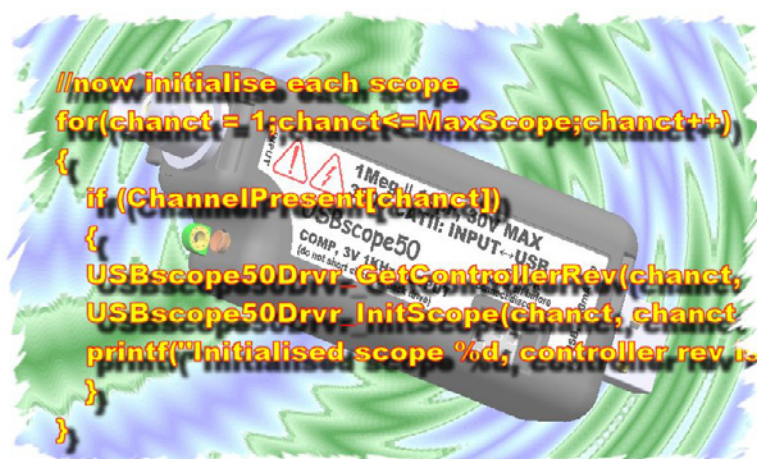




ELAN DIGITAL SYSTEMS LTD.

LITTLE PARK FARM ROAD,
SEGENSWORTH WEST,
FAREHAM,
HANTS. PO15 5SJ.
TEL: (44) (0)1489 579799
FAX: (44) (0)1489 577516
e-mail: support@pccard.co.uk
website: <http://www.pccard.co.uk>

USBscope50 SDK User's Guide (Software Development Kit) ES378



Important Notice: Please refer to ES370 "USBscope50 User's Guide" for safety notices for the USBscope50

All Trademarks are duly acknowledged.
The USBscope50 is Patent Pending.

REVISION HISTORY

ISSUE	PAGES	DATE	NOTES
1	55	31.10.2005	FIRST ISSUE
2	59	15.12.2005	RIS Modes Added

CONTENTS

1	OVERVIEW.....	5
2	THE USB INTERFACE.....	6
2.1	USB Device Drivers	6
2.1.1	Windows 2000, XP	6
2.1.2	Windows 98SE, ME.....	7
2.2	Digital Signing.....	7
3	THE DRIVER INTERFACE	8
3.1	Overview.....	8
3.2	Terminology	8
3.3	Driver Functions	9
3.3.1	USBscope50Drvrv_OpenDrvrv	10
3.3.2	USBscope50Drvrv_CloseDrvrv	11
3.3.3	USBscope50Drvrv_DrvrvIss	12
3.3.4	USBscope50Drvrv_Enumerate.....	13
3.3.5	USBscope50Drvrv_GetProductName	14
3.3.6	USBscope50Drvrv_GetSerialNumber	15
3.3.7	USBscope50Drvrv_GetHWRv	16
3.3.8	USBscope50Drvrv_GetPortNumber.....	17
3.3.9	USBscope50Drvrv_GetControllerRev	18
3.3.10	USBscope50Drvrv_OpenAndReset	19
3.3.11	USBscope50Drvrv_PortOpen	20
3.3.12	USBscope50Drvrv_Close.....	21
3.3.13	USBscope50Drvrv_InitScope	22
3.3.14	USBscope50Drvrv_SetDetectLine.....	23
3.3.15	USBscope50Drvrv_GetDetectLine	24
3.3.16	USBscope50Drvrv_SetBaseAdcClk	25
3.3.17	USBscope50Drvrv_GetBaseAdcClk.....	26
3.3.18	USBscope50Drvrv_SetDecimationRatio	27
3.3.19	USBscope50Drvrv_GetSamplePeriod.....	29
3.3.20	USBscope50Drvrv_GetSampleDepth	30
3.3.21	USBscope50Drvrv_SetUpFrontEnd.....	31
3.3.22	USBscope50Drvrv_SetOffset.....	32
3.3.23	USBscope50Drvrv_SetTrigMaster.....	33
3.3.24	USBscope50Drvrv_GetTrigMaster.....	34
3.3.25	USBscope50Drvrv_SetTrigType.....	35
3.3.26	USBscope50Drvrv_SetNormTrig	36
3.3.27	USBscope50Drvrv_SetTrigThreshold.....	37
3.3.28	USBscope50Drvrv_GetTriggeredStatus	38
3.3.29	USBscope50Drvrv_SetPreTrigDepth.....	39
3.3.30	USBscope50Drvrv_SetTriggerDelay	40
3.3.31	USBscope50Drvrv_AcquisitionStart	41

3.3.32	USBscope50Drvr_AcquisitionEnd.....	42
3.3.33	USBscope50Drvr_GetAcquisitionState	43
3.3.34	USBscope50Drvr_GetBufferBlocks	44
3.3.35	USBscope50Drvr_GetBufferBlocksMultiChan	45
3.3.36	USBscope50Drvr_GetBufferRIS	46
3.3.37	USBscope50Drvr_GetSamplesSinceTrigger.....	47
3.3.38	USBscope50Drvr_GetBufferIncremental	48
3.3.39	USBscope50Drvr_SetCalSource	49
3.3.40	USBscope50Drvr_SetLEDMode.....	50
3.3.41	USBscope50Drvr_FFT	51
3.3.42	USBscope50Drvr_ReScaleSampleData	52
4	RIS MODE	53
4.1	Overview.....	53
4.2	Principals of Operation	53
4.3	Organising the Data.....	54
5	GENERAL PROGRAMMING CONSIDERATIONS	56
5.1	Saved Scope Settings	56
5.2	Leaving Ports Open	56
6	EXAMPLE CODE & HEADER FILES.....	57
6.1	Visual Basic6	57
6.2	C/C++.....	59

Disclaimer

This document has been carefully prepared and checked. No responsibility can be assumed for inaccuracies. Elan reserves the right to make changes without prior notice to any products herein to improve functionality, reliability or other design aspects. Elan does not assume any liability for losses arising out of the use of any product described herein; neither does its use convey any license under its patent rights or the rights of others. Elan does not guarantee the compatibility or fitness for purpose of any product listed herein. Elan products are not authorized for use as components in life support services or systems. Elan should be informed of any such intended use to determine suitability of the products.

Software supplied with Elan PC-Cards, Compact Flash cards or USB devices is provided “as-is” with no warranty, express or implied, as to its quality or fitness for a particular purpose. Elan assumes no liability for any direct or indirect losses arising from use of the supplied code.

Copyright © 2005 Elan Digital Systems Ltd.



1 OVERVIEW

The Software Development Kit for the USBscope50 is an optional component that will allow 3rd party developers to write their own front end GUIs or device drivers.

