BitScope Library V2 API Guide

The **BitScope Library** is a powerful and easy to use **BitScope API** [1] for:

- **Windows XP, 7 & 8** (Windows XP works but is no longer officially supported)
- **Linux** (all major distributions including Debian, Ubuntu, Fedora and RedHat)
- **Mac OS X** (Snow Leopard and all later versions),
- **ARM platforms** including **Raspberry Pi**.

It provides off-the-shelf *language bindings* for the:

- **C** [2], **C++** [3], **Python** [4] and **Pascal** [5] programming languages.

Other language bindings are available upon request.

BitScope Library V2 API Guide
- Library Overview
- BitScope Programming
  - (1) Library and Device Initialization
  - (2) Device Programming and Setup
  - (3) Trace Programming and Capture
  - (4) Acquiring Data from the Device
  - (5) Closing Devices and the Library
- Probe Files and Link Specifications
  - Probe File Locations
    - Windows Probe File Locations
    - Linux, Mac OS X and Raspberry Pi Probe File Locations
  - Link Specification Syntax
  - Probe File Defaults
- Device Simulations
- Diagnostics
- Error Recovery
- Code Examples
- Troubleshooting
  - The Library can’t find my BitScope!
  - Which BitScope am I really connected to?
  - Why does it take so long to connect?
  - What parameter values does the library actually use?
  - Can I use the library in a multi-threaded environment?
  - Help! I don’t understand how BL_xxxx works?
  - This is not working, I’ve found a Bug!
- Library Reference

BL_Acquire - dump data from the device
BL_Close - close all open devices
BL_Count - count devices, channels, or ranges
BL_Coupling - select the channel source coupling
BL_Delay - assign post-trigger delay
BL_Enable - change channel enable status
BL_Error - return most recent error (if any)
BL_Halt - all any pending or prevailing device activity
BL_ID - return the selected device ID
BL_Index - assign the buffer offset (for dumps)
BL_Initialize - initialize the library (optional)
BL_Intro - assign the pre-trigger size (intro region)
BL_Log - dump the pending log
BL_Mode - select and trace mode
BL_Name - return the device link name
BL_Offset - assign channel offset
BL_Open - open one or more devices
BL_Range - select the channel range
BL_Rate - assign the sample rate
BL_Select - select a device, channel or source
BL_Size - assign the capture size (samples)
BL_State - return capture engine state
BL_Time - assign the capture duration (seconds)
BL_Trace - initiate capture
BL_Trigger - set up the trigger
BL_Version - return the version
License
Version

Library Overview

The **Library** provides *full programmed access* to the **BitScope Capture Engine**. It is implemented using a small set of portable functions. Programming BitScope to capture waveforms and logic is simply a matter of calling the **correct sequence of these functions** from your program. There are no complicated data structures, parameter lists or callbacks.

See **BitScope Programming** and **Library Reference** for details.

BitScope Programming

Regardless of the device and library functions used, BitScope programming always

follows a familiar sequence:

> **(1) Initialize -> (2) Setup -> (3) Trace -> (4) Acquire -> (5) Close**

The library must first be **(1) initialized** and one or more devices opened. For each open device the required capture conditions must be **(2) set up** before a **(3) trace** is initiated to capture or generate the signals. When the trace completes the data may be **(4) acquired** to the host computer for display and analysis. Initialization and setup is normally only done once but the trace and acquire steps may be repeated as often as required. When finished the devices should be **(5) closed** which releases the resources allocated to them.

(1) Library and Device Initialization

Before use, *the library must be initialized* and one or more *devices opened*:

- **BL_Initialize** - initialize the library
- **BL_Open** - open one or more devices

The BL_Initialize call is implicit (in most cases); calling BL_Open to open the device(s) is sufficient to initialize the library.

When the device is open, *information about it may be obtained*:

- **BL_Name** - return the device link name
- **BL_Version** - return version information
- **BL_ID** - return the selected device ID
- **BL_Count** – count devices, channels and ranges

With the exception of BL_Count (see below), these functions are necessary only for reporting information to the application and identifying the device.

(2) Device Programming and Setup

After one or more devices have been opened they must be *set up for use*. This is the most detailed programming step but it is only needed once after opening the device unless recovering from an error. The first thing your program needs to know is how many devices, channels, ranges and other properties are available (they may not all be the same). The number of open devices may be obtained via BL_Count.

- **BL_Count** - count devices, analog and logic channels, or ranges

BL_Count is also used to report the number of *analog channels*, *logic channels* and *analog input ranges* (on each analog channel). Which entity it reports depends on

its argument **and** *which device, channel or source is selected a the time it is called*. The *device, channel* and *source* are selected using BL_Select:

- **BL_Select** - select a device, channel or signal source

The first BL_Select argument specifies which type of entity to select. When the device is **selected for the first time**, its *trace mode must also be selected*:

- **BL_Mode** - select a trace mode

It must be called after selecting the device but **before selecting the channel**. This is important because the number of channels available may be fewer than the physical number the device supports in some modes. For example, to select the BNC source on channel 1 on device 0:

```
BL_Select(BL_SELECT_DEVICE,0);
BL_Mode(BL_MODE_FAST);
BL_Select(BL_SELECT_CHANNEL,1);
BL_Select(BL_SELECT_SOURCE,BL_SOURCE_BNC);
```

Once the device and mode are selected, each channel may be selected in turn and configured to choose a source, input offset, voltage range, signal coupling and whether to enable it for capture:

- **BL_Range** - select the channel range
- **BL_Offset** - assign channel offset
- **BL_Coupling** - select the channel source coupling
- **BL_Enable** - change channel enable status

For each channel enabled for capture, this process is repeated as required. When the channels are all configured, the trace may be programmed.

