C (4-channel scopes only)</td>
</tr>
<tr>
<td>PS5000A_CHANNEL_D</td>
<td>analog channel D (4-channel scopes only)</td>
</tr>
<tr>
<td>PS5000A_EXTERNAL</td>
<td>external trigger input (not on MSOs)</td>
</tr>
<tr>
<td>PS5000A_TRIGGER_AUX</td>
<td>reserved</td>
</tr>
<tr>
<td>PS5000A_DIGITAL_PORT0</td>
<td>digital channels 0–7 (MSOs only)</td>
</tr>
<tr>
<td>PS5000A_DIGITAL_PORT1</td>
<td>digital channels 8–15 (MSOs only)</td>
</tr>
<tr>
<td>PS5000A_DIGITAL_PORT2</td>
<td>reserved</td>
</tr>
<tr>
<td>PS5000A_DIGITAL_PORT3</td>
<td>reserved</td>
</tr>
<tr>
<td>PS5000A_PULSE_WIDTH_SOURCE</td>
<td>pulse width qualifier*</td>
</tr>
</tbody>
</table>

* For use as a trigger source by functions such as `ps5000aSetTriggerChannelPropertiesV2`
4.47 ps5000aSetDataBuffer – register data buffer with driver

```c
PICO_STATUS ps5000aSetDataBuffer
(
    int16_t handle,
    PS5000A_CHANNEL source,
    int16_t * buffer,
    int32_t bufferLth,
    uint32_t segmentIndex,
    PS5000A_RATIO_MODE mode
)
```

This function tells the driver where to store the data, either unprocessed or downsampled, that will be returned after the next call to one of the GetValues functions. The function allows you to specify only a single buffer, so for aggregation mode, which requires two buffers, call `ps5000aSetDataBuffers` instead. You must allocate memory for the buffer before calling this function.

**Applicability**

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

**Arguments**

- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `channel`, the channel or port for which you want to set the buffers. See `PS5000A_CHANNEL`.
- `buffer`, pointer to the buffer. Each sample written to the buffer will be a 16-bit ADC count scaled according to the selected voltage range.
- `bufferLth`, the length of the buffer array.
- `segmentIndex`, the number of the memory segment to be used.
- `mode`, the downsampling mode. See `ps5000aGetValues` for the available modes, but note that a single call to `ps5000aSetDataBuffer` can only associate one buffer with one downsampling mode. If you intend to call `ps5000aGetValues` with more than one downsampling mode activated, then you must call `ps5000aSetDataBuffer` several times to associate a separate buffer with each downsampling mode.

**Returns**

- PICO_OK or other code from `PicoStatus.h`
4.48 ps5000aSetDataBuffers – register aggregated data buffers with driver

```c
PICO_STATUS ps5000aSetDataBuffers
(
    int16_t handle,
    PS5000A_CHANNEL source,
    int16_t  * bufferMax,
    int16_t  * bufferMin,
    int32_t  bufferLth,
    uint32_t segmentIndex,
    PS5000A_RATIO_MODE mode
)
```

This function tells the driver the location of one or two buffers for receiving data. You need to allocate memory for the buffers before calling this function. If you do not need two buffers, because you are not using aggregate mode, then you can optionally use `ps5000aSetDataBuffer` instead.

**Applicability**

Block and streaming modes with aggregation.

**Arguments**

- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `channel`, see `ps5000aSetDataBuffer`.
- `bufferMax`, a user-allocated buffer to receive the maximum data values in aggregation mode, or the non-aggregated values otherwise. Each value is a 16-bit ADC count scaled according to the selected voltage range.
- `bufferMin`, a user-allocated buffer to receive the minimum data values in aggregation mode. Not normally used in other modes, but you can direct the driver to write non-aggregated values to this buffer by setting `bufferMax` to NULL. To enable aggregation, the downsampling ratio and mode must be set appropriately when calling one of the `ps5000aGetValues...()` functions.
- `bufferLth`, the length of the `bufferMax` and `bufferMin` arrays.
- `segmentIndex`, the number of the memory segment to be used.
- `mode`, see `ps5000aGetValues`

**Returns**

`PICO_OK` or other code from `PicoStatus.h`
4.49 ps5000aSetDeviceResolution – set the hardware resolution

```c
PICO_STATUS ps5000aSetDeviceResolution
(
    int16_t handle,
    PS5000A_DEVICE_RESOLUTION resolution
)
```

This function sets the sampling resolution of the device. At 12-bit and higher resolutions, the maximum capture buffer length is half that of 8-bit mode. When using 15-bit resolution only 2 channels can be enabled to capture data, and when using 16-bit resolution only one channel is available.

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

After changing the resolution and before calling `ps5000aRunBlock` or `ps5000aRunStreaming`, call `ps5000aSetChannel` to set up the input channels.

**Applicability**
All modes.

**Arguments**

*handle*, the device identifier returned by `ps5000aOpenUnit`.

*resolution*, determines the resolution of the device when opened, the available values are one of the `PS5000A_DEVICE_RESOLUTION`. If resolution is out of range the device will return `PICO_INVALID_DEVICE_RESOLUTION`.

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

### 4.49.1 PS5000A_DEVICE_RESOLUTION enumerated type

```c
typedef enum enPS5000ADeviceResolution
{
    PS5000A_DR_8BIT,
    PS5000A_DR_12BIT,
    PS5000A_DR_14BIT,
    PS5000A_DR_15BIT,
    PS5000A_DR_16BIT
} PS5000A_DEVICE_RESOLUTION;
```

These values specify the resolution of the sampling hardware in the oscilloscope. 8-bit mode divides the input voltage range into 256 levels. The number of levels increases as the resolution increases, up to a maximum of 65 536 levels in 16-bit mode.

**Applicability**
Calls to `ps5000aSetDeviceResolution` etc.

**Values**

- PS5000A_DR_8BIT – 8-bit mode
- PS5000A_DR_12BIT – 12-bit mode
- PS5000A_DR_14BIT – 14-bit mode
PS5000A_DR_15BIT  -  15-bit mode  
PS5000A_DR_16BIT  -  16-bit mode
4.50  ps5000aSetDigitalPort – set up digital inputs

```c
PICO_STATUS ps5000aSetDigitalPort
(
    int16_t handle,
    PS5000A_CHANNEL port,
    int16_t enabled,
    int16_t logiclevel
)
```

This function enables or disables a digital port and sets the logic threshold.

In order to use the fastest sampling rates with digital inputs, disable all analog channels. When all analog channels are disabled you must also select 8-bit resolution to allow the digital inputs to operate alone.

<table>
<thead>
<tr>
<th>Applicability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block and streaming modes with aggregation. MSOs only.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle, the device identifier returned by ps5000aOpenUnit.</td>
</tr>
<tr>
<td>port, identifies the port for digital data:</td>
</tr>
<tr>
<td>PS5000A_DIGITAL_PORT0 = 0x80 (digital channels 0–7)</td>
</tr>
<tr>
<td>PS5000A_DIGITAL_PORT1 = 0x81 (digital channels 8–15)</td>
</tr>
<tr>
<td>enabled, whether or not to enable the port. Enabling a digital port allows the scope to collect data from the port and to trigger on the port. The values are:</td>
</tr>
<tr>
<td>0: disable</td>
</tr>
<tr>
<td>1: enable</td>
</tr>
<tr>
<td>logiclevel, the threshold voltage used to distinguish the 0 and 1 states. Range: –32767 (–5 V) to 32767 (+5 V).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>PICO_OK or other code from PicoStatus.h</td>
</tr>
</tbody>
</table>

4.50.1  MSO digital connector

The PicoScope 5000 Series MSOs have a digital input connector in addition to the analog input BNCs. The illustration below shows the 20-pin IDC header plug as you look at the front panel of the device.
4.51  ps5000aSetEts – set up equivalent-time sampling

```c
PICO_STATUS ps5000aSetEts(
    int16_t handle,
    PS5000A_ETS_MODE mode,
    int16_t etsCycles,
    int16_t etsInterleave,
    int32_t * sampleTimePicoseconds)
```

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

**Applicability**

Block mode. Other restrictions are listed in ETS overview.

**Arguments**

- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `mode`, the ETS mode. See `PS5000A_ETS_MODE`.
- `etsCycles`, the number of cycles to store: the computer can then select `etsInterleave` cycles to give the most uniform spread of samples. Maximum value is one of the following:
  - `PS5242A_MAX_ETS_CYCLES`
  - `PS5243A_MAX_ETS_CYCLES`
  - `PS5244A_MAX_ETS_CYCLES`
  - `PS5X44D_MAX_ETS_CYCLES`
  - `PS5X43D_MAX_ETS_CYCLES`
  - `PS5X42D_MAX_ETS_CYCLES`

- `etsInterleave`, the number of waveforms to combine into a single ETS capture. Maximum value is one of the following:
  - `PS5242A_MAX_INTERLEAVE`
  - `PS5243A_MAX_INTERLEAVE`
  - `PS5244A_MAX_INTERLEAVE`
  - `PS5X44D_MAX_ETS_INTERLEAVE`
  - `PS5X43D_MAX_ETS_INTERLEAVE`
  - `PS5X42D_MAX_ETS_INTERLEAVE`

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

**Returns**

PICO_OK or other code from `PicoStatus.h`
4.51.1 PS5000A_ETS_MODE enumerated type

```c
typedef enum enPS5000AEtsMode
{
    PS5000A_ETS_OFF,
    PS5000A_ETS_FAST,
    PS5000A_ETS_SLOW,
    PS5000A_ETS_MODES_MAX
} PS5000A_ETS_MODE;
```

These types specify which type of ETS (equivalent-time sampling) to use.

<table>
<thead>
<tr>
<th>Applicability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calls to <strong>ps5000aSetEts</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS5000A_ETS_OFF, disables ETS.</td>
</tr>
<tr>
<td>PS5000A_ETS_FAST, enables ETS and provides <em>etsCycles</em> of data, which may contain data from previously returned cycles.</td>
</tr>
<tr>
<td>PS5000A_ETS_SLOW, enables ETS and provides fresh data every <em>etsCycles</em>. This mode takes longer to provide each data set, but the data sets are more stable and are guaranteed to contain only new data.</td>
</tr>
</tbody>
</table>
4.52  ps5000aSetEtsTimeBuffer – set up buffer for ETS timings (64-bit)

```c
PICO_STATUS ps5000aSetEtsTimeBuffer
(
    int16_t handle,
    int64_t * buffer,
    int32_t bufferLth
)
```

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

**Applicability**

- **ETS mode** only.
- If your programming language does not support 64-bit data, use the 32-bit version `ps5000aSetEtsTimeBuffers` instead.

**Arguments**

- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `* buffer`, an array of 64-bit words, each representing the time in femtoseconds ($10^{-15}$ s) at which the sample was captured.
- `bufferLth`, the length of the buffer array.

**Returns**

- `PICO_OK` or other code from `PicoStatus.h`
4.53 ps5000aSetEtsTimeBuffers – set up buffer for ETS timings (32-bit)

```
PICO_STATUS ps5000aSetEtsTimeBuffers
(
    int16_t handle,
    uint32_t * timeUpper,
    uint32_t * timeLower,
    int32_t bufferLth
)
```

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

If your programming language supports 64-bit data, you can use `ps5000aSetEtsTimeBuffer` instead.

**Applicability**

ETS mode only.

**Arguments**

- `handle`, the device identifier returned by `ps5000aOpenUnit`.

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

- `* timeLower`, an array of 32-bit words, each representing the lower 32 bits of the time in femtoseconds at which the sample was captured.

- `bufferLth`, the length of the `timeUpper` and `timeLower` arrays.