The SDK contains the following items:

- SDK Manual (this document)
- USBscope50Drvr_W32.DLL driver library
- Interface header files for C/C++ and VB6
- Source code examples in C and VB6 to demonstrate use of the library
- Kernel drivers and INF files to support the USBscope50's USB interface chip

2 THE USB INTERFACE

2.1 USB Device Drivers

The USBscope50 uses a Silicon Labs USB interface device as its primary means of communication with the host. The exact details of the interface are not important and will not be elaborated here. However, for a 3rd party distribution it is necessary to be able to install the Silabs drivers on the target PC.

The file set used depends on the target Operating System.

The USBscope50 is configured to report the Silabs vendor ID: **10C4**

The USBscope50 reports a product ID: **F001**

The USBscope50 reports a product string: **USBscope50**

2.1.1 Windows 2000, XP

FILE	DESTINATION (same file name unless stated)	COMMENTS
slabw2k.inf	windows\inf	
slabbus.inf	windows\inf	
slabunin2k.exe	windows\system32\drivers	
slabunin.u2k	windows\system32\drivers	
slabbus.sys	windows\system32\drivers	
slabcmnt.sys	windows\system32\drivers	
slabcmnt.sys	windows\system32\drivers\slabcm.sys	rename on copy
slabcr.sys	windows\system32\drivers	
slabser.sys	windows\system32\drivers	
slabwhnt.sys	windows\system32\drivers	
slabwhnt.sys	windows\system32\drivers\slabwh.sys	rename on copy
cp210xman.dll	application folder	

2.1.2 Windows 98SE, ME

FILE	DESTINATION (same file name unless stated)	COMMENTS
slabvxd.inf	windows\inf	
slabwdm.inf	windows\inf	
slabbus.inf	windows\inf	
slabunin.exe	windows\system	
slabuninME.exe	windows\system	
slabunin.u98	windows\system	
slabbus.sys	windows\system	
slabcm95.sys	windows\system	
slabcm95.sys	windows\system\slabcm.sys	rename on copy
slabcr.sys	windows\system	
slabser.sys	windows\system	
slabwh95.sys	windows\system	
slabwh95.sys	windows\system\slabwh.sys	rename on copy
slabcomm.vxd	windows\system	
slabvcd.vxd	windows\system	
slabvcr.vxd	windows\system	
cp210xman.dll	application folder	

2.2 Digital Signing

The INF files used by the USBscope50 are adapted from the original Silabs distribution and have not been digitally signed. During installation in 2K and XP, the user must confirm that he wishes to proceed with the installation because the drivers are not signed.



3 THE DRIVER INTERFACE

3.1 Overview

The function calls needed to control the USBscope50 are contained in the USBscope50Drvr_W32.DLL module.

These sections will detail the calls available.

3.2 Terminology

A “device” will refer to an attached USB device i.e. a USBscope50.

A “channel” is the logical channel number to which a scope is assigned¹. Channels run from 1 upwards.

“byte” is an unsigned 8-bit integer value

“int” is a signed 32-bit integer value

“float” is a 4-byte IEEE floating point single precision value

“double” is an 8-byte IEEE floating point double precision value

“stringNNN” is a Unicode string of length NNN characters

“handle” is a signed 32-bit integer value

For function parameters, the term “input” refers to a value passed into the driver (ByVal in VB terms), “return” refers to the return value from a function call and “inout” refers to a variable address reference (ByRef in VB terms or type* in C terms). The “alias” section defines the call name in the DLL.

The order listed for parameters is the correct order for the function.

All functions are exported from the library as

“extern "C" _declspec(dllexport)”

This convention will link directly with VB, and will also link to C. Note that the `__stdcall` modifier must be used when declaring the C functions. See 6.2 for ready-to-use headers & modules.

¹ Also the order in which they are listed/found in the registry, which is alphabetically sorted...so all USBscope50's will be together in the registry and ordered by their serial numbers.

3.3 *Driver Functions*

3.3.1 USBscope50Drv OpenDrv

input: none
return: handle (0=fail, non zero=success)

alias: _USBscope50Drv_OpenDrv@0

This function prepares the driver module for use. It **must** be called once at the start of your program and before **any** other calls to the driver are made. During this call the cp210xman.dll support DLL is opened. This is used to recover information about the devices attached.

On success, the driver is ready to call. On failure, no calls should be made to the library and an exit strategy is required in your application (most likely the support DLL is missing).

3.3.2 USBscope50Drv CloseDrv

input: handle (as returned from OpenDrv call)

return: none

alias: _USBscope50Drv_CloseDrv@4

This function closes the driver module. It **must** be called only once at the end of your program. During this call the cp210xman.dll support DLL is released. Once you have closed the driver, you cannot make calls to its API functions.

3.3.3 USBscope50Drvr_DrvrIss

inout: string100 (returns driver issue info)

return: none

alias: _USBscope50Drvr_DrvrIss@4

This function copies a string description of the Driver's name and issue into the variable. The format is as follows:

“Name: USBscope50Drvr_W32 Ver: 1, 0, 0, 1”

In VB, the StrConv(s,vbFromUnicode) function can be used to turn the Unicode string into a VB string.

3.3.4 USBscope50Drvr_Enumerate

inout: int (forceport)
inout: int (forcechan)
return: int (number of devices found)

alias: _USBscope50Drvr_Enumerate@8

This function is used to discover any attached USBscope50 devices. The driver holds an internal array of device information, indexed from 1 to “number found” that can be queried later by the application. This index number is used to refer to a specific scope for almost all of the other driver calls and is referred to as a channel number.

The two parameters should be set to 0 under normal conditions. Then the return value will be the number of devices found.

If a particular channel and port number are required to force operation with just one scope (to support MIS mode...see **ES370 “USBscope50 User’s Guide”** for details) they can be passed into this function. The forced port is the COM port number assigned to the particular device (as found from Windows Device Manager) and the forced channel is an “arbitrary” channel number from 1 to 4. If this mode is used, the return value will be 1 if the forced setting was applied and the two parameters will hold their original values. If the forced setting was not applied, the two parameters are set to -1 and the number of devices found in total will be returned (and no forced setting will be in operation). The latter can only happen if an invalid port is requested.

3.3.5 USBscope50Drvr_GetProductName

in: int (channel)
inout: byte (name)
inout: int (length)
return: int (0=fail, non zero=success)

alias: _USBscope50Drvr_GetProductName@12

This function returns an asciiz byte array containing the name of one of the devices reported from the Enumerate call. It is used to check each device found to make sure it is a USBscope50.

channel can range from 1 to “devices found”.

name will contain the product name on success or null on failure. The byte array passed in should be capable of receiving up to 100 bytes plus the asciiz null.

The **length** returned defines the overall length of the name returned not including the asciiz null at the end.