(3) Trace Programming and Capture

After the device, mode and channels are configured, *trace settings are programmed*. First the sample rate and capture size (specified in samples) are assigned:

- **BL_Rate** - assign the sample rate (Hz)
- **BL_Size** - assign the capture size (samples)

This must be done first to establish the core trace settings. Choose values to ensure the duration required will be captured given the selected sample rate. Some modes use finite (device) buffers so your choice may be constrained. An alternative is to assign the duration directly:

- **BL_Time** - assign the capture duration (seconds)

In this case, the capture size and sample rate may be adjusted by the library automatically. Next the trigger, if required, is established:

- **BL_Trigger** - set up the trigger

This function accepts two arguments specifying the trigger level (which is applied to the currently selected channel) and the type of trigger. If signals are to be captured before the trigger, or a delay is required after the trigger, these parameters are specified next using two functions:

- **BL_Intro** - assign the pre-trigger size (intro region)
- **BL_Delay** - assign post-trigger delay (delay before capture)

Both functions are optional (not required when tracing untriggered). At this point *the device is ready to capture waveforms and logic data*. All the preceding steps need not be repeated if the parameters for a series of captures remain unchanged. To commence the trace and capture signals call:

- **BL_Trace** - initiate capture

This function is the one that actually *talks to BitScope and captures waveforms*. This function *may take an arbitrarily long time to complete*. Indeed it *may never complete*, so to avoid locking your program, it may be called *asynchronously* or it may be called with a *specified timeout*. When called *asynchronously* BL_Trace *always returns immediately*, even if the trace has not yet completed. In this case call:

- **BL_State** - return capture engine state

periodically after BL_Trace to monitor progress of the trace. BL_State returns a token reporting trace in progress, trace complete, timeout or an error code. When an asynchronous trace is in progress it may be manually stopped with:

- **BL_Halt** - all any pending or prevailing device activity

An alternative *is to call BL_Trace synchronously* with a *specified timeout*. In this case BL_Trace is guaranteed to return within the time specified but the trace may or may not have completed in that time; it returns TRUE if it has, FALSE otherwise.

(4) Acquiring Data from the Device

Once the trace has completed, the data may be acquired:

- **BL_Acquire** - acquire data from the device.

BL_Acquire uploads data from the device one channel at a time. Before it is called each time, select the channel (and optionally the device) to be acquired with BL_Select. If acquiring from other than the first sample, the starting index may be specified:

- **BL_Index** - assign the buffer offset (for acquisition)

If BL_Index is used it must be called *before BL_Trace*. In any case, the return value of BL_Acquire specifies how many sample are actually acquired. The return value will not be greater than the number requested but it may be fewer:

1. If the number of samples captured is fewer than the number requested,
2. If the trace was terminated early, a timeout or error occured, or
3. A programming error (such as forgetting to enable the channel).

Typically one executes BL_Trace and cycles through a sequence of BL_Acquire, one for each channel (on each device), before executing the next BL_Trace.

(5) Closing Devices and the Library

When you're finished with the library, call **BL_Close**. This closes all open devices (it's not possible to close only one). If you wish to close one of several devices, close them all and reopen those you wish to continue using.

Probe Files and Link Specifications

The **Probe File** is a *configuration file* on the host computer. It tells the Library *where it can find the BitScopes* and *how to connect with them*. The Probe File is a **text file** containing a list of **Link Specifications** defining connection protocols used via RS-232, USB or IP networks. It is also used by BitScope applications and is automatically created by these applications if it does not already exist. The Probe file may also be created manually with a text editor but it if does not exist when the library is used, built-in defaults are used instead.

Probe File Locations

There are two probe files on the host, one **global** and the other **local**. Both global and local Probe Files are named **BitScope.prb** (or *bitscope.prb*). The **global probe file** establishes system defaults and should not be modified. The **local probe file** is automatically created (from the global file, if the local file does not exist) and it may be manually or automatically modified, as required.

Windows Probe File Locations

On a Windows PC, the **local probe file** is located at:

```
C:\Users\<user>\AppData\Local\BitScope\BitScope.prb
```

or on older version of Windows:

```
C:\Documents and Settings\<user>\Local Settings\Application Data\BitScope\BitS
cope.prb
```

where is your logged in user name on your PC. The global probe file is located at

```
<BITSCOPE>\BitScope.prb
```

where **< BITSCOPE >** is the location chosen when you installed the Library. The default (and recommended) location is:

```
C:\Program Files\BitScope\BitScope.prb
```

Linux, Mac OS X and Raspberry Pi Probe File Locations

On a Linux or Mac OS X based system, the local probe file is located as:

```
~/.config/bitscope/bitscope.prb
```

which conforms to the XDG Specification. The global probe file is located at:

```
/etc/bitscope/bitscope.prb
```

The global probe file will not exist if you have not installed the packaged version of the library but defaults apply in this case. On Linux and Mac systems, the probe file can also be located at:

```
~/.bitscope/config/bitscope.prb
```

but this location is depreciated and used for backward compatibility only.