**Returns**

PICO_OK or other code from `PicoStatus.h`
4.54  ps5000aSetNoOfCaptures – set number of captures to collect in one run

```c
PICO_STATUS ps5000aSetNoOfCaptures(
    int16_t handle,
    uint32_t nCaptures
)
```

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

**Applicability**

Rapid block mode

**Arguments**

- handle, the device identifier returned by ps5000aOpenUnit.
- nCaptures, the number of waveforms to capture in one run.

**Returns**

PICO_OK or other code from PicoStatus.h
4.55 ps5000aSetOutputEdgeDetect – change triggering behavior

```
PICO_STATUS ps5000aSetOutputEdgeDetect
(
    int16_t handle,
    int16_t state
)
```

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

The oscilloscope normally triggers only when the output of the trigger logic function changes state. For example, if the function is "A high AND B high", the oscilloscope triggers when A is high and B changes from low to high, but does not repeatedly trigger when A and B remain high. Calling `ps5000aSetOutputEdgeDetect` with `state = 0` changes this behavior so that the oscilloscope triggers continually while the logic trigger function evaluates to TRUE.

To find out whether output edge detection is enabled, use `ps5000aQueryOutputEdgeDetect`.

**Applicability**

**Rapid block mode**

**Arguments**

- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `state`, the desired state of output edge detection:
  - 0 = off
  - 1 = on

**Returns**

PICO_OK or other code from PicoStatus.h
4.56 ps5000aSetPulseWidthDigitalPortProperties – set digital port pulse width

```c
PICO_STATUS ps5000aSetPulseWidthDigitalPortProperties
(int16_t handle,
PS5000A_DIGITAL_CHANNEL_DIRECTIONS *directions
int16_t nDirections)
```

This function will set the individual digital channels' pulse-width trigger directions. Each trigger direction consists of a channel name and a direction. If the channel is not included in the array of `PS5000A_DIGITAL_CHANNEL_DIRECTIONS`, the driver assumes the digital channel's pulse-width trigger direction is `PS5000A_DIGITAL_DONT_CARE`.

**Applicability**

All modes.
Mixed-signal models only.

**Arguments**

- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `directions`, a pointer to an array of `PS5000A_DIGITAL_CHANNEL_DIRECTIONS` structures describing the requested properties. The array can contain a single element describing the properties of one channel, or a number of elements describing several digital channels. If `directions` is NULL, digital pulse-width triggering is switched off. A digital channel that is not included in the array will be set to `PS5000A_DIGITAL_DONT_CARE`.
- `nDirections`, the number of digital channel directions being passed to the driver.

**Returns**

PICO_OK or other code from `PicoStatus.h`
4.57 ps5000aSetPulseWidthQualifier – set up pulse width triggering

```c
PICO_STATUS ps5000aSetPulseWidthQualifier
(
    int16_t handle,
    PS5000A_PWQ_CONDITIONS * conditions,
    int16_t nConditions,
    PS5000A_THRESHOLD_DIRECTION direction,
    uint32_t lower,
    uint32_t upper,
    PS5000A_PULSE_WIDTH_TYPE type
)
```

**THIS FUNCTION IS NOT RECOMMENDED FOR NEW APPLICATIONS.**
In new applications please use `ps5000aSetPulseWidthQualifierProperties`, `ps5000aSetPulseWidthQualifierConditions` and `ps5000aSetPulseWidthQualifierDirections` instead.

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

**Applicability**
All modes

**Arguments**
handle, the device identifier returned by `ps5000aOpenUnit`.

* conditions, an array of `PS5000A_PWQ_CONDITIONS` structures specifying the conditions that should be applied to each channel. In the simplest case, the array consists of a single element. When there are several elements, the overall trigger condition is the logical OR of all the elements. If `conditions` is `NULL` then the pulse-width qualifier is not used.

nConditions, the number of elements in the `conditions` array. If `nConditions` is zero then the pulse-width qualifier is not used.  
Range: 0 to `PS5000A_MAX_PULSE_WIDTH_QUALIFIER_COUNT`.

direction, the direction of the signal required for the pulse width trigger to fire. See `PS5000A_THRESHOLD_DIRECTION` constants for the list of possible values. Each channel of the oscilloscope (except the Ext input) has two thresholds for each direction—for example, `PS5000A_RISING` and `PS5000A_RISING_LOWER`—so that one can be used for the pulse-width qualifier and the other for the level trigger. The driver will not let you use the same threshold for both triggers; so, for example, you cannot use `PS5000A_RISING` as the direction argument for both `ps5000aSetTriggerConditions` and `ps5000aSetPulseWidthQualifier` at the same time. There is no such restriction when using window triggers.

lower, the lower limit of the pulse-width counter, in samples.

upper, the upper limit of the pulse-width counter, in samples. This parameter is used only when the type is set to `PS5000A_PW_TYPE_IN_RANGE` or `PS5000A_PW_TYPE_OUT_OF_RANGE`.
type, the pulse-width type. See \texttt{PS5000A\_PULSE\_WIDTH\_TYPE}.

Returns

\texttt{PICO\_OK} or other code from \texttt{PicoStatus.h}

4.57.1 PS5000A\_PWQ\_CONDITIONS structure

\begin{verbatim}
typedef struct tPS5000APwqConditions{
    PS5000A\_TRIGGER\_STATE channelA;
    PS5000A\_TRIGGER\_STATE channelB;
    PS5000A\_TRIGGER\_STATE channelC;
    PS5000A\_TRIGGER\_STATE channelD;
    PS5000A\_TRIGGER\_STATE external;
    PS5000A\_TRIGGER\_STATE aux;
} PS5000A\_PWQ\_CONDITIONS
\end{verbatim}

A structure of this type is passed to \texttt{ps5000aSetPulseWidthQualifier} in the conditions argument to specify the pulse-width qualifier conditions for all the trigger sources.

Each structure is the logical AND of the states of the scope's inputs. The \texttt{ps5000aSetPulseWidthQualifier} function can OR together a number of these structures to produce the final pulse width qualifier, which can therefore be any possible Boolean function of the scope's inputs.

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

Applicability

Calls to \texttt{ps5000aSetPulseWidthQualifier}

Elements

channelA, channelB, channelC, channelD, external, the type of condition that should be applied to each channel. See \texttt{PS5000A\_TRIGGER\_STATE}.

The channels that are set to \texttt{PS5000A\_CONDITION\_TRUE} or \texttt{PS5000A\_CONDITION\_FALSE} must all meet their conditions simultaneously to produce a trigger. Channels set to \texttt{PS5000A\_CONDITION\_DONT\_CARE} are ignored.

aux, not used.
4.58 ps5000aSetPulseWidthQualifierConditions – set up pulse width triggering

```c
PICO_STATUS ps5000aSetPulseWidthQualifierConditions(
    int16_t handle,  
    PS5000A_CONDITION * conditions, 
    int16_t nConditions, 
    PS5000A_CONDITIONS_INFO info
)
```

This function applies a condition to the pulse-width qualifier. It can either add the new condition to the existing qualifier, or clear the existing qualifier and replace it with the new condition.

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

<table>
<thead>
<tr>
<th>Applicability</th>
</tr>
</thead>
<tbody>
<tr>
<td>All modes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle, the device identifier returned by ps5000aOpenUnit.</td>
</tr>
<tr>
<td>conditions, a list of PS5000A_CONDITION structures</td>
</tr>
<tr>
<td>nConditions, the number of values in the conditions list</td>
</tr>
<tr>
<td>info, whether to add this condition to the existing definition or clear the definition and start a new one. See PS5000A_CONDITIONS_INFO.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>PICO_OK or other code from PicoStatus.h</td>
</tr>
</tbody>
</table>

4.58.1 PS5000A_CONDITIONS_INFO enumerated type

```c
typdef enum enPS5000AConditionsInfo
{
    PS5000A_CLEAR = 0x00000001,
    PS5000A_ADD = 0x00000002
} PS5000A_CONDITIONS_INFO;
```

When you add a trigger condition, these values specify what to do with any existing trigger conditions that you have previously set up.

<table>
<thead>
<tr>
<th>Applicability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setting trigger conditions</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS5000A_CLEAR – clear existing trigger logic and replace with the new condition</td>
</tr>
</tbody>
</table>

Copyright © 2013–2018 Pico Technology Ltd. All rights reserved.
PS5000A_ADD — add the new condition, using Boolean OR, to the existing trigger logic

You can combine both actions by bitwise OR-ing together the flags and casting to a PS5000A_CONDITIONS_INFO data type:

(PS5000A_CONDITIONS_INFO) (PS5000A_CLEAR | PS5000A_ADD)
4.59 ps5000aSetPulseWidthQualifierDirections – set up pulse width triggering

```
PICO_STATUS ps5000aSetPulseWidthQualifierDirections
(
    int16_t handle,
    PS5000A_DIRECTION * directions,
    int16_t nDirections
)
```

This function specifies the directions for all the trigger sources used with the pulse-width qualifier.

<table>
<thead>
<tr>
<th>Applicability</th>
</tr>
</thead>
<tbody>
<tr>
<td>All modes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle, the device identifier returned by <code>ps5000aOpenUnit</code>.</td>
</tr>
<tr>
<td>directions, a list of <code>PS5000A_DIRECTION</code> structures specifying which direction to apply to each trigger source</td>
</tr>
<tr>
<td>nDirections, the number of items in the directions list</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>PICO_OK or other code from <code>PicoStatus.h</code></td>
</tr>
</tbody>
</table>
4.60  ps5000aSetPulseWidthQualifierProperties – set up pulse width triggering

```c
PICO_STATUS ps5000aSetPulseWidthQualifierProperties(
    int16_t handle,
    uint32_t lower,
    uint32_t upper,
    PS5000A_PULSE_WIDTH_TYPE type
)
```

This function sets up the pulse width timings and logic type of the pulse-width trigger qualifier.

**Applicability**

All modes

**Arguments**

- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `lower`, the lower limit of the pulse-width counter, in samples. This argument is required for all pulse width types.
- `upper`, the upper limit of the pulse-width counter, in samples. This argument is used only when the type is `PS5000A_PW_TYPE_IN_RANGE` or `PS5000A_PW_TYPE_OUT_OF_RANGE`.
- `type`, the type of pulse width trigger. See `PS5000A_PULSE_WIDTH_TYPE`.

**Returns**

PICO_OK or other code from `PicoStatus.h`

### 4.60.1  PS5000A_PULSE_WIDTH_TYPE enumerated type

```c
typedef enum enPS5000APulseWidthType
{
    PS5000A_PW_TYPE_NONE,
    PS5000A_PW_TYPE_LESS_THAN,
    PS5000A_PW_TYPE_GREATER_THAN,
    PS5000A_PW_TYPE_IN_RANGE,
    PS5000A_PW_TYPE_OUT_OF_RANGE
} PS5000A_PULSE_WIDTH_TYPE;
```

These values specify the type of pulse-width trigger. You can require the pulse width to be less than or greater than a specified limit, or between two limits, or not between two limits.