In VB, you advised to use the following structure:

```
Dim temparray(0 To 100) As Byte  
Dim length As Long
```

```
USBscope50Drvr_GetProductName 1, temparray(0), length
```

3.3.6 USBscope50Drvr_GetSerialNumber

in: int (channel)
inout: byte (serial)
inout: int (length)
return: int (0=fail, non zero=success)

alias: _USBscope50Drvr_GetSerialNumber@12

This function returns an asciiz byte array containing the serial number of one of the devices reported from the Enumerate call.

channel can range from 1 to “devices found”.

serial will contain the product serial number on success or null on failure. The byte array passed in should be capable of receiving up to 100 bytes plus the asciiz null.

The **length** returned defines the overall length of the serial number returned not including the asciiz null at the end.

3.3.7 USBscope50Drvr_GetHWRev

in: int (channel)
inout: int (rev)
return: int (0=fail, non zero=success)

alias: _USBscope50Drvr_GetHWRev@8

This function returns the hardware revision of one of the devices reported from the Enumerate call.

channel can range from 1 to “devices found”.

rev will contain the product hardware revision on success or -1 on failure.

As an example, a rev 1.02 scope would return 0x0102 (hex)

3.3.8 USBscope50Drvr_GetPortNumber

in: int (channel)
inout: int (port)
return: int (0=fail, non zero=success)

alias: _USBscope50Drvr_GetPortNumber@8

This function returns the COM port number allocated by the O.S. to one of the devices reported from the Enumerate call.

channel can range from 1 to “devices found”.

port will contain the COM port number for the channel.

Port numbers are only used for information; driver calls refer to a channel number rather than a COM port number.

3.3.9 USBscope50Drvr_GetControllerRev

in: int (channel)
inout: int (rev)
return: int (0=fail, non zero=success)

alias: _USBscope50Drvr_GetControllerRev@8

This function returns the internal scope controller revision for one of the devices reported from the Enumerate call.

channel can range from 1 to “devices found”.

rev will contain the controller revision. For example 0x22 (hex)

3.3.10 USBscope50Drvr_OpenAndReset

in: int (channel)
return: int (0=fail, non zero=success)

alias: _USBscope50Drvr_OpenAndReset@4

This function opens the COM port associated with a channel, initialises the channel and leaves it ready for the application to initialise and configured as required.

The call can fail if the COM port cannot be opened for any reason.

channel can range from 1 to “devices found”.

3.3.11 USBscope50Drvr_PortOpen

in: int (channel)

return: int (0=closed, 1=opened)

alias: _USBscope50Drvr_PortOpen@4

This function returns the state of the port associated with a channel. The port will return closed until the OpenAndReset function is called.

channel can range from 1 to “devices found”.

3.3.12 USBscope50Drvr_Close

in: int (channel)
return: int (0=closed, 1=opened)

alias: _USBscope50Drvr_Close@4

This function closes the port associated with a channel. It should be used at the end of your application to cleanly terminate each channel that was previously opened using OpenAndReset.

channel can range from 1 to “devices found”.

3.3.13 USBscope50Drvr_InitScope

in: int (channel)
in: int (master)
return: none

alias: _USBscope50Drvr_InitScope@8

This function performs critical hardware initialisation of a scope ready for it to be used by an application. It performs an initial set up and does some internal calibration and trimming operations. A stack validation process² is advised prior initialising the scopes.

channel can range from 1 to “devices found”.

master can be 0 for a “slave” scope or non-zero for a “master” scope. If you are using just one scope, then set master to “1”. If you have detected more than one scope, you must check the stack is in a valid state (use SetDetectLine and GetDetectLine) and once this is OK you must assign exactly one scope to be the master. This is normally the first one found but is an arbitrary choice in reality (all scopes can acts as masters or slaves). Internally, the master scope is set to generate certain signals whereas slaves are set to monitor them. It is important **not** to set more than one master³ !!!

Note that it is important to call this initialisation function for all scopes one after the other without making intermediate settings per scope. This is because part of the initialisation involves a partial reset of the internal scope controller on each channel to guarantee synchronisation with the other channels in the stack.

² To makes sure that all scopes are correctly connected to each other via the stack

³ You will get very strange behaviour from one or all scopes, although no damage will result.

3.3.14 USBscope50Drvr_SetDetectLine

in: int (channel)
in: int (master)
in: int (state)
return: none

alias: _USBscope50Drvr_SetDetectLine@12

This function allows control over a general purpose IO pin that runs between all stacked scopes. Any scope may drive it and/or sense it. It is used to detect whether all scopes are connected together properly in the stack. For single scope use, it is not relevant.

channel can range from 1 to “devices found”.

master is set to 0 to “float” the detect line on that scope i.e. make it high impedance and so allow another scope to drive it (in this case state will be don’t care). Set master to a non-zero value to make that scope drive the detect line with the state requested.

state can be 0 or 1 (only).

As a general strategy, this function is best used directly after OpenAndReset has been done for all channels. Then, set all detect lines to “float” i.e. master=0. Then for just the master channel, set master=1 and set the detect line to 0 and then to 1. For each of these detect line states use GetDetectLine to read all the scope’s detect sense inputs. Any scope that does not report a state that follows the master’s is deemed not in the stack and hence invalidates the entire stack⁴. In this case, the application should follow a suitable exit strategy and close all open scopes.

⁴ The stack requires ALL scopes to be properly joined together.

3.3.15 USBscope50Drvr_GetDetectLine

in: int (channel)
return: int (detect line's sensed state)

alias: _USBscope50Drvr_GetDetectLine@4

This function returns the detect line's sense for a particular scope. It can be used in conjunction with SetDetectLine to form the basis of a stack-validation procedure before scopes are initialised further. For single scope use, it is not relevant.

channel can range from 1 to “devices found”.
The return value is the state of the detect line i.e. 0 or 1.

3.3.16 USBscope50Drvr_SetBaseAdcClk

in: int (channel)
in: int (clock required)
return: none

alias: _USBscope50Drvr_SetBaseAdcClk@8

This function sets the scope's base clock rate for acquisitions.

channel can range from 1 to “devices found”.

clock must be either 50, 25 or 10, no other choices are allowed. These represent the base adc clock rate in MHz.

This call, in conjunction with the SetDecimationRatio, is used to set the overall sampling speed.

Only 50MHz may be used in RIS mode.

3.3.17 USBscope50Drvr_GetBaseAdcClk

in: int (channel)

return: int (clock in use)

alias: _USBscope50Drvr_GetBaseAdcClk@4

This function returns the clock in use by the ADC.

channel can range from 1 to “devices found”.

The return will be 50, 25 or 10. These represent the base adc clock rate in MHz.