Link Specification Syntax

Each line in the probe file is either *blank*, a *comment* or a **link specification**. A *link specification* defines a connection with a BitScope and has the same syntax when used directly with BL_Open as it does when it appears in a probe file. The following is an example probe file for a Linux system:

```
# PROBE FILE SYNTAX
#
# Whitespace, blank lines or lines including '#' are ignored.
```

```
#
# The remaining lines each define a single "link". Each link
# specifies a connection method and identifies a device.
#
# The complete syntax of a LINK is:
#
#      LINK => SERIAL|NETWORK|NONE
#
#   NETWORK => UDP:ADDRESS(:PORT)?(:ID)?(:BC)?
#    SERIAL => (TTY|USB):DEVICE(:BAUD)?(:BC)?
#      NONE => NIL:BC(:PRODUCT)?(:VENDOR)?
#
#       TTY => serial connection method
#       USB => USB/Serial connection method
#       UDP => User Datagram Protcol (network) method
#       NIL => No connection method (simulate device)
#
#    DEVICE => serial or USB device (eg, /dev/ttyUSB0)
#   ADDRESS => IP address or host name or "MULTICAST"
#   PRODUCT => Device identifier (up to 8 alpha numeric)
#    VENDOR => Device vendor identifier (8 alpha-mueric)
#      PORT => IP port number (4 hex digits)
#      BAUD => Serial baud rate (normally optional)
#        ID => BitScope LIA ID (4 hex digits)
#        BC => byte-code revision ID (eg, BC000301)
#
# For example, to connect to a serial BitScope:

TTY:/dev/ttyS2

# a USB BitScope:

USB:/dev/ttyUSB1

# a serial BitScope, but only if it's a BS311

TTY:/dev/ttyS2:BS031100

# a network BitScope via multicast

UDP:MULTICAST

# or (if there are more than one BitScope on the local net)
# and you want to use port numbers to differentiate them

UDP:MULTICAST:4321
UDP:MULTICAST:4322

# or using different IP addresses with unicast:

UDP:192.168.1.2
UDP:192.168.1.5
```

```
# or if your BitScope has the hostname LABSCOPE

UDP:LABSCOPE

# or to connect with our demo BitScope on the Internet

UDP:sydney.bitscope.com

# you get the picture.
```

Windows systems are the same except the names used for serial and USB ports are COM1, COM2 … COM instead of /dev/ttyS0 or /dev/ttyUSB0 etc.

Probe File Defaults

If neither the local or global both probe files exist, built-in defaults are used. The library attempts to locate available BitScopes automatically in this case using an internal algorithm. Alternatively an explicit link specification can be provide to BL_Open when called (in which case the probe file is ignored). Defaults are also provided by the global probe file, if it exists. On Windows typical defaults are:

```
UDP:MULTICAST
USB:COM3
USB:COM4
USB:COM5
USB:COM6
USB:COM7
USB:COM8
USB:COM9
UDP:SYDNEY
TTY:COM1
TTY:COM2
NIL:BS032500
```

and on Linux based systems they are:

```
USB:/dev/ttyUSB0
UDP:MULTICAST
USB:/dev/ttyUSB1
USB:/dev/ttyUSB2
TTY:/dev/ttyS0
TTY:/dev/ttyS1
UDP:SYDNEY
NIL:BS032500
```

When using BL_Open with a probe file, links are probed in the order they appear in the local file, then the global file and finally the defaults are used. This is why the library can “magically open a BitScope”, only to find it’s not the BitScope you

intended. Use BL_Open with an explicit link specification if you do not want this to happen.

Device Simulations

There two special types of device supported by the library: **NIL** and **SIM**.

A **NIL device** is a minimal *device simulation* sufficient to open and configure but not synthesize capture data. It is useful when developing software with library for use with BitScope devices you may not have available because it allows you to use the Library API to test that your code works with any supported model. A NIL device link specification looks like:

```
NIL:<DEVICE>
```

where **< DEVICE >** is the device type identifier.

There are currently 22 device simulations supported by the library:

```
BS001003 BS012000 BS032600 BS044501 BS001004 BS001001 BS001000
BS010000 BS032500 BS032000 BS031100 BS031000 BC000301 BC000300
BS044500 BS044202 BS044201 BC000441 BC000440 BS005000 BC000220
BC000120
```

A **SIM device** is a more *sophisticated device simulation* that simulates the operation of some model BitScopes. In addition to a NIL device is synthesizes data as if captured by a real BitScope. It is available in some developer editions of the library only.

Diagnostics

You may be interested to know what the library is doing as your program runs. The **BL_Log** function is available to provide access to this information. It tells you what the library does and how long execution takes in response to your API function calls. Example output of BL_Log looks like this:

```
Data acquisition complete. Dump Log...
C    0 SEND open 1
R  243 CALL status device 1
C    0 SEND select -1
C    0 SEND do set mode 0
C    0 SEND do set channel 0
```

```
C    0 SEND do set factor 1.00
R    0 CALL select 0
C    0 SEND do set rate 0.00
S   13 EXEC update rate 20000000.00
C    0 SEND do set count 0
S    0 EXEC update count 12288
C    0 SEND do set mode 0
C    0 SEND do set channel 0
C    0 SEND do set factor 1.00
R    0 CALL status analog 2
R    0 CALL status logic 8
R    1 CALL select 0
C    0 SEND do set mode 0
C    0 SEND do set channel 0
C    0 SEND do set factor 1.00
R    0 CALL status analog 2
R    0 CALL status logic 8
R    0 CALL do mod channel 0
R    0 CALL do mod mode 0
R    0 CALL do mod intro 0.00
R    0 CALL do mod delay 0.00
R    1 CALL do mod rate 1000000.00
R    0 CALL do mod count 4
R    0 CALL do mod channel 0
C    0 SEND trigger
R    0 CALL do mod level 0.00
R    1 CALL do mod source 1
R    0 CALL status range 5
C    0 SEND do set range 5
R    0 CALL do get scale 11.00
R    0 CALL do mod offset 0.00
C    0 SEND do set enable true
R    0 CALL do get time 0.00
R    0 CALL do get rate 1000000.00
R    0 CALL do get count 4
R   11 CALL trace forced 0.00
R    0 CALL do mod channel 0
C    0 SEND do mod size 4
R    0 CALL dump
R    3 CALL do get size 4
```

A detailed description of these diagnostics is beyond the scope of this manual but several general points to note are the:

1. first columm reports command (C), reply (R) or state (S) execution
2. second reports how many milliseconds between each command
3. third reports RPC (CALL), messaging (SEND) or execution (EXEC)
4. remaining lines indicate what actions are taken and vary per log entry

The BL_Log function may not be enabled in all editions of the library. For example, in optimized production releases it is usually disabled for performance reasons. By contrast, in developer editions more detailed diagnostics may available for

specialized purposes. However using these generally requires a more advanced knowledge of the BitScope Capture Engine and its Virtual Machine Architecture. More information is available for OEM developers upon request.