**Applicability**

Pulse-width triggering

**Values**

- `PS5000A_PW_TYPE_NONE` – do not use the pulse width qualifier
- `PS5000A_PW_TYPE_LESS_THAN` – pulse width less than lower
- `PS5000A_PW_TYPE_GREATER_THAN` – pulse width greater than lower.
- `PS5000A_PW_TYPE_IN_RANGE` – pulse width between lower and upper.
- `PS5000A_PW_TYPE_OUT_OF_RANGE` – pulse width not between lower and upper.
4.61 ps5000aSetSigGenArbitrary – set up arbitrary waveform generator

```c
PICO_STATUS ps5000aSetSigGenArbitrary
(``
  int16_t handle,
  int32_t offsetVoltage,
  uint32_t pkToPk,
  int32_t startDeltaPhase,
  int32_t stopDeltaPhase,
  uint32_t deltaPhaseIncrement,
  int32_t dwellCount,
  int16_t * arbitraryWaveform,
  int32_t arbitraryWaveformSize,
  PS5000A_SWEEP_TYPE sweepType,
  PS5000A_EXTRA_OPERATIONS operation,
  PS5000A_INDEX_MODE indexMode,
  uint32_t shots,
  uint32_t sweeps,
  PS5000A_SIGGEN_TRIG_TYPE triggerType,
  PS5000A_SIGGEN_TRIG_SOURCE triggerSource,
  int16_t extInThreshold
`
)
```

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

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

The phase accumulator initially increments by `startDeltaPhase`. If the AWG is set to sweep mode, the phase increment is increased or decreased at specified intervals until it reaches `stopDeltaPhase`. The easiest way to obtain the values of `startDeltaPhase` and `stopDeltaPhase` necessary to generate the desired frequency is to call `ps5000aSigGenFrequencyToPhase`. Alternatively, see Calculating `deltaPhase` below for more information on how to calculate these values.

### Applicability
All modes. B, D and D MSO models only.

### Arguments
- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `offsetVoltage`, the voltage offset, in microvolts, to be applied to the waveform.
- `pkToPk`, the peak-to-peak voltage, in microvolts, of the waveform signal.
  
  Note that if the signal voltages defined by the combination of `offsetVoltage` and `pkToPk` extend outside the voltage range of the signal generator, the output waveform will be clipped.
- `startDeltaPhase`, the initial value added to the phase accumulator as the generator begins to step through the waveform buffer.
stopDeltaPhase, the final value added to the phase accumulator before the generator restarts or reverses the sweep.

deltaPhaseIncrement, the amount added to the delta phase value every time the dwellCount period expires. This determines the amount by which the generator sweeps the output frequency in each dwell period.

dwellCount, the time, in 50 ns steps, between successive additions of deltaPhaseIncrement to the delta phase accumulator. This determines the rate at which the generator sweeps the output frequency. Minimum value: PS5000A_MIN_DWELL_COUNT

arbitraryWaveform, a buffer that holds the waveform pattern as a set of samples equally spaced in time. If pkToPk is set to its maximum (4 V) and offsetVoltage is set to 0, the output range will be [-2 V, +2 V]. Obtain the maximum and minimum allowed sample values by calling ps5000aSigGenArbitraryMinMaxValues.

arbitraryWaveformSize, the length of the arbitrary waveform buffer, in samples. Obtain the minimum and maximum allowed values by calling ps5000aSigGenArbitraryMinMaxValues.

sweepType, determines whether the startDeltaPhase is swept up to the stopDeltaPhase, down to it, or repeatedly up and down. See PS5000A_SWEEP_TYPE.

operation, the type of waveform to be produced. See PS5000A_EXTRA_OPERATIONS.

indexMode, specifies how the signal will be formed from the arbitrary waveform data. Set to PS5000A_SINGLE for single index mode or PS5000A_DUAL for dual index mode. See AWG index modes for details.

shots, sweeps, triggerType, triggerSource, extInThreshold: see ps5000aSigGenBuiltIn

Returns
PICO_OK or other code from PicoStatus.h

4.61.1 PS5000A_INDEX_MODE enumerated type

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

typedef enum enPS5000AIndexMode
{
    PS5000A_SINGLE,
    PS5000A_DUAL,
    PS5000A_QUAD,
    PS5000A_MAX_INDEX_MODES
} PS5000A_INDEX_MODE;
PS5000A_SINGLE: **Single mode.** The generator outputs the raw contents of the buffer repeatedly. This mode is the only one that can generate asymmetrical waveforms. You can also use this mode for symmetrical waveforms, but the dual mode makes more efficient use of the buffer memory.

PS5000A_DUAL: **Dual mode.** The generator outputs the contents of the buffer from beginning to end, and then does a second pass in the reverse direction through the buffer. This allows you to specify only the first half of a waveform with twofold symmetry, such as a Gaussian function, and let the generator fill in the other half.

PS5000A_QUAD: Not used.

4.61.2 Calculating deltaPhase

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

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

where:

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

You can call ps5000aSigGenFrequencyToPhase to calculate the value for deltaPhase for the desired frequency.

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>PicoScope 5242B PicoScope 5442B</th>
<th>PicoScope 5243B PicoScope 5443B</th>
<th>PicoScope 5244B PicoScope 5444B</th>
</tr>
</thead>
<tbody>
<tr>
<td>dacFrequency</td>
<td>200 MHz</td>
<td>200 MHz</td>
<td>200 MHz</td>
</tr>
<tr>
<td>dacPeriod (= 1/dacFrequency)</td>
<td>5 ns</td>
<td>5 ns</td>
<td>5 ns</td>
</tr>
<tr>
<td>phaseAccumulatorSize</td>
<td>4 294 967 296 (2^{52})</td>
<td>4 294 967 296 (2^{52})</td>
<td>4 294 967 296 (2^{52})</td>
</tr>
<tr>
<td>awgBufferSize</td>
<td>16 384 (2^{14})</td>
<td>32 768 (2^{15})</td>
<td>49 152 (3 \times 2^{14})</td>
</tr>
</tbody>
</table>
4.61.3 PS5000A_SWEEP_TYPE enumerated type

typedef enum enPS5000ASweepType
{
    PS5000A_UP,
    PS5000A_DOWN,
    PS5000A_UPDOWN,
    PS5000A_DOWNUP,
    PS5000A_MAX_SWEEP_TYPES
} PS5000A_SWEEP_TYPE;

These values specify the frequency sweep mode of the signal generator or arbitrary waveform generator.

<table>
<thead>
<tr>
<th>Applicability</th>
<th>Signal generator or AWG setup</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS5000A_UP</td>
<td>sweep the frequency from lower limit up to upper limit</td>
</tr>
<tr>
<td>PS5000A_DOWN</td>
<td>sweep the frequency from upper limit down to lower limit</td>
</tr>
<tr>
<td>PS5000A_UPDOWN</td>
<td>sweep the frequency up and then down</td>
</tr>
<tr>
<td>PS5000A_DOWNUP</td>
<td>sweep the frequency down and then up</td>
</tr>
</tbody>
</table>

4.61.4 PS5000A_EXTRA_OPERATIONS enumerated type

typedef enum enPS5000AExtraOperations
{
    PS5000A_ES_OFF,
    PS5000A_WHITENOISE,
    PS5000A_PRBS
} PS5000A_EXTRA_OPERATIONS;

These values specify additional signal types for the signal generator.

<table>
<thead>
<tr>
<th>Applicability</th>
<th>Signal generator or AWG setup</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS5000A_ES_OFF</td>
<td>normal signal generator operation specified by wavetype, or normal AWG operation.</td>
</tr>
<tr>
<td>PS5000A_WHITENOISE</td>
<td>produces white noise and ignores all settings except pkToPk and offsetVoltage.</td>
</tr>
<tr>
<td>PS5000A_PRBS</td>
<td>produces a pseudorandom binary sequence with a bit rate specified by the start and stop frequencies.</td>
</tr>
</tbody>
</table>
4.62 ps5000aSetSigGenBuiltIn – set up standard signal generator

```
PICO_STATUS ps5000aSetSigGenBuiltIn
(
    int16_t handle,
    int32_t offsetVoltage,
    uint32_t pkToPk,
    PS5000A_WAVE_TYPE waveType,
    float startFrequency,
    float stopFrequency,
    float increment,
    float dwellTime,
    PS5000A_SWEEP_TYPE sweepType,
    PS5000A_EXTRA_OPERATIONS operation,
    uint32_t shots,
    uint32_t sweeps,
    PS5000A_SIGGEN_TRIG_TYPE triggerType,
    PS5000A_SIGGEN_TRIG_SOURCE triggerSource,
    int16_t extInThreshold
)
```

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

**Applicability**
All models

**Arguments**
- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `offsetVoltage`, the voltage offset, in microvolts, to be applied to the waveform.
- `pkToPk`, the peak-to-peak voltage, in microvolts, of the waveform signal. Note that if the signal voltages described by the combination of `offsetVoltage` and `pkToPk` extend outside the voltage range of the signal generator, the output waveform will be clipped.
- `waveType`, the type of waveform to be generated. See `PS5000A_WAVE_TYPE`.
- `startFrequency`, the frequency that the signal generator will initially produce.

Minimum value:
- `PS5000A_MIN_FREQUENCY`

Maximum value (depends on wave type):
- `PS5000A_SINE_MAX_FREQUENCY`
- `PS5000A_SQUARE_MAX_FREQUENCY`
- `PS5000A_TRIANGLE_MAX_FREQUENCY`
- `PS5000A_SINC_MAX_FREQUENCY`
- `PS5000A_RAMP_MAX_FREQUENCY`
- `PS5000A_HALF_SINE_MAX_FREQUENCY`
- `PS5000A_GAUSSIAN_MAX_FREQUENCY`
stopFrequency, the frequency at which the sweep reverses direction or returns to the initial frequency. For allowable values, see startFrequency.

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

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

sweepType, whether the frequency will sweep from startFrequency to stopFrequency, in the opposite direction, or repeatedly reverse direction. See PS5000A_SWEEP_TYPE.

operation, the type of waveform to be produced, specified by one of the following enumerated types (not 5000A models). See PS5000A_EXTRA_OPERATIONS.

shots, 0: sweep the frequency as specified by sweeps
  1..PS5000A_MAX_SWEEPS_SHOTS: the number of cycles of the waveform to be produced after a trigger event. sweeps must be zero.
  PS5000A_SHOT_SWEEP_TRIGGER_CONTINUOUS_RUN: start and run continuously after trigger occurs

sweeps, 0: produce number of cycles specified by shots
  1..PS5000A_MAX_SWEEPS_SHOTS: the number of times to sweep the frequency after a trigger event, according to sweepType. shots must be zero.
  PS5000A_SHOT_SWEEP_TRIGGER_CONTINUOUS_RUN: start a sweep and continue after trigger occurs

triggerType, the type of trigger (edge or level) that will be applied to the signal generator. See PS5000A_SIGGEN_TRIG_TYPE. If a gated trigger is used, either shots or sweeps, but not both, must be non-zero.

triggerSource, the source that will trigger the signal generator. See PS5000A_SIGGEN_TRIG_SOURCE. If a trigger source other than P5000A_SIGGEN_NONE is specified, either shots or sweeps, but not both, must be non-zero.

extInThreshold, used to set trigger level for external trigger.