3.3.18 USBscope50Drvr_SetDecimationRatio

in: int (channel)
in: int (ratio required)
return: none

alias: _USBscope50Drvr_SetDecimationRatio@8

This function sets the division ratio used by the adc sampling logic. The ratio **must** be set to 1 when using 50 and 25MHz for the base adc clock. Only the 10MHz clock supports decimation, which can range from 1 to 400000. A ratio of 1 means that scope will sample at 10MSPS, and ratio of 400000 means that the scope will sample at (10/400000)MSPS. Only certain ratios are possible.

channel can range from 1 to “devices found”. All scopes in a stack **must** be set to the same sample rate for predictable results.

ratio can only be one of the following values:
1,2,4,10,20,40,100,200,400,1000,2000,4000,10000,20000,40000,100000,200000,400000

The following table shows the rates available (note that the first 4 vary the screen zoom to affect the T/Div adjustment):

T/Div	Pts/Div	Time zoom ⁵	Base clock	Ratio
200ns	10	10	50	1
400ns	20	5	50	1
1us	50	2	50	1
2us	100	1	50	1
4us	100	1	25	1
10us	100	1	10	1
20us	100	1	10	2
40us	100	1	10	4
100us	100	1	10	10
200us	100	1	10	20
400us	100	1	10	40
1ms	100	1	10	100
2ms	100	1	10	200
4ms	100	1	10	400
10ms	100	1	10	1000
20ms	100	1	10	2000
40ms	100	1	10	4000
100ms	100	1	10	10000
200ms	100	1	10	20000
400ms	100	1	10	40000
1s	100	1	10	100000
2s	100	1	10	200000
4s	100	1	10	400000

This table refers only to real time single-shot sampling. The sample period can be computed from T/Div divided by Pts/Div.

Random Interleaved Sampling or RIS mode is discussed in section 4. For reference, the RIS sampling rates are listed below:

T/Div	Pts/Div	Time zoom	Base clock	Ratio
4ns	4	25	50	1
10ns	10	10	50	1
20ns	20	5	50	1
40ns	40	2.5	50	1
100ns	100	1	50	1

⁵ Note the Pts/Div and zoom factors are essentially the same parameter...they are listed separately to clarify the reduction in the number of points plotted when zoom is employed.

3.3.19 USBscope50Drvr_GetSamplePeriod

in: int (channel)
in: int (current (or stopped))
in: int (haltperiod (when stopped))
return: float (period in seconds)

alias: _USBscope50Drvr_GetSamplePeriod@8

This function returns the sample period set for the scope. Rather than just return the currently in-use period, the function also allows return of the period that was in use when the scope's acquisition was halted.

channel can range from 1 to “devices found”.

current is treated as a flag; pass in zero to have the function return a value equal to the **haltperiod** parameter, or pass in “current” as non-zero to return the current sample period in use by the scope.

This function is normally used with “current” set to 1. However, if the application sets the scope into a halted state, it is important for the application (not the driver) to remember the period at that point and save it in a variable⁶. Then, by passing this value into each use of this function within the application and by controlling the “current” parameter to reflect the scope's acquisition state, the usefulness of the function is extended as it now returns the period currently in use **or** the period that was used to acquire the data when the scope was halted. This allows common code to be used when calling the GetSamplePeriod function even if the user has halted the scope and has subsequently changed the timebase control.

This function always returns 1ns when in RIS mode because this is the fundamental time resolution, changes in the displayed timebase are achieved by simply altering the number of points per division plotted on the screen (i.e. the data is zoomed on the time axis as listed in 3.3.18)

⁶ To allow the data to be saved perhaps, or to correctly display spectrum FFT bounds. It is also useful so as to allow the user to adjust the timebase when the scope is in a stopped state, and have this action work as a virtual timebase “zoom” effect. You can try this with the USBscope50 software.

3.3.20 USBscope50Drvr_GetSampleDepth

in: none

return: int (samples)

alias: _USBscope50Drvr_GetSampleDepth@0

This function returns the number of sample points per full acquisition. For the USBscope50 it always returns 3000. The application can use this call to size dynamic arrays that will be used to hold sample data.

3.3.21 USBscope50Drvr_SetUpFrontEnd

in: int (channel)
in: int (gain)
in: int (dc)
in: int (gnd)
in: int (ris)
return: none

alias: _USBscope50Drvr_SetUpFrontEnd@20

This function configures the scope's analogue front-end. It controls the coupling, gain and sampling mode for each scope. If you make the same setting twice or more in a row, only the first call will change the hardware.

channel can range from 1 to “devices found”.

gain can be 0,1 or 2 which equate to front end attenuation settings of 1, 10 or 100 (equating to full scale ranges of +/- 0.3, 3.0, 30.0V with a “x1” scope probe, or +/-3.0, 30.0, 300.0V with a “x10” scope probe). The ADC resolution for each range is 8 bits which gives a minimum ADC step size of approximately 23.6mV, 236mV and 2.36V for the x10 ranges.

dc is a flag: zero gives AC coupling; non-zero gives DC coupling.

gnd is a flag: zero sets for a normal input; non-zero grounds the scope channel internally.

ris is a flag: zero sets the scope for normal single-shot acquisition; non-zero sets the scope into “random interleaved sampling” or RIS mode. More on this mode in section 4.

Remember that the higher the attenuation setting, the larger the minimum ADC step size. This will mean that any small offset in the front end will effectively look amplified in the higher ranges i.e. the offset error with a grounded input probe will look larger.

3.3.22 USBscope50Drvr_SetOffset

in: int (channel)
in: float (offsetpct)
return: none

alias: _USBscope50Drvr_SetOffset@8

This function configures the scope's front-end offset. Adjusting the offset allows a channel to be positioned up or down the vertical scale. The offset is achieved by means of d-to-a converter in the scope's front-end circuitry.

channel can range from 1 to “devices found”.

offsetpct is the desired offset, with a setting of 100.0 putting the channels trace to the very top of the vertical range, and value of 0.0 setting it to the middle of the range and a setting of –100.0 setting it to the very bottom of the range.

Note that the offset setting is made irrespective of the front-end attenuator range setting, making application software very easy to structure⁷. Also note that when you offset a channel the sample values that are returned by the scope will also offset (because the offset is actually added to the channel voltage in the front end). The application must therefore subtract any offset when outputting readings derived from the data. For example, if the offset is used to move the channel half way up the vertical scale, then any readings computed from the sample data would need to have this amount of voltage subtracted from them (i.e. the reading is relative to the offset zero point). This is also true when performing math functions on the channel data...the math functions must work out readings relative to the offset zero point.

⁷ In the USBscope50 software, the control for offset actually uses bound of +127 to –127 but these are simply converted to 100.0 and –100.0 respectively before being passed to this function.

3.3.23 USBscope50Drvr_SetTrigMaster

in: int (channel)
in: int (master)
return: none

alias: _USBscope50Drvr_SetTrigMaster@8

This function configures a scope to be a trigger master or a trigger slave.

channel can range from 1 to “devices found”.

master is a flag: set to zero to set up the scope to be a trigger slave; set to non-zero to make the scope be a trigger master.