Error Recovery

In the real world, errors happen.

For example, the power can fail or network connections can drop out.

The Library handles many errors automatically, for example by reinitiating a device request. However, some errors are too significant to recover this way in which case the client program must take additional steps to recover normal operation.

Errors are generally detected one of three ways:

1. BL_Trace returns FALSE when *executing synchronously*, or
2. BL_State reports **BL_STATE_ERROR** when *executing asynchronously*.
3. BL_Acquire returns fewer samples than requested when called.

Take care with the third mechanism, BL_Acquire may return fewer samples than requested if you've made a set up programming error too!

In general, error recovery simply requires that you reprogram the BitScope the same way you did when you first opened it. That is, if an error is detected, re-execute your setup code before continuing as normal. If the error persists, one of the three error reporting mechanisms above will persist in which case you may wish to use **BL_Error** to determine the nature of the error (e.g. link down, power failure etc) and report it to the user to take appropriate action.

Most importantly, error detection and recovery is easy:

1. Minor errors (e.g. dropped packets) are handled automatically,
2. Major errors (e.g power failure) are recovered automatically (when restored)
3. No client side error callbacks or exception handlers are required,
4. BitScope is idempotent so recovery coding is simple.

The last point simply means you can program BitScope via the library the same way repeatedly and it will always do the same thing in response.

Code Examples

Programming with library is very easy.

Here is a C example that captures a single analog channel:

```
/* capture.c -- BitLib 2.0 Analog Capture (Single Channel) (ANSI C) */

#include <stdio.h>
#include <limits.h>
#include <bitlib.h> /* required */

#define MY_DEVICES 1 /* open one device only */
#define MY_PROBE_FILE "" /* default probe file if unspecified */

#define MY_DEVICE 0
#define MY_CHANNEL 0
#define MY_MODE BL_MODE_FAST
#define MY_RATE 1000000 /* capture sample rate */
#define MY_SIZE 4 /* number of samples to capture */

int main(int argc, char *argv[]) {
   /*
    * Open and select the first channel on the first device.
    */
    printf("\nStarting: Attempting to open %d device%s...\n",MY_DEVICES,MY_DEV
ICES!=1?"s":"");
    if ( ! BL_Open(MY_PROBE_FILE,MY_DEVICE) ) {
        printf("Failed to find a devices.\n");
        goto exit;
    }
    if ( BL_Select(BL_SELECT_DEVICE,MY_DEVICE) != MY_DEVICE ) {
        printf("Failed to select device %d.\n",MY_DEVICE);
        goto exit;
    }
    if ( BL_Select(BL_SELECT_CHANNEL,MY_CHANNEL) != MY_CHANNEL ) {
        printf("Failed to select channel %d.\n",MY_CHANNEL);
        goto exit;
    }
   /*
    * Prepare to capture one channel...
    */
    if ( BL_Mode(MY_MODE) != MY_MODE ) {
        printf("Failed to select mode %d.\n",MY_MODE);
        goto exit;
    }
    BL_Intro(BL_ZERO); /* optional, default BL_ZERO */
    BL_Delay(BL_ZERO); /* optional, default BL_ZERO */
    BL_Rate(MY_RATE); /* optional, default BL_MAX_RATE */
    BL_Size(MY_SIZE); /* optional, default BL_MAX_SIZE */
    BL_Select(BL_SELECT_CHANNEL,MY_CHANNEL); /* choose the channel */
    BL_Trigger(BL_ZERO,BL_TRIG_RISE); /* optional when untriggered */
    BL_Select(BL_SELECT_SOURCE,BL_SOURCE_POD); /* use the POD input */
    BL_Range(BL_Count(BL_COUNT_RANGE)); /* maximum range */
    BL_Offset(BL_ZERO); /* optional, default 0 */
    BL_Enable(TRUE); /* at least one channel must be initialised */
```

```
   /*
    * Capture and acquire the data...
    */
    printf("   Trace: %d samples @ %.0fHz = %fs\n",BL_Size(BL_ASK),BL_Rate(BL_
ASK), BL_Time(BL_ASK));
    if ( BL_Trace(BL_TRACE_FORCED,BL_SYNCHRONOUS) ) { /* capture data (without
a trigger) */
        int i, n = MY_SIZE; double d[n]; /* let's get 5 samples */
        BL_Select(BL_SELECT_CHANNEL,MY_CHANNEL); /* optional if only one chann
el */
        if ( BL_Acquire(n, d)  == n ) { /* acquire (i.e. dump) the capture dat
a */
            printf("Acquired:");
            for (i = 0; i < n; i++)
                printf(" %f", d[i]);
            printf("\n\n");
        }
    }
    printf("Data acquisition complete. Dump Log...\n");
    printf("%s\n",BL_Log());
  exit:
    BL_Close(); /* call this to release library resources */
    return 0;
}
```