Returns
PICO_OK or other code from PicoStatus.h

4.62.1 PS5000A_SIGGEN_TRIG_TYPE enumerated type

typedef enum enPS5000ASigGenTrigType
{
  PS5000A_SIGGEN_RISING,
  PS5000A_SIGGEN_FALLING,
  PS5000A_SIGGEN_GATE_HIGH,
  PS5000A_SIGGEN_GATE_LOW
} PS5000A_SIGGEN_TRIG_TYPE;

These values specify how triggering of the signal generator or arbitrary waveform generator works. The signal generator can be started by a rising or falling edge on the trigger signal or can be gated to run while the trigger signal is high or low. The gated trigger remembers the phase of the waveform when the trigger signal goes inactive and resumes the waveform from the same phase when the trigger signal goes active again.
**Applicability**

Signal generator or AWG setup

**Values**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS5000A_SIGGEN_RISING</td>
<td>trigger on rising edge</td>
</tr>
<tr>
<td>PS5000A_SIGGEN_FALLING</td>
<td>trigger on falling edge</td>
</tr>
<tr>
<td>PS5000A_SIGGEN_GATE_HIGH</td>
<td>run while trigger is high</td>
</tr>
<tr>
<td>PS5000A_SIGGEN_GATE_LOW</td>
<td>run while trigger is low</td>
</tr>
</tbody>
</table>

### 4.62.2 PS5000A_SIGGEN_TRIG_SOURCE enumerated type

```c
typedef enum enPS5000ASigGenTrigSource
{
    PS5000A_SIGGEN_NONE,
    PS5000A_SIGGEN_SCOPE_TRIG,
    PS5000A_SIGGEN_AUX_IN,
    PS5000A_SIGNAL_EXT_IN,
    PS5000A_SIGGEN_SOFT_TRIG
} PS5000A_SIGNAL_TRIG_SOURCE;
```

These values specify how triggering of the signal generator or arbitrary waveform generator works. The signal generator can be started by a rising or falling edge on the trigger signal or can be gated to run while the trigger signal is high or low.

**Applicability**

Signal generator or AWG setup

**Values**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS5000A_SIGGEN_NONE</td>
<td>run without waiting for trigger</td>
</tr>
<tr>
<td>PS5000A_SIGNAL_SCOPE_TRIG</td>
<td>use scope trigger</td>
</tr>
<tr>
<td>PS5000A_SIGNAL_EXT_IN</td>
<td>use EXT input</td>
</tr>
<tr>
<td>PS5000A_SIGGEN_SOFT_TRIG</td>
<td>wait for software trigger provided by</td>
</tr>
<tr>
<td></td>
<td>ps5000aSigGenSoftwareControl</td>
</tr>
</tbody>
</table>

### 4.62.3 PS5000A_WAVE_TYPE enumerated type

```c
typedef enum enPS5000AWaveType
{
    PS5000A_SINE,
    PS5000A_SQUARE,
    PS5000A_TRIANGLE,
    PS5000A_RAMP_UP,
    PS5000A_RAMP_DOWN,
    PS5000A_SINC,
    PS5000A_GAUSSIAN,
    PS5000A_HALF_SINE,
    PS5000A_DC_VOLTAGE,
    PS5000A_WHITE_NOISE,
    PS5000A_MAX_WAVE_TYPES
} PS5000A_WAVE_TYPE;
```

These values specify which standard waveform is produced by the signal generator.
## Applicability
Signal generator setup

<table>
<thead>
<tr>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS5000A_SINE</td>
<td>sine wave</td>
</tr>
<tr>
<td>PS5000A_SQUARE</td>
<td>square wave</td>
</tr>
<tr>
<td>PS5000A_TRIANGLE</td>
<td>triangle wave</td>
</tr>
<tr>
<td>PS5000A_DC_VOLTAGE</td>
<td>DC voltage</td>
</tr>
<tr>
<td>PS5000A_RAMP_UP</td>
<td>rising sawtooth (not 5000A Series)</td>
</tr>
<tr>
<td>PS5000A_RAMP_DOWN</td>
<td>falling sawtooth (not 5000A Series)</td>
</tr>
<tr>
<td>PS5000A_SINC</td>
<td>sin (x)/x (not 5000A Series)</td>
</tr>
<tr>
<td>PS5000A_GAUSSIAN</td>
<td>Gaussian (not 5000A Series)</td>
</tr>
<tr>
<td>PS5000A_HALF_SINE</td>
<td>half (full-wave rectified) sine (not 5000A Series)</td>
</tr>
</tbody>
</table>
4.63 ps5000aSetSigGenBuiltInV2 – high-precision signal generator setup

```c
PICO_STATUS ps5000aSetSigGenBuiltInV2
(
    int16_t handle,
    int32_t offsetVoltage,
    uint32_t pkToPk,
    PS5000A_WAVE_TYPE waveType,
    double startFrequency,
    double stopFrequency,
    double increment,
    double dwellTime,
    PS5000A_SWEEP_TYPE sweepType,
    PS5000A_EXTRA_OPERATIONS operation,
    uint32_t shots,
    uint32_t sweeps,
    PS5000A_SIGGEN_TRIG_TYPE triggerType,
    PS5000A_SIGGEN_TRIG_SOURCE triggerSource,
    int16_t extInThreshold
)
```

This function allows you to set the signal generator frequency with double precision. In all other respects it is the same as `ps5000aSetSigGenBuiltIn`.

**Applicability**
All models

**Arguments**
See `ps5000aSetSigGenBuiltIn`

**Returns**
PICO_OK or other code from PicoStatus.h
4.64  ps5000aSetSigGenPropertiesArbitrary – change AWG settings

```c
PICO_STATUS ps5000aSetSigGenPropertiesArbitrary
(
    int16_t     handle,
    uint32_t    startDeltaPhase,
    uint32_t    stopDeltaPhase,
    uint32_t    deltaPhaseIncrement,
    uint32_t    dwellCount,
    PS5000A_SWEEP_TYPE sweepType,
    uint32_t    shots,
    uint32_t    sweeps,
    PS5000A_SIGGEN_TRIG_TYPE triggerType,
    PS5000A_SIGGEN_TRIG_SOURCE triggerSource,
    int16_t     extInThreshold
)
```

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

<table>
<thead>
<tr>
<th>Applicability</th>
</tr>
</thead>
<tbody>
<tr>
<td>All modes.</td>
</tr>
<tr>
<td>PicoScope 5000B, 5000D and 5000D MSO models only.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>See <code>ps5000aSetSigGenArbitrary</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>PICO_OK or other code from <code>PicoStatus.h</code></td>
</tr>
</tbody>
</table>
4.65  ps5000aSetSigGenPropertiesBuiltIn – change function generator settings

```c
PICO_STATUS ps5000aSetSigGenPropertiesBuiltIn
(
    int16_t handle,
    double startFrequency,
    double stopFrequency,
    double increment,
    double dwellTime,
    PS5000A_SWEEP_TYPE sweepType,
    uint32_t shots,
    uint32_t sweeps,
    PS5000A_SIGGEN_TRIG_TYPE triggerType,
    PS5000A_SIGGEN_TRIG_SOURCE triggerSource,
    int16_t extInThreshold
)
```

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

**Applicability**

All modes

**Arguments**

See `ps5000aSetSigGenBuiltIn`

**Returns**

`PICO_OK` or other code from `PicoStatus.h`
4.66 ps5000aSetSimpleTrigger – set edge or level trigger

```c
PICO_STATUS ps5000aSetSimpleTrigger(
    int16_t handle,
    int16_t enable,
    PS5000A_CHANNEL source,
    int16_t threshold,
    PS5000A_THRESHOLD_DIRECTION direction,
    uint32_t delay,
    int16_t autoTrigger_ms
)
```

This function simplifies arming the trigger, removing the need to call the three trigger functions `ps5000aSetTriggerChannelPropertiesV2`, `ps5000aSetTriggerChannelConditionsV2` and `ps5000aSetTriggerChannelDirectionsV2` individually. It supports only the edge and level trigger types (not window triggers), only the analog and external trigger input channels, 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 `ps5000aOpenUnit`.
- `enable`, zero to disable the trigger, any non-zero value to set the trigger.
- `source`, the channel on which to trigger (CHANNEL_A to CHANNEL_D and EXTERNAL values only).
- `threshold`, the ADC count at which the trigger will fire.
- `direction`, the direction in which the signal must move to cause a trigger. The following directions are supported: ABOVE, BELOW, RISING, FALLING and RISING_OR_FALLING.
- `delay`, the time between the trigger occurring and the first sample. For example, if `delay = 100`, the scope would wait 100 sample periods before sampling. At a `timebase` of 500 MS/s, or 2 ns per sample, the total delay would then be `100 x 2 ns = 200 ns`. Range: 0 to `MAX_DELAY_COUNT`.
- `autoTrigger_ms`, the number of milliseconds after which the device starts capturing if no trigger occurs. If this is set to zero, the scope device waits indefinitely for a trigger. The value passed here overrides any value set by calling `ps5000aSetAutoTriggerMicroSeconds`. For greater precision, call `ps5000aSetAutoTriggerMicroSeconds` after calling this function.

**Returns**

- `PICO_OK` or other code from `PicoStatus.h`
4.67 ps5000aSetTriggerChannelConditions – specify which channels to trigger on

```c
PICO_STATUS ps5000aSetTriggerChannelConditions
(
    int16_t handle,
    PS5000A_TRIGGER_CONDITIONS * conditions,
    int16_t nConditions
)
```

**This function is not recommended for new applications.**
In new applications please use `ps5000aSetTriggerChannelConditionsV2` instead.

This function sets up trigger conditions on the scope's inputs. The trigger is defined by one or more `PS5000A_TRIGGER_CONDITIONS` structures that are then ORed together. Each structure is itself the AND of the states of one or more of the inputs. This AND-OR logic allows you to create any possible Boolean function of the scope's inputs.

If complex triggering is not required, use `ps5000aSetSimpleTrigger`.

### Applicability
All modes

### Arguments
- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `* conditions`, an array of `PS5000A_TRIGGER_CONDITIONS` structures specifying the conditions that should be applied to each channel. In the simplest case, the array consists of a single element. When there is more than one element, the overall trigger condition is the logical OR of all the elements.
- `nConditions`, the number of elements in the `conditions` array. If `nConditions` is zero, triggering is switched off.

### Returns
- `PICO_OK` or other code from `PicoStatus.h`

#### 4.67.1 PS5000A_TRIGGER_CONDITIONS structure

```c
typedef struct tPS5000ATriggerConditions {
    PS5000A_TRIGGER_STATE channelA;
    PS5000A_TRIGGER_STATE channelB;
    PS5000A_TRIGGER_STATE channelC;
    PS5000A_TRIGGER_STATE channelD;
    PS5000A_TRIGGER_STATE external;
    PS5000A_TRIGGER_STATE aux;
    PS5000A_TRIGGER_STATE pulseWidthQualifier;
} PS5000A_TRIGGER_CONDITIONS
```

A structure of this type is passed to `ps5000aSetTriggerChannelConditions` in the `conditions` argument to specify a set of trigger conditions across all input channels.
Each structure is the logical AND of all available trigger sources. The `ps5000aSetTriggerChannelConditions` function can OR together a number of these structures to produce the final trigger condition, which can therefore be any possible Boolean function of the scope’s inputs.

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

**Elements**

- `channelA`, `channelB`, `channelC`, `channelD`, `external`, `pulseWidthQualifier`, the type of condition that should be applied to each channel.
- See `PS5000A_TRIGGER_STATE`.

The channels that are set to `PS5000ACONDITION_TRUE` or `PS5000ACONDITION_FALSE` must all meet their conditions simultaneously to produce a trigger. Channels set to `PS5000ACONDITION_DONT_CARE` are ignored.

`aux`, not used.
4.68 ps5000aSetTriggerChannelConditionsV2 – specify which channels to trigger on

PICO_STATUS ps5000aSetTriggerChannelConditionsV2
(
    int16_t handle,
    PS5000A_CONDITION * conditions,
    int16_t nConditions,
    PS5000A_CONDITIONS_INFO info
)

The trigger is set up by defining an array of one or more PS5000A_CONDITION structures that are then ANDed together. The function can be called multiple times, in which case the trigger logic is ORed with that defined by previous calls. This AND-OR logic allows you to create almost any † Boolean function of the scope's inputs. To cease ORing trigger channel conditions and start again with a new set, call with info = PS5000A_CLEAR.

If you only need to trigger on a single analog input with edge or level detection, it's easier to use ps5000aSetSimpleTrigger.