Exactly one scope at a time can be the trigger master, the rest **must** be set as slaves. This setting is analogous to the trigger channel selection on a regular scope except that the setting must be made by configuring each scope independently (so that only one is the master).

3.3.24 USBscope50Drv_GetTrigMaster

in: none

return: int (channel)

alias: _USBscope50Drv_GetTrigMaster@0

This function returns the channel number of the scope currently set to be the trigger master.

3.3.25 USBscope50Drvr_SetTrigType

in: int (channel)

in: int (type)

return: none

alias: _USBscope50Drvr_SetTrigType@8

This function configures the scope's trigger types. It is only applicable for the scope that is acting as the trigger master as it is responsible for generating an overall trigger signal. However, for ease of programming it is usually simpler to configure all scopes to the same trigger type. It also only applies if the scope is put into normal trigger mode. In free-run mode the setting has no relevance because the scope will acquire immediately when AcquisitionStart is called.

The **channel** should be set to the channel designated as the trigger channel, although it is OK to set all scopes to the same trigger type. **type** can be set to 0,1,2 or 3 corresponding to “less than”, “greater than”, “neg edge” and “pos edge” respectively.

3.3.26 USBscope50Drvr_SetNormTrig

in: int (channel)
in: int (norm)
return: none

alias: _USBscope50Drvr_SetNormTrig@8

This function configures the scope's trigger mode to be normal or free running. If free-run is selected, trigger delay is automatically set to zero in the hardware and the pre-trigger depth is also zeroed. On selecting normal trigger, trigger delay is not adjusted and the desired pre-trigger depth must be re-configured by the application.

channel can range from 1 to “devices found”. All scopes must be set to the same trigger mode.

norm is a flag: set to zero to configure free-run; set to non-zero to configure external “normal” triggering.

Free running mode does not rely on any external input signal conditions to cause the scope to complete an acquisition, the application only needs to call AcquisitionStart to commence an acquisition and it will complete as soon as all samples have been collected by the scope. Conversely, the normal mode will cause the scope to block the completion of an acquisition until the external input signal meets the triggering criteria set via SetTrigType and the trigger threshold.

Note that “auto” triggering is achieved in the USBscope50 software by periodically putting the scope into normal trigger mode and waiting for some fixed period of time to see if it triggers. If it does trigger, the scope is left in normal mode, otherwise the scope is reverted back to free running mode.

3.3.27 USBscope50Drvr_SetTrigThreshold

in: int (channel)
in: float (threshpct)
return: none

alias: _USBscope50Drvr_SetTrigThreshold@8

This function configures the scope's trigger threshold voltage for use when in normal triggering (it has no relevance in free-run mode). The trigger threshold voltage is applied to a hardware comparator inside the USBscope50 and the output from this comparator drives the triggering circuitry. The threshold is only important for the scope that is set as the trigger master, although for simplicity, the application may choose to configure all scopes to the same trigger threshold.

The **channel** should be set to the channel designated as the trigger channel, although it is OK to set all scopes to the same trigger threshold.

threshpct is the percentage of the current input voltage range. A value of 100.0 will set the threshold to the very top of the vertical deflection, a value of 0.0 will set it to the middle of the deflection, and a value of -100.0 will set it to the very bottom of the vertical deflection.

The setting requested is made irrespective of the current front-end configuration or channel offset applied. This makes application programming easy as no scaling or adjustment needs to be made as the channel is offset or as the attenuator is changed. If it is desired to move the trigger threshold with the trigger channel's offset, then this can be achieved via a simple numerical addition of the channel's offset percentage and the desired trigger threshold percentage. Values that overflow 100 or -100 are clamped by the driver. Remember that the full range of adjustment will still be +/-100% so if the channel is offset by say -50% and your application control only allows a +/-100% adjustment range, then the highest trigger threshold you could set would be +50%. The application would need to alter the adjustment range to suit to ensure that +100% could still be reached.

3.3.28 USBscope50Drvr_GetTriggeredStatus

in: int (channel)

return: int (0=not triggered, 1=triggered)

alias: _USBscope50Drvr_GetTriggeredStatus@4

This function returns the status of an internal latched signal showing whether the scope has triggered or not.

channel can range from 1 to “devices found”.

3.3.29 USBscope50Drvr_SetPreTrigDepth

in: int (channel)
in: float (buffpct)
return: none

alias: _USBscope50Drvr_SetPreTrigDepth@8

This function set up the number of samples that are preserved during an acquisition that were captured before the trigger point. This only applies when the scope is in normal triggering mode.

channel can range from 1 to “devices found”.

buffpct is the percentage of the buffer before the trigger point that you wish to preserve. A setting of 0.0 means that the trigger point will be the first sample preserved, a setting of 50.0 means that half the buffer will be preserved. The largest value that can be set is 99.0.

Pre-triggering refers to the way the scope saves data into its buffer. The scope uses a circular buffer to continuously capture sample data once the AcquisitionStart command is issued. The hardware then uses the value set for the pre-trigger depth to decide when it has acquired enough samples to allow a trigger event to happen (the scope “arms” itself internally). Once the trigger event does happen, samples continue to flow into the buffer but will cease once the number of samples acquired after the trigger event, equals the buffer depth (3000) minus the pre-trigger depth. When the sample buffer is read-out, the first samples will be from a time prior to the trigger event though to the most “recent” samples that happened after the trigger event⁸.

⁸ It is worth testing this with the USBscope50 software so that you can fully appreciate the effect of pre-trigger. Set the scope to normal triggering and observe a square wave...now try adjusting the pre-trigger slider at the bottom edge of the display. You will see that its effect is to move the waveform left and right relative to the trigger point.

3.3.30 USBscope50Drvr_SetTriggerDelay

in: int (channel)
in: float (delay)
return: none

alias: _USBscope50Drvr_SetTriggerDelay@8

This function sets up a delay that the scope will wait, after a trigger event, until it completes its acquisition sweep. This only applies when the scope is in normal triggering mode. Only the trigger channel controls the trigger delay but for convenience it is OK to set all scopes to the same delay if desired.

The **channel** should be set to the channel designated as the trigger channel, although it is OK to set all scopes to the same trigger delay. **delay** is the absolute delay in seconds. The scope has an internal delay counter that is 16-bits in length. The value loaded into the counter is computed by the driver by taking the requested delay and dividing it by the current sample period⁹. Hence the delay is always rounded (or truncated) to the nearest whole sample period. Requesting a delay that is more than 65535 times the current sample period will give unpredictable results.

⁹ The USBscope50 software uses a control with a range from 0 to 65535 which it turns into an absolute delay by multiplying by the current sample period. If you do the same remember to re-compute your displayed delay time if you change the timebase.