and this is a Python example that dumps a report about a device:

```
'''report.py -- BitLib 2.0 Capture Device Report Generator (Python)'''

from bitlib import *

MY_DEVICE = 0 # one open device only
MY_CHANNEL = 0 # channel to capture and display
MY_PROBE_FILE = "" # default probe file if unspecified
MY_MODE = BL_MODE_FAST # preferred trace mode
MY_RATE = 1000000 # default sample rate we'll use for capture.
MY_SIZE = 5 # number of samples we'll capture (simply a connectivity test)
TRUE = 1

MODES = ("FAST","DUAL","MIXED","LOGIC","STREAM")
SOURCES = ("POD","BNC","X10","X20","X50","ALT","GND")

def main(argv=None):
    #
    # Open the first device found (only)
    #
    print "\nStarting: Attempting to open one device..."
    if BL_Open(MY_PROBE_FILE,1):
        #
        # Open succeeded (report versions).
        #
        print " Library: %s (%s)" % (
```

```
            BL_Version(BL_VERSION_LIBRARY),
            BL_Version(BL_VERSION_BINDING))
        #
        # Select this device (optional, it's already selected).
        #
        BL_Select(BL_SELECT_DEVICE,MY_DEVICE)
        #
        # Report the link, device and channel information.
        #
        print "    Link: %s" % BL_Name(0)
        print "BitScope: %s (%s)" % (BL_Version(BL_VERSION_DEVICE),BL_ID())
        print "Channels: %d (%d analog + %d logic)" % (
            BL_Count(BL_COUNT_ANALOG)+BL_Count(BL_COUNT_LOGIC),
            BL_Count(BL_COUNT_ANALOG),BL_Count(BL_COUNT_LOGIC))
        #
        # Determine which modes the device supports.
        #
        print "   Modes:" + "".join(["%s" % (
            (" " + MODES[i]) if i == BL_Mode(i) else "") for i in range(len(MO
DES))])
        #
        # Report canonic capture specification in LOGIC (if supported) or FAST
mode (otherwise.
        #
        BL_Mode(BL_MODE_LOGIC) == BL_MODE_LOGIC or BL_Mode(BL_MODE_FAST)
        print " Capture: %d @ %.0fHz = %fs (%s)" % (
            BL_Size(),BL_Rate(),
            BL_Time(),MODES[BL_Mode[]])
        #
        # Report the maximum offset range (if the device supports offsets).
        #
        BL_Range(BL_Count(BL_COUNT_RANGE));
        if BL_Offset(-1000) != BL_Offset(1000):
            print "  Offset: %+.4gV to %+.4gV" % (
                BL_Offset(1000), BL_Offset(-1000))
        #
        # Report the input source provided by the device and their respective
ranges.
        #
        for i in range(len(SOURCES)):
            if i == BL_Select(2,i):
                print "     %s: " % SOURCES[i] + " ".join(["%5.2fV" % BL_Range
(n) for n in range(BL_Count(3)-1,-1,-1)])
        #
        # Set up to capture MY_SIZE samples at MY_RATE from CH-A via the POD i
nput using the highest range.
        #
        BL_Mode(MY_MODE) # prefered trace mode
        BL_Intro(BL_ZERO); # optional, default BL_ZERO
        BL_Delay(BL_ZERO); # optional, default BL_ZERO
        BL_Rate(MY_RATE); # optional, default BL_MAX_RATE
        BL_Size(MY_SIZE); # optional default BL_MAX_SIZE
        BL_Select(BL_SELECT_CHANNEL,MY_CHANNEL); # choose the channel
```

```
        BL_Trigger(BL_ZERO,BL_TRIG_RISE); # optional when untriggered */
        BL_Select(BL_SELECT_SOURCE,BL_SOURCE_POD); # use the POD input */
        BL_Range(BL_Count(BL_COUNT_RANGE)); # maximum range
        BL_Offset(BL_ZERO); # optional, default 0
        BL_Enable(TRUE); # at least one channel must be initialised
        #
        # Perform an (untriggered) trace (this is the actual data capture).
        #
        BL_Trace()
        #
        # Acquire (i.e. upload) the captured data (which may be less than MY_S
IZE!).
        #
        DATA = BL_Acquire()
        print " Data(%d): " % MY_SIZE + ", ".join(["%f" % DATA[n] for n in ran
ge(len(DATA))])
        print "Complete: trace and acquisition complete. Dump the log...\n"
        print "%s" % BL_Log()
        #
        # Close the library to release resources (we're done).
        #
        print "Finished: close the library to release resources."
        BL_Close()
    else:
        print "  FAILED: device not found (check your probe file)."

if __name__ == "__main__":
    import sys
    sys.exit(main())
```

There are other examples included with the packaged library for each platform.

Troubleshooting

The Library can’t find my BitScope!

Is the link specification you have given to BL_Open correct?

Are you using a probe file? Does it exist? Does it have the correct link specification?

Does the serial or USB port for the BitScope exist on your system when the BitScope is connected? Does your login name have permission to access it? If using a Network BitScope is the network address routable (does it “ping“) from your system?

Is your BitScope powered on and does it have a good quality power supply? Are the USB or Network and power cables you are using okay? Is your USB port working?

Have you tried a different PC? Does other BitScope software connect okay?

Drop us a line any time at support@bitscope.com if you still have problems!

Which BitScope am I really connected to?

Sometimes when opening a BitScope you get what looks like the wrong one. It probably is. This is likely due to the use of probe file defaults. Call BL_Open with an explicit link specification for the BitScope you want to avoid this possibility.

Why does it take so long to connect?

If the probe file specifies a lot of links which are invalid or the devices they refer to are not available, BL_Open may take a long time to work through them all (in the order they appear in the probe file) until it reaches the one that ultimate connects. Note: using probe file that has *no correct value* can result in the same thing due to defaults.

If this is the case, simply comment out (preceed with #) each link you don't need (or delete them entirely) or use an explicit link description instead of the probe file when calling BL_Open.

What parameter values does the library actually use?