**Applicability**

All modes

**Arguments**

* handle, the device identifier returned by ps5000aOpenUnit.

* conditions, an array of PS5000A_CONDITION structures specifying the conditions that should be applied to each channel. In the simplest case, the array consists of a single element. When there is more than one element, the overall trigger condition is the logical OR of all the elements.

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

* info, specifies whether to clear the existing conditions or add the current condition to them using logical OR. See PS5000A_CONDITIONS_INFO.

**Returns**

PICO_OK or other code from PicoStatus.h

† There is a restriction that applies only in rare cases with 5000D MSO devices. If you apply trigger conditions to all six channels (analog A to D and digital ports P0 and P1), there is a restriction on how P0 and P1 may appear in your Boolean expression. You can implement an arbitrary function f(A,B,C,D,P0|P1) or f(A,B,C,D,P0&P1), but not of any other combination of P0 and P1 or of the ports on their own.

### 4.68.1 PS5000A_CONDITION structure

typedef struct tPS5000ACondition
{
    PS5000A_CHANNEL source;
    PS5000A_TRIGGER_STATE condition;
} PS5000A_CONDITION
A structure of this type is passed to `ps5000aSetTriggerChannelConditionsV2` in the `conditions` argument to specify the trigger conditions.

Each structure defines a condition to apply to one of the scope's input channels or ports. The `ps5000aSetTriggerChannelConditionsV2` function can OR together a number of these structures to produce the final trigger condition.

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

<table>
<thead>
<tr>
<th>Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>source – the channel or digital port used as the trigger source.</td>
</tr>
<tr>
<td>condition – how the source condition contributes to the overall trigger logic. See <code>PS5000A_TRIGGER_STATE</code>.</td>
</tr>
</tbody>
</table>

### 4.68.2 PS5000A_TRIGGER_STATE enumerated type

```c
typedef enum enPS5000ATriggerState
{
    PS5000A_CONDITION_DONT_CARE,
    PS5000A_CONDITION_TRUE,
    PS5000A_CONDITION_FALSE,
    PS5000A_CONDITION_MAX
} PS5000A_TRIGGER_STATE;
```

These values specify how each trigger condition is combined with the overall trigger logic.

<table>
<thead>
<tr>
<th>Applicability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setting trigger conditions</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PS5000A_CONDITION_DONT_CARE</code> – the source condition has no effect on the logic</td>
</tr>
<tr>
<td><code>PS5000A_CONDITION_TRUE</code> – the source condition must be true</td>
</tr>
<tr>
<td><code>PS5000A_CONDITION_FALSE</code> – the source condition must be false</td>
</tr>
</tbody>
</table>
4.69 ps5000aSetTriggerChannelDirections – set up signal polarities for triggering

```c
PICO_STATUS ps5000aSetTriggerChannelDirections
(
    int16_t handle,
    PS5000A_THRESHOLD_DIRECTION channelA,
    PS5000A_THRESHOLD_DIRECTION channelB,
    PS5000A_THRESHOLD_DIRECTION channelC;
    PS5000A_THRESHOLD_DIRECTION channelD;
    PS5000A_THRESHOLD_DIRECTION ext,
    PS5000A_THRESHOLD_DIRECTION aux
)
```

**THIS FUNCTION IS NOT RECOMMENDED FOR NEW APPLICATIONS.**
In new applications please use `ps5000aSetTriggerChannelDirectionsV2` instead.

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

**Applicability**
All modes

**Arguments**
- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `channelA, channelB, channelC, channelD, ext`, the direction in which the signal must pass through the threshold to activate the trigger. See `PS5000A_THRESHOLD_DIRECTION` for allowable values. If using an edge or level trigger in conjunction with a pulse-width trigger, see `ps5000aSetPulseWidthQualifierDirections` for more information.
- `aux`, not used.

**Returns**
- `PICO_OK` or other code from `PicoStatus.h`
4.69.1 PS5000A_THRESHOLD_DIRECTION enumerated type

typedef enum enPS5000AThresholdDirection
{
    PS5000A_ABOVE,
    PS5000A_BELOW,
    PS5000A_RISING,
    PS5000A_FALLING,
    PS5000A_RISING_OR_FALLING,
    PS5000A_ABOVE_LOWER,
    PS5000A_BELOW_LOWER,
    PS5000A_RISING_LOWER,
    PS5000A_FALLING_LOWER,
    PS5000A_INSIDE = PS5000A_ABOVE,
    PS5000A_OUTSIDE = PS5000A_BELOW,
    PS5000A_ENTER = PS5000A_RISING,
    PS5000A_EXIT = PS5000A_FALLING,
    PS5000A_ENTER_OR_EXIT = PS5000A_RISING_OR_FALLING,
    PS5000A_POSITIVE_RUNT = 9,
    PS5000A_NEGATIVE_RUNT,
    PS5000A_NONE = PS5000A_RISING
} PS5000A_THRESHOLD_DIRECTION;

These values specify the direction(s) in which the trigger source must cross the threshold(s) to cause a trigger event.

<table>
<thead>
<tr>
<th>Value</th>
<th>Trigger type</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS5000A_ABOVE</td>
<td>gated</td>
<td>above the upper threshold</td>
</tr>
<tr>
<td>PS5000A_ABOVE_LOWER</td>
<td>gated</td>
<td>above the lower threshold</td>
</tr>
<tr>
<td>PS5000A_BELOW</td>
<td>gated</td>
<td>below the upper threshold</td>
</tr>
<tr>
<td>PS5000A_BELOW_LOWER</td>
<td>gated</td>
<td>below the lower threshold</td>
</tr>
<tr>
<td>PS5000A_RISING</td>
<td>threshold</td>
<td>rising edge, using upper threshold</td>
</tr>
<tr>
<td>PS5000A_RISING_LOWER</td>
<td>threshold</td>
<td>rising edge, using lower threshold</td>
</tr>
<tr>
<td>PS5000A_FALLING</td>
<td>threshold</td>
<td>falling edge, using upper threshold</td>
</tr>
<tr>
<td>PS5000A_FALLING_LOWER</td>
<td>threshold</td>
<td>falling edge, using lower threshold</td>
</tr>
<tr>
<td>PS5000A_RISING_OR_FALLING</td>
<td>threshold</td>
<td>either edge</td>
</tr>
<tr>
<td>PS5000A_INSIDE</td>
<td>window-qualified</td>
<td>inside window</td>
</tr>
<tr>
<td>PS5000A_OUTSIDE</td>
<td>window-qualified</td>
<td>outside window</td>
</tr>
<tr>
<td>PS5000A_ENTER</td>
<td>window</td>
<td>entering the window</td>
</tr>
<tr>
<td>PS5000A_EXIT</td>
<td>window</td>
<td>leaving the window</td>
</tr>
<tr>
<td>PS5000A_ENTER_OR_EXIT</td>
<td>window</td>
<td>either entering or leaving the window</td>
</tr>
<tr>
<td>PS5000A_POSITIVE_RUNT</td>
<td>window-qualified</td>
<td></td>
</tr>
<tr>
<td>PS5000A_NEGATIVE_RUNT</td>
<td>window-qualified</td>
<td></td>
</tr>
<tr>
<td>PS5000A_NONE</td>
<td>none</td>
<td></td>
</tr>
</tbody>
</table>
4.70 ps5000aSetTriggerChannelDirectionsV2 – set up signal polarities for triggering

```
PICO_STATUS ps5000aSetTriggerChannelDirectionsV2
(
    int16_t handle,
    PS5000A_DIRECTION * directions,
    uint16_t nDirections
)
```

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

**Applicability**

All modes.

Analog channels only (use `ps5000aSetTriggerDigitalPortProperties` for digital channels).

**Arguments**

- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `directions`, an array of directions in which the signal must pass through the threshold to activate the trigger. If you want to combine this with a pulse-width trigger, see `ps5000aSetPulseWidthQualifierDirections` for more information.
- `nDirections`, the length of the `directions` array.

**Returns**

PICO_OK or other code from PicoStatus.h

### 4.70.1 PS5000A_DIRECTION structure

```
typedef struct tPS5000ADirection
{
    PS5000A_CHANNEL source;
    PS5000A_THRESHOLD_DIRECTION direction;
    PS5000A_THRESHOLD_MODE mode;
} PS5000A_DIRECTION;
```

A structure of this type is passed to `ps5000aSetTriggerChannelDirectionsV2` in the `conditions` argument to specify the direction in which the specified source signal must cross the threshold(s) to produce a trigger event.

Each structure defines a condition to apply to one of the scope's input channels. The `ps5000aSetTriggerChannelDirectionsV2` function can OR together a number of these structures to produce the final trigger condition.

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

**Elements**

- `source`, the channel used as the trigger source
- `direction`, indicates the direction in which the signal must cross the threshold; see `PS5000A_THRESHOLD_DIRECTION`
mode, whether to use a level trigger (a single threshold) or a window trigger (two thresholds defining a range). See `PS5000A_THRESHOLD_MODE`.

### 4.70.2 PS5000A_THRESHOLD_MODE enumerated type

```c
typedef enum enPS5000AThresholdMode
{
    PS5000A_LEVEL,
    PS5000A_WINDOW
} PS5000A_THRESHOLD_MODE;
```

These values specify the type of threshold (level or window) used by a trigger condition.

<table>
<thead>
<tr>
<th>Applicability</th>
<th>Setting trigger conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Values</td>
<td></td>
</tr>
<tr>
<td>PS5000A_LEVEL</td>
<td>use a level trigger (an edge or level trigger with a single threshold)</td>
</tr>
<tr>
<td>PS5000A_WINDOW</td>
<td>use a window trigger (two thresholds defining a range)</td>
</tr>
</tbody>
</table>
4.71 ps5000aSetTriggerChannelProperties – set up trigger thresholds

```
PICO_STATUS ps5000aSetTriggerChannelProperties
(
    int16_t handle,
    PS5000A_TRIGGER_CHANNEL_PROPERTIES * channelProperties,
    int16_t nChannelProperties,
    int16_t auxOutputEnable,
    int32_t autoTriggerMilliseconds
)
```

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

**Applicability**

All modes

**Arguments**

- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `* channelProperties`, a pointer to an array of `PS5000A_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 length of the `channelProperties` array. If zero, triggering is switched off.
- `auxOutputEnable`, not used.
- `autoTriggerMilliseconds`, the time in milliseconds for which the scope device will wait before collecting data if no trigger event occurs. If this is set to zero, the scope device will wait indefinitely for a trigger.

**Returns**

`PICO_OK` or other code from `PicoStatus.h`
4.71.1 PS5000A_TRIGGER_CHANNEL_PROPERTIES structure

typedef struct tPS5000ATriggerChannelProperties
{
    int16_t thresholdUpper;
    uint16_t thresholdUpperHysteresis;
    int16_t thresholdLower;
    uint16_t thresholdLowerHysteresis;
    PS5000A_CHANNEL channel;
    PS5000A_THRESHOLD_MODE thresholdMode;
} PS5000A_TRIGGER_CHANNEL_PROPERTIES

A structure of this type is passed to ps5000aSetTriggerChannelProperties in the channelProperties argument to specify the trigger thresholds and threshold mode (level or window) for the specified channel.