3.3.31 USBscope50Drvr_AcquisitionStart

in: int (channel)

return: none

alias: _USBscope50Drvr_AcquisitionStart@4

This function is called to begin an acquisition sweep. All channels must be included with the trigger master channel being started **last**.

channel can range from 1 to “devices found” and must be called for all scopes and importantly, so that the last scope started is the trigger master.

Once the acquisition has been started, the status can be checked by calling GetAcquisitionState.

An acquisition **must** be ended by calling AcquisitionEnd. Note also, that changing the scope’s sample rate while an acquisition is in progress is to be avoided; call AcquisitionEnd before making such a change.

This function automatically handles both single shot and RIS acquisitions i.e. call it to start an acquisition when in either mode.

3.3.32 USBscope50Drvr_AcquisitionEnd

in: int (channel)

return: none

alias: _USBscope50Drvr_AcquisitionEnd@4

This function is called to end an acquisition sweep. All channels must be ended but the order is not important.

channel can range from 1 to “devices found” and must be called for all scopes in any order that is convenient.

This function must be called after an acquisition has completed or if an acquisition needs to be terminated for any reason. It must also be used to terminate any acquisition prior to adjusting the scope's sample rate.

This function automatically handles both single shot and RIS acquisitions i.e. call it to stop an acquisition when in either mode.

3.3.33 USBscope50Drv _GetAcquisitionState

in: int (channel)

return: int (0=stopped, 1=acquiring)

alias: _USBscope50Drv _GetAcquisitionState@4

This function is called to enquire the current status of an acquisition. It can be used to poll the scope to see when it has completed an acquisition started with a call to AcquisitionStart. When using more than one scope it is advisable to get the status from the master channel although this is not essential; the status returned will apply to all the channels in the stack.

channel can range from 1 to “devices found” but the master channel is the best scope to poll for this status.

3.3.34 USBscope50Drvr_GetBufferBlocks

in: int (channel)
inout: float (data array)
in: int (blocks)
return: none

alias: _USBscope50Drvr_GetBufferBlocks@12

This function is used to fetch zero or more blocks of 512 data samples into a user's buffer.

channel can range from 1 to “devices found”.

data is a pointer to an array of floats, that **must** be large enough to hold the requested total size of (512*blocks) floats

blocks is the number of 512 sized sample blocks to recover.

After an acquisition has ended or been terminated this function is used to get the sample data points into an array. Each data point can range from +128.0 down to -127.0 although the exact range may vary a little (+/-5%) from this theoretical band due to the scaling and corrections applied to the data points as they are unloaded from the hardware. Regardless of the range variation, +128 should be considered as full-scale positive and -127 as full scale negative.

To recover the entire data buffer use a block count of 6. This will return 3072 points, the last 72 of which can be discarded.

Note that each channel must have its data recovered individually. A faster method is also available, see GetBufferBlocksMultiChan.

For VB users, do not try and pass in one dimension of a multi-dimensional array as the internal array ordering between VB and driver may be different. Instead, create a temporary one dimensional array of singles and pass this in as temp(0). After the call, copy the data from temp to your multi-dimensional array using a loop.

3.3.35 USBscope50Drvr_GetBufferBlocksMultiChan

in: int (firstchannel)
in: int (lastchannel)
inout: float (data array)
in: int (blocks)
return: none

alias: _USBscope50Drvr_GetBufferBlocks@16

This function is used to fetch zero or more blocks of 512 data samples into a user's buffer, for one or more channels.

firstchannel can range from 1 to “devices found”. This is the first channel who's data buffer will be collected.

lastchannel can range from 1 to “devices found” and must be higher or equal to firstchannel. This is the last channel who's data buffer will be collected.

data is a single, one-dimensional array into which the data will be packed, in channel order. It **must** be large enough to hold $(512 * \text{blocks} * (\text{lastchannel} - \text{firstchannel} + 1))$ floats

blocks is the number of 512 sized sample blocks to recover.

This function uses some processing optimisations to increase the data throughput rate for multi-scope applications. It can still be used for single channels if required by setting firstchannel and lastchannel to the same value.

The data is packed into the data array so that data for firstchannel lies between indexes 0 and $(512 * \text{blocks} - 1)$. Data for $(\text{firstchannel} + 1)$ lies between $(512 * \text{blocks})$ and $(512 * \text{blocks} * 2 - 1)$ etc.

3.3.36 USBscope50Drvr_GetBufferRIS

in: int (channel)
inout: float (data array)
in: int (samples)
inout: float (risbin)
inout: int (warning)
inout: int (reject)
return: none

alias: _USBscope50Drvr_GetBufferRIS@24

This function is used to fetch zero or more bytes data samples into a user's buffer after an RIS acquisition.

channel can range from 1 to “devices found”.

data is a pointer to an array of floats, that **must** be large enough to hold the requested total size of (samples) floats

samples is the number of samples to recover, from 0 to sampledepth.

risbin is the “bin” number that this data set belongs to, from 0.000 to 19.999. This value's fractional part can be truncated or used to alter the way the software uses this sample data in combination with previous sample data.

warning is always set to 0 by this function. Do NOT pass NULL!

reject is always set to 0 by this function. Do NOT pass NULL!

3.3.37 USBscope50Drvr_GetSamplesSinceTrigger

in: int (channel)
return: int (samples)

alias: _USBscope50Drvr_GetSamplesSinceTrigger@4

This function is used to get the number of cumulative samples that have been acquired since a trigger event (or the acquisition start in free-run mode). It is used only for slow timebase settings e.g. slower than 40ms/div and can be used to implement a roll-mode display i.e. one where the sample buffer is acquired in a quasi-continuous fashion and re-painted a section at a time.

channel can range from 1 to “devices found”. Normally the channel will be the master channel but in fact this is arbitrary.

3.3.38 USBscope50Drvr_GetBufferIncremental

in: int (channel)
inout: float (data array)
in: int (samples)
in: int (numsofar)
return: none

alias: _USBscope50Drvr_GetBufferIncremental@16

This function is used to fetch zero or more data samples into a user's buffer for use in a roll-mode scope display. It is not intended for any other use. See GetSamplesSinceTrigger.

channel can range from 1 to “devices found”.

data is a pointer to an array of floats, that **must** be large enough to hold the requested total size of (samples) floats

samples is the number of sample points to recover.

numsofar is the running total maintained by the application of the data points collected since trigger. The data is copied into the data array at the index (numsofar).

When running in roll-mode, the application repeatedly checks the number of samples that have been collected by the hardware since trigger (or acquisition start for free-run mode) using the GetSamplesSinceTrigger call. This value is checked to see if it has advanced sufficiently to make it worth collecting and/or displaying (or saving) these new data points. For the USBscope50 software, a threshold of 10 new points is used. Then, the GetBufferIncremental call is used to recover these samples onto the “end” of the data buffer in use by the application. The new samples are copied to the array at index (numsofar). The application must ensure that numsofar is zeroed before each acquisition sweep, and it then increases by no more than the number of incremental samples reported since trigger prior to the next GetBufferIncremental call. Additionally, the GetAcquisitionState call is used to see when the overall acquisition finishes, so that the process can start over again.