When programming a trace the sample rate, trace size and other parameters such as pre-trigger intro *may be constrained by what the device is capable of delivering*.

For all such parameters you must *check the return value of the parameter when you assign a new value* because *the return value is what the library will actually use*.

Usually the requested and returned value will be the same but if you run up against the device constraints they may be different. Further, there are situations where changing one parameter (e.g. channel enable or trace mode) can alter the constraints that apply to others (e.g. sample rate or trace size).

The **general rule** is; if device setup parameters are changed some trace parameters may need to be reprogrammed. Follow the recommended programming sequence to avoid any confusion.

Can I use the library in a multi-threaded environment?

Yes, but you need to know what you're doing!

Access to the library functions is thread-safe but you need to be aware that many functions rely on the device, channel or input source being preselected by BL_Select and this can lead to a race condition in your code if you're not careful.

To the extent we recommend you use multi-threaded programming we advise that all access to the library be confined to one thread with the call to the "business end" of the library (BL_Trace and BL_State) being available for use in another thread.

It's safe to call *BL_Trace synchronously in another thread* and *use BL_State in the main thread* (or any other multi-threaded synchronization mechanism) to tell the main thread when the trace has completed.

However in general you don't need to do this; instead, call BL_Trace asynchronously from the main thread and poll (e.g. in a timer handler) BL_State to check progress.

Help! I don't understand how BL_xxxx works?

Email us any time at support@bitscope.com for assistance.

This is not working, I've found a Bug!

You may well have done. Please email full details to support@bitscope.com!

Library Reference

BL_Acquire - dump data from the device

This is the primary *data acquisition* function.

```
int BL_Acquire(int ASize, double * AData); /* C/C++ Prototype */
function BL_Acquire(ASize : Integer; AData : PDouble) : Integer; { Pascal Prot
otype }
```

Call **BL_Acquire** to dump data from a channel to the host (Acquire) *after the data has been captured by the device* (BL_Trace). Data is dumped from the selected channel on the selected device (BL_Select). To dump data from multiple channels, possibly on multiple devices, loop through and select each channel on each device

and call BL_Acquire at each iteration.

Acquired data is written to the array in memory pointed to by **AData**. The number of samples acquired is **ASize**. The size of the array provided *must be at least* **ASize**. The dump commences at the first sample in the channel buffer unless a different (positive) starting index is specified (BL_Index). The sum of the dump size and start index must be less than or equal to the total trace size (BL_Size).

The return value is the number of samples acquired. It equals the number of samples requested (**ASize**) unless:

1. more samples than were captured were requested, or
2. the trace was terminated early (BL_Halt or *timeout*), or
3. an error occured when communicating with the device.

If 3 occurs you may need to take steps to recover the error. Like BL_Trace (when called *synchronously*) this function may take a while to complete. How long it takes depends on the speed of the link between the host and the device.

BL_Close - close all open devices

Close all open device connections and free all resources.

```
void BL_Close (void); /* C/C++ Prototype */
procedure BL_Close; { Pascal Prototype }
```

If you wish to close only one of several devices, close them all and reopen those you wish to continue using. In most situations, retaining an open device for later us is not expensive.

BL_Count - count devices, channels, or ranges

Count available devices, channels and ranges.

```
int BL_Count(int AType); /* C/C++ Prototype */
function BL_Count(AType : Integer) : Integer; { Pascal Prototype }
```

When one or more devices have been opened (BL_Open) one usually needs to know how many opened successfully. For each device one may also need to know many channels and how many analog ranges are available.

BL_Count returns a count based on its **AType** argument:

- **BL_COUNT_DEVICE** - successfully opened devices
- **BL_COUNT_ANALOG** - analog channels on the selected device
- **BL_COUNT_LOGIC** - logic channels on the selected device

- **BL_COUNT_RANGE** - number of analog ranges on the selected channel

When counting channels or ranges a device must first be selected (BL_Select).

BL_Coupling - select the channel source coupling

Choose DC, AC or RF coupling on the selected channel.

```
int BL_Coupling(int ACoupling) : Integer; /* C/C++ Prototype */
function BL_Coupling(ACoupling : Integer = BL_ASK) : Integer; { Pascal Prototy
pe }
```

Many BitScope models offer software selectable DC or AC coupling for their BNC terminated inputs. Some others offer an RF coupling option.

BL_Coupling selects the coupling used via **ACoupling** argument:

- **BL_COUPLING_DC** - direct connection for DC signals (the default)
- **BL_COUPLING_AC** - AC connection (eliminates DC bias)
- **BL_COUPLING_RF** - RF connection for very high frequencies

BL_Coupling **returns the newly selected coupling** *if successful* or the prevailing coupling otherwise, so check the return value to determine if the requested coupling is available. Use **BL_ASK** to enquire the prevailing coupling without changing it.

BL_Delay - assign post-trigger delay

Apply an optional *post-trigger delay* the trace.

```
double BL_Delay(double ADelayTime); /* C/C++ Prototype */
function BL_Delay(ADelayTime : Double) : Double; { Pascal Prototype }
```

All BitScopes offer a facility to insert a precise delay after the trigger before waveform capture proceeds (in most trace modes).

BL_Delay establishes this post-trigger delay (**ADelayTime**, in seconds). It returns the same value *if the device can support it*, otherwise the nearest value is returned.

Always check the return value because *it is the value that is actually used*.

To specify that no post trigger delay should be used, specify **BL_ZERO**.

To specify a *pre-trigger capture duration* use BL_Intro.

BL_Enable - change channel enable status

Enable or disable a channel from participating in a trace.

```
bool BL_Enable(book AEnable); /* C/C++ Prototype */
function BL_Enable(AEnable : Boolean) : Boolean; { Pascal Prototype }
```

Use **BL_Enable** to enable or disable the selected channel on the selected device from particpating in a trace. It's generally good idea to disable channels you are not using because it frees up device buffers and other resources for the other channels.