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

**Elements**

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

thresholdUpperHysteresis, the hysteresis by which the trigger must exceed the upper threshold before it will fire. It is scaled in 16-bit counts.

thresholdLower, the lower threshold at which the trigger must fire. This is scaled in 16-bit ADC counts at the currently selected range for that channel.

thresholdLowerHysteresis, the hysteresis by which the trigger must exceed the lower threshold before it will fire. It is scaled in 16-bit counts.

channel, the channel to which the properties apply.

thresholdMode, either a level or window trigger. See PS5000A_THRESHOLD_MODE.
4.72 ps5000aSetTriggerChannelPropertiesV2 – set up trigger thresholds

```
PICO_STATUS ps5000aSetTriggerChannelPropertiesV2
(  
    int16_t handle,
    PS5000A_TRIGGER_CHANNEL_PROPERTIES_V2 * channelProperties,
    int16_t nChannelProperties,
    int16_t auxOutputEnable
)
```

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

**Applicability**
All modes

**Arguments**

- `handle`, the device identifier returned by `ps5000aOpenUnit`
- `* channelProperties`, a pointer to an array of `PS5000A_TRIGGER_CHANNEL_PROPERTIES_V2` 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 length of the `channelProperties` array. If zero, triggering is switched off.
- `auxOutputEnable`, not used.

**Returns**
PICO_OK or other code from PicoStatus.h

### 4.72.1 PS5000A_TRIGGER_CHANNEL_PROPERTIES_V2 structure

```
typedef struct tPS5000ATriggerChannelPropertiesV2
{
    int16_t thresholdUpper;
    uint16_t thresholdUpperHysteresis;
    int16_t thresholdLower;
    uint16_t thresholdLowerHysteresis;
    PS5000A_CHANNEL channel;
} PS5000A_TRIGGER_CHANNEL_PROPERTIES_V2
```

A structure of this type is passed to `ps5000aSetTriggerChannelPropertiesV2` in the `channelProperties` argument to specify the trigger thresholds for a given channel.

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

**Elements**

- `thresholdUpper`, the upper threshold at which the trigger must fire. This is scaled in 16-bit ADC counts at the currently selected range for that channel.
thresholdUpperHysteresis, the hysteresis by which the trigger must exceed the upper threshold before it will fire. It is scaled in 16-bit counts.

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

cchannel, the channel to which the properties apply.

Upper and lower thresholds
The digital triggering hardware in your PicoScope has two independent trigger thresholds called upper and lower. For some trigger types you can freely choose which threshold to use. See PS5000A_THRESHOLD_DIRECTION for a list of trigger types and the thresholds that they support. Dual thresholds are used for pulse-width triggering, when one threshold applies to the level trigger and the other to the pulse-width qualifier; and for window triggering, when the two thresholds define the upper and lower limits of the window.

Each threshold has its own trigger and hysteresis settings.

4.72.2 Hysteresis

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

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

The trigger fires at A as the signal rises past the trigger threshold. It does not fire at B because the signal has not yet dipped below the hysteresis threshold. The trigger fires again at C after the signal has dipped below the hysteresis threshold and risen again past the trigger threshold.
4.73 ps5000aSetTriggerDelay – set up post-trigger delay

```c
PICO_STATUS ps5000aSetTriggerDelay
(
    int16_t handle,
    uint32_t delay
)
```

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

**Applicability**

All modes (but delay is ignored in streaming mode).

**Arguments**

- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `delay`, the time between the trigger occurring and the first sample. For example, if `delay = 100`, the scope waits 100 sample periods before sampling. At a `timebase` of 500 MS/s, or 2 ns per sample, the total delay is then:

  \[
  100 \times 2 \text{ ns} = 200 \text{ ns}
  \]

  Range: 0 to `MAX_DELAY_COUNT`

**Returns**

PICO_OK or other code from `PicoStatus.h`
4.74 ps5000aSetTriggerDigitalPortProperties – set up digital inputs for triggering

```c
PICO_STATUS ps5000aSetTriggerDigitalPortProperties
(
    int16_t handle,
    PS5000A_DIGITAL_CHANNEL_DIRECTIONS * directions
    int16_t nDirections
)
```

This function sets the individual digital channels' trigger directions. Each trigger direction consists of a channel name and a direction. If the channel is not included in the array of `PS5000A_DIGITAL_CHANNEL_DIRECTIONS`, the driver assumes that the digital channel's trigger direction is `PS5000A_DIGITAL_DONT_CARE`.

Use with the other functions listed under Triggering. In particular, you must call `ps5000aSetTriggerChannelConditionsV2` if you want to include the digital ports in the trigger conditions.

**Applicability**
PicoScope 5000D MSO models only.

**Arguments**
- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `* directions`, a pointer to an array of `PS5000A_DIGITAL_CHANNEL_DIRECTIONS` structures describing the requested properties. The array can contain a single element describing the properties of one channel, or a number of elements describing several digital channels. If `directions` is NULL, digital triggering is switched off. A digital channel that is not included in the array is set to `PS5000A_DIGITAL_DONT_CARE`. The outcomes of all the `DIRECTIONS` structures in the array are ORed together to produce the final trigger signal.
- `nDirections`, the number of digital channel directions being passed to the driver.

**Returns**
PICO_OK or other code from PicoStatus.h
4.74.1 PS5000A_DIGITAL_CHANNEL_DIRECTIONS structure

typedef struct tPS5000ADigitalChannelDirections
{
    PS5000A_DIGITAL_CHANNEL    channel;
    PS5000A_DIGITAL_DIRECTION  direction;
} PS5000A_DIGITAL_CHANNEL_DIRECTIONS;

A structure of this type is passed to `ps5000aSetTriggerDigitalPortProperties` in the `directions` argument to specify the trigger direction for the specified digital channel.

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

Elements

- `channel`, the digital channel to be set up.
- `direction`, the direction in which the digital input must cross the threshold(s) to cause a trigger event.

4.74.2 PS5000A_DIGITAL_CHANNEL enumerated type

typedef enum enPS5000ADigitalChannel
{
    PS5000A_DIGITAL_CHANNEL_0,
    PS5000A_DIGITAL_CHANNEL_1,
    PS5000A_DIGITAL_CHANNEL_2,
    PS5000A_DIGITAL_CHANNEL_3,
    PS5000A_DIGITAL_CHANNEL_4,
    PS5000A_DIGITAL_CHANNEL_5,
    PS5000A_DIGITAL_CHANNEL_6,
    PS5000A_DIGITAL_CHANNEL_7,
    PS5000A_DIGITAL_CHANNEL_8,
    PS5000A_DIGITAL_CHANNEL_9,
    PS5000A_DIGITAL_CHANNEL_10,
    PS5000A_DIGITAL_CHANNEL_11,
    PS5000A_DIGITAL_CHANNEL_12,
    PS5000A_DIGITAL_CHANNEL_13,
    PS5000A_DIGITAL_CHANNEL_14,
    PS5000A_DIGITAL_CHANNEL_15,
    PS5000A_DIGITAL_CHANNEL_16,
    PS5000A_DIGITAL_CHANNEL_17,
    PS5000A_DIGITAL_CHANNEL_18,
    PS5000A_DIGITAL_CHANNEL_19,
    PS5000A_DIGITAL_CHANNEL_20,
    PS5000A_DIGITAL_CHANNEL_21,
    PS5000A_DIGITAL_CHANNEL_22,
    PS5000A_DIGITAL_CHANNEL_23,
    PS5000A_DIGITAL_CHANNEL_24,
    PS5000A_DIGITAL_CHANNEL_25,
    PS5000A_DIGITAL_CHANNEL_26,
    PS5000A_DIGITAL_CHANNEL_27,
    PS5000A_DIGITAL_CHANNEL_28,
    PS5000A_DIGITAL_CHANNEL_29,
These values specify one of the digital channels of a mixed-signal PicoScope 5000 Series model.

**Applicability**
Setting trigger conditions

**Values**

<table>
<thead>
<tr>
<th>PS5000A_DIGITAL_CHANNEL_0</th>
<th>least significant bit of PORT0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS5000A_DIGITAL_CHANNEL_7</td>
<td>most significant bit of PORT0</td>
</tr>
<tr>
<td>PS5000A_DIGITAL_CHANNEL_8</td>
<td>most significant bit of PORT1</td>
</tr>
<tr>
<td>PS5000A_DIGITAL_CHANNEL_15</td>
<td>most significant bit of PORT1</td>
</tr>
<tr>
<td>PS5000A_DIGITAL_CHANNEL_16</td>
<td>not used</td>
</tr>
<tr>
<td>PS5000A_DIGITAL_CHANNEL_31</td>
<td>not used</td>
</tr>
</tbody>
</table>

4.74.3 PS5000A_DIGITAL_DIRECTION enumerated type

```c
typedef enum enPS5000ADigitalDirection {
    PS5000A_DIGITAL_DONT_CARE,
    PS5000A_DIGITAL_DIRECTION_LOW,
    PS5000A_DIGITAL_DIRECTION_HIGH,
    PS5000A_DIGITAL_DIRECTION_RISING,
    PS5000A_DIGITAL_DIRECTION_FALLING,
    PS5000A_DIGITAL_DIRECTION_RISING_OR_FALLING,
    PS5000A_DIGITAL_MAX_DIRECTION
} PS5000A_DIGITAL_DIRECTION;
```

These values specify the polarity of a digital channel used as a trigger source.

**Applicability**
Setting trigger conditions

**Values**

| PS5000A_DIGITAL_DONT_CARE                  | ignore input                      |
| PS5000A_DIGITAL_DIRECTION_LOW             | input must be low                 |
| PS5000A_DIGITAL_DIRECTION_HIGH            | input must be high                |
| PS5000A_DIGITAL_DIRECTION_RISING          | input must have a rising edge     |
| PS5000A_DIGITAL_DIRECTION_FALLING         | input must have a falling edge    |
| PS5000A_DIGITAL_DIRECTION_RISING_OR_FALLING | input must have an edge of either polarity |
4.75  ps5000aSigGenArbitraryMinMaxValues – get AWG parameters

```c
PICO_STATUS ps5000aSigGenArbitraryMinMaxValues(
    int16_t handle,
    int16_t * minArbitraryWaveformValue,
    int16_t * maxArbitraryWaveformValue,
    uint32_t * minArbitraryWaveformSize,
    uint32_t * maxArbitraryWaveformSize
)
```

This function returns the range of possible sample values and waveform buffer sizes that can be supplied to `ps5000aSetSigGenArbitrary` for setting up the arbitrary waveform generator (AWG). These values vary between different models in the PicoScope 5000 Series.

**Applicability**

All models with AWG

**Arguments**

- **handle**, the device identifier returned by `ps5000aOpenUnit`.
- **minArbitraryWaveformValue**, on exit, the lowest sample value allowed in the `arbitraryWaveform` buffer supplied to `ps5000aSetSigGenArbitrary`.
- **maxArbitraryWaveformValue**, on exit, the highest sample value allowed in the `arbitraryWaveform` buffer supplied to `ps5000aSetSigGenArbitrary`.
- **minArbitraryWaveformSize**, on exit, the minimum value allowed for the `arbitraryWaveformSize` argument supplied to `ps5000aSetSigGenArbitrary`.
- **maxArbitraryWaveformSize**, on exit, the maximum value allowed for the `arbitraryWaveformSize` argument supplied to `ps5000aSetSigGenArbitrary`.

**Returns**

PICO_OK or other code from PicoStatus.h
4.76 ps5000aSigGenFrequencyToPhase – convert frequency to phase count

```c
PICO_STATUS ps5000aSigGenFrequencyToPhase(
    int16_t handle,
    double frequency,
    PS5000A_INDEX_MODE indexMode,
    uint32_t bufferLength,
    uint32_t * phase
)
```

This function converts a frequency to a phase count for use with the arbitrary waveform generator (AWG). The value returned depends on the length of the buffer, the index mode passed and the device model. The phase count can then be used as one of the deltaPhase arguments for `ps5000aSetSigGenArbitrary` or `ps5000aSetSigGenPropertiesArbitrary`.