3.3.39 USBscope50Drvr_SetCalSource

in: int (channel)

in: int (state)

return: none

alias: _USBscope50Drvr_SetCalSource@8

This function is used to enable and disable the 1KHz square wave used for probe compensation / calibration.

channel can range from 1 to “devices found”.

state is a flag: set to 0 to turn the source off (low); set to non-zero to turn the source on (3V pk-pk).

You are strongly advised to leave the source off whenever possible to avoid its fast edges coupling into high-impedance sources.

3.3.40 USBscope50Drvr_SetLEDMode

in: int (channel)

in: int (mode)

return: none

alias: _USBscope50Drvr_SetLEDMode@8

This function is used to control the scope's LED.

channel can range from 1 to “devices found”.

mode can be 0 (off), 1 (slow blink), 2 (fast blink), 3 (on)

3.3.41 USBscope50Drvr_FFT

inout: double (fft array)
in: int (length)
in: double (dB)
in: int (window)
inout: int (peakbin)
return: none

alias: _USBscope50Drvr_FFT@24

This function is used to do an FFT on the time data.

fftarray holds the time domain sample data on the way in, and contains half the number of frequency domain data points on the way out (i.e. the FFT is conducted in-place). The output data is magnitude only. The array is indexed from zero. On the way out, the zero'th bin is a DC term i.e. the magnitude of the response at $f=0\text{Hz}$. The bins are spaced at the scope's sample rate in Hz divided by the "length" (i.e. if the sample rate is set to 50MSPS or 50MHz, then for a 2K point FFT, each of the 1024 frequency bins is $50/2048\text{MHz}$ wide or 24.4KHz approx). You are advised to normalise the input data on the way in to avoid rounding errors; the USBscope50 software takes the raw sample data and divides each point by 255 to give an input data range of ± 0.5 .

length is the number of time domain points in the input data set. This value **must** be a power of 2. The number of frequency bins returned will be half this value.

dB is a value that can be used to convert the magnitude response from linear to decibels. Pass in 0 to keep the FFT result in linear form i.e. the bins will hold the amplitude data in volts for each bin. Alternatively, pass in a positive value that will represent "full scale amplitude" to get the result in decibels. For the normalised scheme mentioned above, the dB figure would be passed as 0.5.

window is the desired spectral window to use 0: rectangular (i.e. no shaping) 1: Hanning 2: Hamming 3: Triangular 4: Welch.

peakbin reports the bin with the highest value in it, excluding bin zero.

3.3.42 USBscope50Drvr_ReScaleSampleData

inout: float (olddata array)
inout: float (newdata array)
in: int (points)
in: float (oldofspct)
in: float (newofspct)
in: float (oldvolts)
in: float (newvolts)
return: none

alias: _USBscope50Drvr_ReScaleSampleData@28

This function is used to re-scale sample data. It is used when the scope has been halted but the user then adjust the channel offset or the channel volts per division. Because these actions are no longer acting on the hardware to change the acquired data, the software must re scale the data. Note that time domain scaling (caused by adjusting the timebase) is dealt with by zooming in or out on the time axis.

olddata is the original un-scaled data array that was saved when the scope was halted. The data points will be the +128/-127 range as described previously.

newdata is the new scaled data array that was computed from the olddata array.

points is the number of sample points to re-scale.

oldofspct is the channel offset in percent when the scope was halted

newofspct is the desired channel offset in percent

oldvolts is the channel volts/division setting when the scope was halted. This figure would be 0.3,3 or 30 for example. The exact values are actually not important because only the ratio with newvolts matters.

newvolts is the desired channel volts/division setting

4 RIS Mode

4.1 Overview

RIS mode uses the scope sampling at 50MSPS, to achieve an effective time resolution of 1ns, 20 times better the natural time resolution in single shot mode.

Please refer to ES370 USBscope50 User's Guide, for details on the operational limitations of RIS mode. It is suggested that you try the RIS mode using the USBscope50 application software before trying to code RIS in your own application. This will allow you to become familiar with its capabilities and understand what it can and can't do.

4.2 Principals of Operation

The scope includes special hardware that is able to measure the time between when the analogue input to the scope causes a hardware trigger, and the time of the first 50MSPS data sample. This time is expressed as a "bin" number between 0.000 to 19.999 i.e. there are 20 bins that give rise to the effective 20-fold improvement in time resolution. If the bin number is 0, the trigger and first sample were coincident, if the bin number were 10.5 there was a 10.5ns delay, and if the bin number were 19.999 then the delay was nearly a full 20ns sample period. The measured delay is, of course, completely random and so multiple acquisitions are needed until a "hit" is made on each of the 20 bins. It can therefore take some time for this to happen. Software should perform multiple acquisitions as quickly as possible to build up the waveform.

To use the scope in RIS mode, use the SetUpFrontEnd call (3.3.21) and set the ris flag to 1. Configure the sample rate to 50MSPS. The scope **must** be set to normal triggering and the pre-trigger depth must be set to 0. Apart from this the scope's configuration is performed the same as for single shot mode.

To acquire data, simply use the AcquisitionStart call (3.3.31) and end it (once it has finished) using the AcquisitionEnd call (3.3.32). Collect the data from the acquisition using the GetBufferRIS call (3.3.36). If you plan to keep the depth of your acquisition constant

at 3000 points then there is no point getting more than 3000/20 samples from this call i.e. pass in 150 as the number of samples to collect. Collecting more points will merely slow the data recovery down as they have to be sent over the USB. As will now be explained, these 150 samples will get spread out in time by a factor of 20.

4.3 Organising the Data

Each acquisition will return a set of data, probably 150 points as noted above.

In software, your sample data array is 3000 points deep, so where do you put these 150 data points? The answer is that you spread them out into every 20th location in the array, and the starting point in the array is given by the bin number reported when getting the 150 points. For example, if you get 150 points and the bin number was 0.000, then put sample[0] in array[0], sample[1] in array[20], sample[2] in array[40], sample[3] in array[60] etc.

If the bin had been 14.000, then put sample[0] in array[14], sample[1] in array[34], sample[2] in array[54], sample[3] in array[74] etc.

Keep taking acquisitions and for each one, spread the data out into your array as described. Over a period of time, the waveform will “build up” (as more and more “bins” get hit).

There are two choices on how to process the floating point bin number. You could simply throw away the fractional part to yield a bin number between 0 and 19 inclusive. This can be used as a direct index into your array. A downside to this approach is that the bin value becomes quite coarse and this can show as small step discontinuities in the displayed waveform near to fast changing edges.