The channel enable status should be established at the device programming and setup step, i.e. **before** assigning any of the trace parameters.

If you change a channel's enable state, it is recommended that you also reassign trace parameters such sample rate and size even if the intended values remain unchanged because constraints may apply.

BL_Error - return most recent error (if any)

Report the most recent error code (if any).

```
int BL_Error (void); /* C/C++ Prototype */
function BL_Error : Integer; { Pascal Prototype }
```

In normal operation errors can occur. In general you don't need to know what caused the error beyond whether it's SNAFU or FUBAR and this level of detail is reported by BL_State.

However, when encountering a FUBAR class error you may need to report to the user an error code which will tell them what to fix to recover from it (e.g. restore device power). In this case the return value of **BL_Error** provides this information.

BL_Halt - all any pending or prevailing device activity

Terminate all activity on the selected device.

```
bool BL_Halt(void); /* C/C++ Prototype */
function BL_Halt : Boolean; { Pascal Prototype }
```

Call **BL_Halt** to terminate any pending or prevailing device activity. Returns TRUE if successful, FALSE otherwise. BL_Halt returns FALSE only if an error occured when talking to the device; in this case you may need to recover the error. BL_Halt does not need to be called if the capture engine state is not BL_STATE_ACTIVE but it is safe to call it at any time.

BL_ID - return the selected device ID

Read the unique ID of the selected device.

```
char * BL_ID(void); /* C/C++ Prototype */
function BL_ID : PAnsiChar; { Pascal Prototype }
```

Every BitScope device has a unique ID. **BL_ID** returns this ID.

This information is useful to record with captured data so as to keep a record of which device produced that data. It may also be used in a link specification to ensure one connects to a uniquely identified device.

BL_Index - assign the buffer offset (for dumps)

Assign an offset in the capture buffer from which to acquire data.

```
bool BL_Index(AAddr : Integer); /* C/C++ Prototype */
function BL_Index(AAddr : Integer) : Boolean; { Pascal Prototype }
```

When dumping data from the device buffer using BL_Acquire is may be desirable to commence the dump from a sample other than the first. Use **BL_Index** to specify this offset.

BL_Initialize - initialize the library (optional)

Initialize the library infrastructure for use.

```
void BL_Initialize[]; /* C/C++ Prototype */
procedure BL_Initialize; { Pascal Prototype }
```

This function is optional; it is called implicitly when the first device is opened. It may be called explicit before any devices are opened if preferred.

BL_Intro - assign the pre-trigger size (intro

region)

Apply an optional *pre-trigger capture duration* the trace.

```
double function BL_Intro(double APreTrigger); /* C/C++ Prototype */
function BL_Intro(APreTrigger : Double) : Double; { Pascal Prototype }
```

BitScopes offer a facility to capture *before the trigger* (in most trace modes). **BL_Intro** establishes this pre-trigger capture duration (**APreTrigger**, in seconds). It returns the same value *if the device can support it*, otherwise the nearest value is returned.

Always check the return value because *it is the value that is actually used*.

To specify that no pre-trigger capture should be used, specify **BL_ZERO**.

To specify a *post-trigger delay* use BL_Intro.

BL_Log - dump the pending log

Dump the library diagnostic log.

```
char * BL_Log(Void); /* C/C++ Prototype */
function BL_Log : PAnsiChar; { Pascal Prototype }
```

In developer editions of the library the internal operation of the library is logged. **BL_Log** returns a dump of this information to assist with program development using the library. See Diagnostics for more information.

BL_Mode - select and trace mode

Choose the **trace mode** to be used for the next capture.

```
int function BL_Mode(int AMode); /* C/C++ Prototype */
function BL_Mode(AMode : Integer = BL_MODE_FAST) : Integer; { Pascal Prototype
}
```

All BitScope models offer a range of **trace modes**. Trace modes define how the device captures the input signals:

- **BL_MODE_FAST** - analog capture at the fastest rates available
- **BL_MODE_DUAL** - dual channel sample synchronous analog capture
- **BL_MODE_MIXED** - mixed analog + logic signal capture
- **BL_MODE_LOGIC** - logic only capture mode

- **BL_MODE_STREAM** - streaming mixed signal capture

All models offer **BL_MODE_FAST** and **BL_MODE_MIXED**. The other modes are available on some models and not others; it depends on the hardware design of each model. See the hardware guide for your device.

BL_MODE_FAST is recommended for analog waveform capture only and **BL_MODE_MIXED** when also capturing logic data. **BL_MODE_DUAL** is recommended for single A/D devices when sample synchronous analog waveform capture is required. **BL_MODE_LOGIC** is recommend when capturing logic only and **BL_MODE_STREAM** is used for continuous waveform capture.

It is important to establish the trace mode at the device programming step because the choice of mode changes the device constraints that may apply.

BL_Name - return the device link name

Read the canonic link specification of the selected device.

```
char * BL_Name(char * AStr); /* C/C++ Prototype */
function BL_Name(AStr : PAnsiChar) : PAnsiChar; { Pascal Prototype }
```

The host connects to each BitScope device accorind to a link specification.

Call BL_Name to return the canonic link specification.

BL_Offset - assign channel offset

Apply a voltage offset to the selected channel

```
double BL_Offset(double AValue); /* C/C++ Prototype */
function BL_Offset(AValue : Double) : Double; { Pascal Prototype }
```

Most BitScope models allow an input offset to be applied to the signal captured on analog channels. **BL_Offset** assigns this offset (**AValue**, in volts) to the selected channel on the selected device. It returns the same value *if the device can support it*, otherwise the nearest value is returned.