**Applicability**
All models with AWG

**Arguments**
- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `frequency`, the required AWG output frequency.
- `indexMode`, see `PS5000A_INDEX_MODE`.
- `bufferLength`, the number of samples in the AWG buffer.
- `phase`, on exit, the `deltaPhase` argument to be sent to the AWG setup function

**Returns**
PICO_OK or other code from PicoStatus.h
4.77 ps5000aSigGenSoftwareControl – trigger the signal generator

```c
PICO_STATUS ps5000aSigGenSoftwareControl
(int16_t handle,
 int16_t state)
```

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

1. Call `ps5000aSetSigGenBuiltIn` or `ps5000aSetSigGenArbitrary` to set up the signal generator, setting the `triggerSource` argument to `PS5000A_SIGGEN_SOFT_TRIG`.

2. (a) If you set the signal generator `triggerType` to edge triggering (`PS5000A_SIGGEN_RISING` or `PS5000A_SIGGEN_FALLING`), call `ps5000aSigGenSoftwareControl` once to trigger the signal generator.

(b) If you set the signal generator `triggerType` to gated-low triggering (`PS5000A_SIGGEN_GATE_LOW`), call `ps5000aSigGenSoftwareControl` with `state = 0` to start the sweep and then again with `state = 1` to stop it.

(c) If you set the signal generator `triggerType` to gated-high triggering (`PS5000A_SIGGEN_GATE_HIGH`), call `ps5000aSigGenSoftwareControl` with `state = 1` to start the sweep and then again with `state = 0` to stop it.

The gating can also be used to stop and start output if the number of shots has been set for a continuous run (`shots` is set to `PS5000A_SHOT_SWEEP_TRIGGER_CONTINUOUS_RUN`):

(d) If `triggerType = PS5000A_SIGGEN_GATE_HIGH`, `state = 1` will cause the signal generator to output, `state = 0` will cause it to stop.

(e) If `triggerType = PS5000A_SIGGEN_GATE_LOW`, the signal generator starts to output immediately. Setting `state = 1` will cause it to stop.

**Applicability**

Use with `ps5000aSetSigGenBuiltIn` or `ps5000aSetSigGenArbitrary`.

**Arguments**

- `handle`, the device identifier returned by `ps5000aOpenUnit`.
- `state`, specifies the new state of the gate signal. A change of `state` either starts or stops the sweep depending on the `triggerType`, as detailed above. Effective only when the signal generator `triggerType` is set to `PS5000A_SIGGEN_GATE_HIGH` or `PS5000A_SIGGEN_GATE_LOW`. Ignored for other trigger types. Effective only when the signal generator `triggerType` is set to `PS5000A_SIGGEN_GATE_HIGH` or `PS5000A_SIGGEN_GATE_LOW`. Ignored for other trigger types.
  - `0`: set the gate signal low
  - `1`: set the gate signal high

**Returns**

- `PICO_OK` or other code from `PicoStatus.h`
### 4.78  ps5000aStop – stop data capture

```
PICO_STATUS ps5000aStop
(
    int16_t        handle
)
```

This function stops the scope device from sampling data.

When running the device in streaming mode, you should 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, ETS mode or rapid block mode, you can call this function to interrupt data capture.

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

<table>
<thead>
<tr>
<th>Applicability</th>
</tr>
</thead>
<tbody>
<tr>
<td>All modes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle, the device identifier returned by ps5000aOpenUnit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>PICO_OK or other code from PicoStatus.h</td>
</tr>
</tbody>
</table>
4.79  ps5000aTriggerWithinPreTriggerSamples – change triggering behavior

```c
PICO_STATUS ps5000aTriggerWithinPreTriggerSamples
(int16_t handle,
PS5000A_TRIGGER_WITHIN_PRE_TRIGGER state)
```

This function selects a mode in which the scope can be triggered anywhere within the pre-trigger samples, as opposed to the normal operation of only arming the trigger after all the pre-trigger samples have been collected. To find out where in the captured data the trigger event occurred when using this mode, call `ps5000aGetTriggerInfoBulk`.

Triggering within the pre-trigger samples is only available in block mode. Triggering must be enabled using `ps5000aSetSimpleTrigger` or the suite of `ps5000aSetTriggerChannel… advanced trigger functions`. This mode is not compatible with trigger delay (set using `ps5000aSetTriggerDelay`) or ETS (set using `ps5000aSetEts`). `ps5000aRunBlock` returns an error code if this mode is selected at the same time as trigger delay or ETS.

This mode is not compatible with streaming. Calling `ps5000aRunStreaming` returns an error code if triggering within pre-trigger samples is armed.

<table>
<thead>
<tr>
<th>Applicability</th>
<th>All modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td>handle, the device identifier returned by <code>ps5000aOpenUnit</code>. state, the desired state of the trigger. See <code>PS5000A_TRIGGER_WITHIN_PRE_TRIGGER</code>.</td>
</tr>
<tr>
<td>Returns</td>
<td>PICO_OK or other code from <code>PicoStatus.h</code></td>
</tr>
</tbody>
</table>

### 4.79.1  PS5000A_TRIGGER_WITHIN_PRE_TRIGGER enumerated type

```c
typedef enum enPS5000ATriggerWithinPreTrigger
{
    PS5000A_DISABLE,
    PS5000A_ARM
} PS5000A_TRIGGER_WITHIN_PRE_TRIGGER;
```

These values enable or disable the trigger during the pre-trigger period.

<table>
<thead>
<tr>
<th>Applicability</th>
<th>Setting trigger conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Values</td>
<td>PS5000A_DISABLE – uses triggering in the normal way (default)</td>
</tr>
</tbody>
</table>
4.80 Callback functions

A callback function is a function, within your own application, that the ps5000a driver calls to signal that data is ready.

For programming languages and environments that do not support callbacks, we provide a set of wrapper functions.
4.80.1 ps5000aBlockReady – indicate when block-mode data ready

typedef void (CALLBACK *ps5000aBlockReady) (
    int16_t handle,
    PICO_STATUS status,
    void * pParameter
)

This callback function is part of your application. You register it with the ps5000a driver using ps5000aRunBlock, and the driver calls it back when block-mode data is ready. You can then download the data using the ps5000aGetValues function.

**Applicability**

Block mode only

**Arguments**

- **handle**, the device identifier returned by ps5000aOpenUnit.
- **status**, indicates whether an error occurred during collection of the data.
- **pParameter**, a void pointer passed from ps5000aRunBlock. Your callback function can write to this location to send any data, such as a status flag, back to your application.

**Returns**

nothing
4.80.2  ps5000aDataReady – indicate when post-collection data ready

typedef void (CALLBACK *ps5000aDataReady)(
    int16_t handle,
    PICO_STATUS status,
    uint32_t noOfSamples,
    int16_t overflow,
    void * pParameter
)

This is a **callback** function that you write to collect data from the driver. You supply a pointer to the function when you call **ps5000aGetValuesAsync**, and the driver calls your function back when the data is ready.

### Applicability
All modes

### Arguments
handle,  the device identifier returned by **ps5000aOpenUnit**.

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 **ps5000aGetValuesAsync**. 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
4.80.3  ps5000aStreamingReady – indicate when streaming-mode data ready

typedef void (CALLBACK *ps5000aStreamingReady)(
    int16_t handle,
    int32_t noOfSamples,
    uint32_t startIndex,
    int16_t overflow,
    uint32_t triggerAt,
    int16_t triggered,
    int16_t autoStop,
    void * pParameter
)

This callback function is part of your application. You register it with the driver using ps5000aGetStreamingLatestValues, and the driver calls it back when streaming-mode data is ready. You can then download the data using the ps5000aGetValuesAsync function.

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

<table>
<thead>
<tr>
<th>Applicability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Streaming mode only</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle, the device identifier returned by ps5000aOpenUnit.</td>
</tr>
<tr>
<td>noOfSamples, the number of samples to collect.</td>
</tr>
<tr>
<td>startIndex, an index to the first valid sample in the buffer. This is the buffer that was previously passed to ps5000asetDataBuffer.</td>
</tr>
<tr>
<td>overflow, returns a set of flags that indicate whether an overvoltage has occurred on any of the channels. It is a bit pattern with bit 0 denoting Channel A.</td>
</tr>
<tr>
<td>triggerAt, an index to the buffer indicating the location of the trigger point relative to startIndex. This parameter is valid only when triggered is non-zero.</td>
</tr>
<tr>
<td>triggered, a flag indicating whether a trigger occurred. If non-zero, a trigger occurred at the location indicated by triggerAt.</td>
</tr>
<tr>
<td>autoStop, the flag that was set in the call to ps5000aRunStreaming.</td>
</tr>
<tr>
<td>* pParameter, a void pointer passed from ps5000aGetStreamingLatestValues. The callback function can write to this location to send any data, such as a status flag, back to the application.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>nothing</td>
</tr>
</tbody>
</table>
4.81 Wrapper functions

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

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

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

The ps5000aWrap.dll wrapper DLL has a callback function for streaming data collection that copies data from the driver buffer specified to a temporary application buffer of the same size. To do this, the driver and application buffers must be registered with the wrapper and the corresponding channel(s) must be specified as being enabled. You should process the data in the temporary application buffer accordingly, for example by copying the data into a large array.

Procedure:
1. Open the oscilloscope using ps5000aOpenUnit.

1a. Call setChannelCount so that the wrapper determines the number of analog channels on the device and if it has digital ports.

2. Select channels, ranges and AC/DC coupling using ps5000aSetChannel.

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

2b. Optional: Call ps5000aSetDigitalPort to configure a digital port (mixed-signal scopes only).

2c. Inform the wrapper which digital ports have been enabled by calling setEnabledDigitalPorts.

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

4. Call ps5000aSetDataBuffer (or for aggregated data collection ps5000aSetDataBuffers) to tell the driver where your data buffer(s) is(are).

4a. Register the data buffer(s) with the wrapper and set the application buffer(s) into which the data will be copied. Call setAppAndDriverBuffers (or setMaxMinAppAndDriverBuffers for aggregated data collection).

5. Start the oscilloscope running using ps5000aRunStreaming.

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

6a. Call the wrapper’s AvailableData function to obtain information on the number of samples collected and the start index in the buffer.

6b. Call the wrapper’s IsTriggerReady function for information on whether a trigger has occurred and the trigger index relative to the start index in the buffer.

7. Process data returned to your application data buffers.
8. Call AutoStopped if the autoStop parameter has been set to 1 in the call to `ps5000aRunStreaming`.

9. Repeat steps 6 to 8 until AutoStopped returns true or you wish to stop data collection.

10. Call `ps5000aStop`, even if the autoStop parameter was set to TRUE.

11. To disconnect a device, call `ps5000aCloseUnit`. 
5  Reference

5.1  Driver status codes

Every function in the ps5000a library returns a driver status code from the list of `PICO_STATUS` values in the file `PicoStatus.h`. This file is included in PicoSDK.

5.2  Enumerated types and constants

The enumerated types and constants used by the ps5000a library are defined in the `ps5000aApi.h` header file. We recommend that you refer to these objects by name unless your programming language allows only numerical values.