A more sophisticated approach is to use the fractional part of the bin number to decide how much “weight” to place on the new sample relative to the sample that is already in that array position i.e. the old data. This technique essentially provides filtering between the old

and new data and helps to smooth such discontinuities. The USBscope50 application software uses such a technique and applies a weighting that ramps linearly from 0.000 when frac=0.000, to 1.000 when frac=0.500, and then ramps back down to 0.000 again when frac=0.999. The weighting is then used as follows:

$$\text{Array}[n] = \text{Array}[n] * (1.000 - \text{weighting}) + \text{Sample}[m] * (\text{weighting} - 1.000)$$

As can be seen, if the bin reported is “right in the middle of the bin” then the new sample replaces the existing array data. If the bin reported is at “either end of the bin” (i.e. frac is near 0.000 or 0.999), the old sample in the array is preserved, and at points in-between the new and old samples are averaged together in proportion to the “nearness” to the bin’s centre.

It is important to realise that the bin number reported will always have a level of uncertainty (inaccuracy) in it, and that uncertainty means that the fractional part of the bin number should not be taken as an absolute, it is just an approximation. The number of bins **could** be increased from 20 by using the fractional part with more significance. This will slow the overall “waveform build up” time because the randomness of the bin hits makes it less likely that any one bin will get hit. The number of bins **could** be reduced from 20 by placing less significance on the fractional part and by merging several bins. This will reduce the “waveform build up” time at the expense of lower time domain resolution.¹⁰



¹⁰ We will not support any technical queries arising from deviation from the standard 20 bin mode.


5 General Programming Considerations

5.1 *Saved Scope Settings*

The driver does not save or restore scope settings. It is up to the application to perform this function, using the registry, an INI file or other method as appropriate.

5.2 *Leaving Ports Open*

During development a common problem is that each channel's port is opened but never closed, perhaps due to a logical programming error or because the application is terminated prematurely. Leaving the COM port open usually means it cannot be opened again next time, even if the scope is unplugged and re-plugged. The effect is that Enumerate will return zero scopes. The remedy is a reboot.



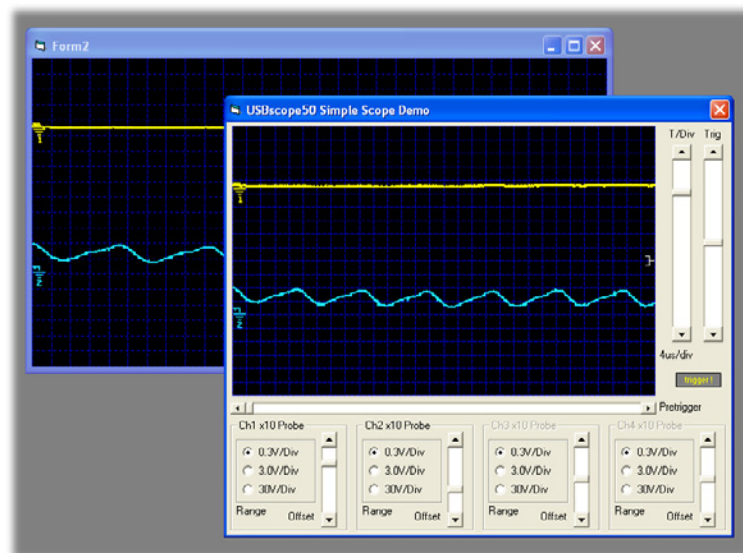
6 Example Code & Header Files

6.1 Visual Basic6

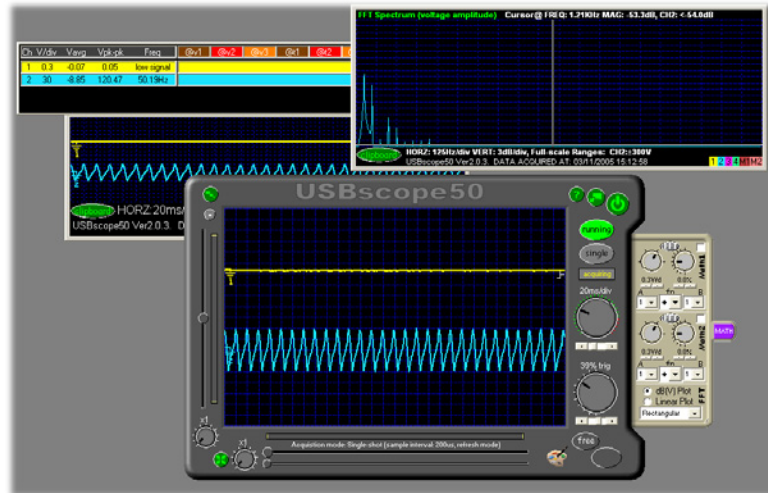
A single .bas header file is needed to access the library functions:

USBscope50Drvr_if.bas

Included in the SDK are two VB6 projects. The first is a demo of a simple multi-channel scope. It does not attempt to cover all the features that USBscope50 does, but is useful to show the basis of a working application. The demo uses a font file to create the ground channel markers on the plot. The file is in the demo folder but must be registered with Windows before it will work (copy it to windows\fonts)



The second demo is a frozen release of the full USBscope50 application software. As such, it supports many more the features that the scope can perform, but inevitably is many times more complex that the simple scope demo. Note too that the full project uses an extra header file, which contains some undocumented library calls, used primarily for debugging and diagnostics.



6.2 C/C++

A single .h header file and a single .cpp file are needed to access the library functions:

```
USBscope50Drvr_if.h  
USBscope50Drvr_if.cpp
```

The cpp file declares the external functions, which are themselves instantiated as pointers to functions using the LoadLibrary and GetProcAddress methods to attach to the DLL.

In order to use the function calls, include the .h file and the .cpp file in your project and call USBscope50Drvr_OpenDrvr as documented above at 3.3.1. On exit be sure to call CloseDrvr.

The C demo is a simple 32-bit console application to recover some acquisition data and average it to DC value. The overall configuration of the scopes and the steps to validate the stack are clearly shown there, together with a basic free-running acquisition. The methods used to implement auto-triggering are more complex, and requires the application to periodically put the scope into a normal triggered mode, wait to see if it triggers and then either time out and revert to free-run mode or to leave the scope in normal trigger if it did trigger. Code to demonstrate mode this can be found in one of the VB demo files. Open the form1.frm file in a text editor (find it under examples\vb6\simplescope). You can ignore the first part of the file as it is simply the form constructor information. Scroll down and search for a function called RunTimer_Timer. In there you will see the logic required for auto triggering¹¹.

¹¹ Don't worry if you're not a VB programmer...the structure of the code is the only important concept and much of that can be very easily mapped to C by an experienced coder. Note that the RunTimer_Timer function is invoked every 10ms using a VB timer object...effectively this routine drives the entire scope acquisition and display.