5.3  Numeric data types

Here are the sizes and ranges of the numeric data types used in the ps5000a API:

<table>
<thead>
<tr>
<th>Type</th>
<th>Bits</th>
<th>Signed or unsigned?</th>
</tr>
</thead>
<tbody>
<tr>
<td>int16_t</td>
<td>16</td>
<td>signed</td>
</tr>
<tr>
<td>uint16_t</td>
<td>16</td>
<td>unsigned</td>
</tr>
<tr>
<td>enum</td>
<td>32</td>
<td>enumerated</td>
</tr>
<tr>
<td>int32_t</td>
<td>32</td>
<td>signed</td>
</tr>
<tr>
<td>uint32_t</td>
<td>32</td>
<td>unsigned</td>
</tr>
<tr>
<td>float</td>
<td>32</td>
<td>signed (IEEE 754)</td>
</tr>
<tr>
<td>int64_t</td>
<td>64</td>
<td>signed</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>signed (IEEE 754)</td>
</tr>
</tbody>
</table>
5.4 Glossary

**Aggregation.** The ps5000a API can use a method called aggregation to reduce the amount of data your application needs to process. This means that for every block of consecutive samples, it stores only the minimum and maximum values. You can set the number of samples in each block, called the aggregation parameter, when you call `ps5000aRunStreaming` for real-time capture, and when you call `ps5000aGetStreamingLatestValues` to obtain post-processed data.

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

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

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

**Callback.** A mechanism that the API 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.

**Coupling mode.** This mode selects either AC or DC coupling in the oscilloscope's input path. Use AC mode for small signals that may be superimposed on a DC level. Use DC mode for measuring absolute voltage levels. Set the coupling mode using `ps5000aSetChannel`.

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

**External trigger.** This is the BNC socket marked EXT or Ext. It can be used to start a data collection run but cannot be used to record data.

**Flexible power.** The 5000 Series oscilloscopes can be powered by either the two-headed USB cable supplied for obtaining power from two USB ports, or a single USB port and the AC adaptor (included with 4-channel models only).

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

**MSO. Mixed-signal oscilloscope.** PicoScope 5000D MSO models have analog and digital input channels so can be used as both an oscilloscope and a logic analyzer at the same time.

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

**Signal generator.** The signal generator output is the BNC socket marked GEN or Gen on the oscilloscope. If you connect a BNC cable between this and one of the channel inputs, you can send a signal into one of the channels. It can generate a sine, square or triangle wave that can be swept back and forth.
Streaming mode. A sampling mode in which the oscilloscope samples data and returns it to the computer in an unbroken stream. This mode of operation is effective when the input signal being sampled contains only low frequencies.

USB 1.1. An early version of the Universal Serial Bus standard found on older PCs. Although your PicoScope 5000 Series device will work with a USB 1.1 port, it will operate much more slowly than with a USB 2.0 or 3.0 port.

USB 2.0. A typical USB 2.0 port supports a data transfer rate that is 40 times faster than USB 1.1. USB 2.0 is backwards-compatible with USB 1.1.

USB 3.0. A typical USB 3.0 port supports a data transfer rate that is 10 times faster than USB 2.0. USB 3.0 is backwards-compatible with USB 2.0 and USB 1.1.

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

Voltage range. The voltage range is a pair of input voltages that correspond to the maximum and minimum sample values that the driver can return in the selected mode. For example, in the range identified by the enumeration value `PS5000A_2V`, the oscilloscope returns the maximum sample value when the input voltage is +2 V and the minimum sample value when the input voltage is –2 V.
Index

A
AC/DC coupling 34, 78, 138
Access 2
ADC count 64, 66
Aggregation 20, 138
Analog bandwidth 138
Analog offset 33, 78
API function calls 26
Arbitrary waveform generator 98
    buffer length 126
    index modes 99
    sample values 126

B
Bandwidth limiter 77, 78
Block mode 7, 8, 9, 138
    asynchronous call 11
    callback 132
    polling status 62
    running 74
    using 10
Buffer size 138

C
Callback 8, 138
    block mode 132
    definition 133
    ETS mode 18
    streaming mode 134
Callback functions 131
Channel flags 28
Channel information 35
Channels 27, 79
    enabling 78
    settings 78
Closing units 29
Communication 72
Constants 137
Copyright 2
Coupling type 138
    setting 78

D
Data acquisition 20
Data buffers

E
Enumerated types 137
Enumerating oscilloscopes 31
ETS 86, 138
    overview 18
    setting time buffers 87, 88
    setting up 85
    using 19
External trigger 138

F
Fitness for purpose 2
Functions
    overview 26
    ps5000aBlockReady 132
    ps5000aChangePowerSource 26
    ps5000aChannelCombinationsStateless 27
    ps5000aCloseUnit 29
    ps5000aCurrentPowerSource 30
    ps5000aDataReady 133
    ps5000aEnumerateUnits 31
    ps5000aFlashLed 32
    ps5000aGetAnalogueOffset 33
    ps5000aGetChannelInformation 35
    ps5000aGetDeviceResolution 36
    ps5000aGetMaxDownSampleRatio 37
    ps5000aGetMaxSegments 38
    ps5000aGetMinimumTimebaseStateless 39
    ps5000aGetNoOfCaptures 40, 41
    ps5000aGetStreamingLatestValues 42
    ps5000aGetTimebase 22, 43
    ps5000aGetTimebase2 44
Functions
ps5000aGetTriggerInfoBulk 45
ps5000aGetTriggerTimeOffset 49
ps5000aGetTriggerTimeOffset64 50
ps5000aGetUnitInfo 51
ps5000aGetValues 11, 52
ps5000aGetValuesAsync 11, 54
ps5000aGetValuesBulk 55
ps5000aGetValuesOverlapped 56
ps5000aGetValuesOverlappedBulk 58
ps5000aGetValuesTriggerTimeOffsetBulk 59
ps5000aGetValuesTriggerTimeOffsetBulk64 60
ps5000aIsLedFlashing 61
ps5000aIsReady 62
ps5000aIsTriggerOrPulseWidthQualifierEnabled 63
ps5000aMaximumValue 5, 64
ps5000aMemorySegments 65
ps5000aMinimumValue 5, 66
ps5000aNearestSampleIntervalStateless 67
ps5000aNoOfStreamingValues 68
ps5000aOpenUnit 69
ps5000aOpenUnitAsync 70
ps5000aOpenUnitProgress 71
ps5000aPingUnit 72
ps5000aQueryOutputEdgeDetect 73
ps5000aRunBlock 74
ps5000aRunStreaming 75
ps5000aSetAutoTriggerMicroSeconds 76
ps5000aSetChannel 5, 78
ps5000aSetDataBuffers 80
ps5000aSetDataBuffers 81
ps5000aSetDeviceResolution 82
ps5000aSetDigitalPort 84
ps5000aSetEts 18, 85
ps5000aSetEtsTimeBuffer 87
ps5000aSetEtsTimeBuffers 88
ps5000aSetNoOfCaptures 89
ps5000aSetOutputEdgeDetect 90
ps5000aSetPulseWidthDigitalPortProperties 91
ps5000aSetPulseWidthQualifier 92
ps5000aSetPulseWidthQualifierConditions 94
ps5000aSetPulseWidthQualifierDirections 96
ps5000aSetPulseWidthQualifierProperties 97
ps5000aSetSigGenArbitrary 98
ps5000aSetSigGenBuiltIn 102
ps5000aSetSigGenPropertiesArbitrary 107
ps5000aSetSigGenPropertiesBuiltIn 108
ps5000aSetSimpleTrigger 7, 109
ps5000aSetTriggerChannelConditions 7, 110
ps5000aSetTriggerChannelConditionsV2 112
ps5000aSetTriggerChannelDirections 7, 114
ps5000aSetTriggerChannelDirectionsV2 116
ps5000aSetTriggerChannelProperties 7, 118
ps5000aSetTriggerChannelPropertiesV2 120
ps5000aSetTriggerDelay 122
ps5000aSetTriggerDigitalPortProperties 123
ps5000aSigGenArbitraryMinMaxValues 126
ps5000aSigGenFrequencyToPhase 127
ps5000aSigGenSoftwareControl 128
ps5000aStop 11, 129
ps5000aStreamingReady 134
ps5000aTriggerWithinPreTriggerSamples 130

H
Hysteresis 119

I
Input range, selecting 78
Intended use 1

L
LED 61
flashing 32
Legal information 2
Liability 2

M
Memory in scope 9
Mission-critical applications 2
MSO 138
Multi-unit operation 25

O
One-shot signals 18
Opening a unit 69
checking progress 71
without blocking 70
Output edge detection 73, 90
Overvoltage 138

P
PC Oscilloscope 1
PC requirements 3
PICO_STATUS enum type 137
PicoScope 5000 Series 1
PicoScope software 1, 4, 137
Power source 26, 30, 138
flexible power options 24
ps5000a.dll 4
PS5000A_BANDWIDTH_LIMITER enumerated type 77
S
Sampling interval 39, 67
Sampling rate maximum 9, 138
Scaling 5
Segmented memory 9, 10, 20, 65
Serial numbers 31
Setup time 9
Signal generator 138
arbitrary waveforms 98
built-in waveforms 102
calculating phase 127
frequency sweep type 101
software trigger 128
trigger source 104
type 103
wave type 104
Spectrum analyzer 1
Status codes 137
Stopping sampling 129
Streaming block mode deferred 56
rapid block mode 55
rapid block mode deferred 58
stored 21
streaming mode 42
Retrieving times
rapid block mode 59, 60
R
Ranges 34, 35
Rapid block mode 8, 12, 40, 41
aggregation 16
no aggregation 14
setting number of captures 89
using 12
Resolution 69, 82, 139
Retrieving data 52, 54
Index

Trigger 7
  digital port pulse width 91
digital ports 123
directions 114, 116
external 5
hysteresis 121
pulse-width qualifier 92
pulse-width qualifier conditions 93
pulse-width qualifier directions 96
pulse-width qualifier properties 94, 97
requesting status 63
setting up 109
stability 18
states 113
threshold mode 117
thresholds 120
time offset 49, 50
timeout 76
timestamps 45
within pre-trigger 130

U
  Unit information, reading 51
  Upgrades 2
  Usage 2
  USB 1, 3, 139
    hub 25

V
  Viruses 2
  Voltage range 5, 139
    selecting 78

W
  WinUsb.sys 4
<table>
<thead>
<tr>
<th>UK headquarters</th>
<th>North America office</th>
<th>Asia-Pacific office</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pico Technology</td>
<td>Pico Technology</td>
<td>Pico Technology</td>
</tr>
<tr>
<td>James House</td>
<td>320 N Glenwood Blvd</td>
<td>Room 2252, 22/F, Centro</td>
</tr>
<tr>
<td>Colmworth Business Park</td>
<td>Tyler</td>
<td>568 Hengfeng Road</td>
</tr>
<tr>
<td>St. Neots</td>
<td>Texas 75702</td>
<td>Zhabei District</td>
</tr>
<tr>
<td>Cambridgeshire</td>
<td>United States of America</td>
<td>Shanghai 200070</td>
</tr>
<tr>
<td>PE19 8YP</td>
<td></td>
<td>PR China</td>
</tr>
<tr>
<td>United Kingdom</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tel: +44 (0) 1480 396 395</td>
<td>Tel: +1 800 591 2796</td>
<td>Tel: +86 21 2226-5152</td>
</tr>
<tr>
<td>Fax: +44 (0) 1480 396 296</td>
<td>Fax: +1 620 272 0981</td>
<td></td>
</tr>
<tr>
<td><a href="mailto:sales@picotech.com">sales@picotech.com</a></td>
<td><a href="mailto:sales@picotech.com">sales@picotech.com</a></td>
<td><a href="mailto:pico.china@picotech.com">pico.china@picotech.com</a></td>
</tr>
<tr>
<td><a href="mailto:support@picotech.com">support@picotech.com</a></td>
<td><a href="mailto:support@picotech.com">support@picotech.com</a></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><a href="http://www.picotech.com">www.picotech.com</a></td>
</tr>
</tbody>
</table>