Always check the return value because *it is the value that is actually used*.

To specify that no offset should be applied, specify **BL_ZERO**.

BL_Open - open one or more devices

Open one or more device according supplied link specifications.

```
int BL_Open(char * AProbeStr; int ACount); /* C/C++ Prototype */
function BL_Open(AProbeStr : PAnsiChar; ACount : Integer = 1) : Integer; { Pas
cal Prototype }
```

Call **BL_Open** to open a device specified by **AProbeStr**. If the second argument (**ACount**) is provided it tells BL_Open how many devices to open (defaults to 1).

The first argument can be:

1. omitted, in which case the default probe file is used,
2. the canonic name of the probe file (in a standard location)
3. the full name of the probe file (in any location)
4. a literal link specification or
5. a colon separated list of link specifications.

BL_Open attempts to open the devices using these techniques in the order listed.

It returns the number of devices sucessfully opened.

Each opened device may then be selected with an index from 0 to N-1 where N is the value returned by BL_Open or BL_Count (BL_COUNT_DEVICES).

BL_Range - select the channel range

Select the analog input range the selected source

```
double BL_Range(int AIndex); /* C/C++ Prototype */
function BL_Range(AIndex : Integer) : Double; { Pascal Prototype }
```

All BitScope models provide a number of analog input ranges which scale the signal prior to capture and conversion to the digital domain.

Use **BL_Range** to assign the desired range (**AIndex**) to the selected source. The source index used must be between 0 and BL_Count (BL_COUNT_RANGE)-1;

BL_Range returns the full voltage span of the selected range.

BL_Rate - assign the sample rate

Assign the capture sample rate for the next trace.

```
double function BL_Rate(double ASampleRate); /* C/C++ Prototype */
function BL_Rate(ASampleRate : Double = BL_ASK) : Double; { Pascal Prototype }
```

The capture sample rate is a fundamentally important trace parameter.

It *must always be specified* and it *should be the first parameter assigned* when preparing a new trace (if different from the previous trace).

BL_Rate establishes the *preferred sample rate* (**ASampleRate**, in Hz). It returns the same value *if the device can support it*, otherwise the nearest value is returned.

To choose the highest sample rate supported by the selected device in the current mode, specify **BL_MAX_RATE**. To enquire what the prevailing sample rate is without changing it, specify **BL_ASK**. Regardless of how BL_Rate is called **always check the return value** because *it is the value that is actually used*.

BL_Select - select a device, channel or source

Select the a device, channel or source for subsequent use.

```
int BL_Select(int AType, int AIndex); /* C/C++ Prototype */
function BL_Select(AType : Integer; AIndex : Integer = BL_ASK ) : Integer; { P
ascal Prototype }
```

The library makes available multiple devices, channels and input sources. Most API functions operate on one of these at a time. Use **BL_Select** to choose which one.

BL_Selects chooses which entity type to select via its first (**AType**) argument:

- **BL_SELECT_DEVICE** - select a device
- **BL_SELECT_CHANNEL** - select a channel on the (previously) selected device
- **BL_SELECT_SOURCE** - select an input source of the selected channel

The entity of that type is selected via the second (**AIndex**) argument.

BL_Select **returns the index of the selected entity** *if the selection was successful*. The prevailing selection is returned unchanged otherwise, so check the return value is the same as the AIndex argument to confirm the new selection was successful. Use **BL_ASK** to enquire the prevailing selection without changing it.

Devices enumerate from **0 to BL_Count (BL_COUNT_DEVICES)-1**.

Analog channels enumerate from **0 to 3** and **logic channels** from **4 to 11**.

Sources are selected from a **pre-defined set of options**:

- **BL_SOURCE_POD** - analog or logic channel POD input
- **BL_SOURCE_BNC** - analog channel BNC input (if available)
- **BL_SOURCE_X10** - analog input prescaled by 10
- **BL_SOURCE_X20** - analog input prescaled by 20

- **BL_SOURCE_X50** - analog input prescaled by 50
- **BL_SOURCE_ALT** - alternate input (data acquisition)
- **BL_SOURCE_GND** - ground reference input

Not all device types support all sources. Check the return index when attempting to select one to see if it's available.

BL_Size - assign the capture size (samples)

Assign the capture size for the next trace.

```
int BL_Size(int ASize); /* C/C++ Prototype */
function BL_Size(ASize : Integer = BL_ASK) : Integer; { Pascal Prototype }
```

The capture size is a fundamentally important trace parameter. It *must always be specified* and it *should be the second parameter assigned* (after BL_Rate) when preparing a new trace (if different from the previous trace).

BL_Size establishes the *preferred trace size* (**ASize**, in samples). It returns the same value *if the device can support it*, otherwise the nearest value is returned. To choose the largest size supported by the selected device in the current mode, specify **BL_MAX_SIZE**. To enquire what the prevailing size is without changing it, specify **BL_ASK**. Regardless of how BL_Size is called **always check the return value** because *it is the value that is actually used*.

BL_State - return capture engine state

Report capture engine state while tracing.

```
int BL_State (void); /* C/C++ Prototype */
function BL_State : Integer; { Pascal Prototype }
```

When a trace is initiated asynchronously the state of the capture engine is needed to be able to know when the trace has completed and acquisition can proceed. Use **BL_State** to obtain this information. *BL_State always returns immediately* reporting the capture engine is:

- **BL_STATE_IDLE** - idle and ready to program
- **BL_STATE_ACTIVE** - actively capturing data
- **BL_STATE_DONE** - completed capturing data successfully
- **BL_STATE_ERROR** - completed capturing data unsuccessfully

The *only state in which it is valid to call BL_Acquire* is **BL_STATE_DONE**. This state means the trace completed successfully, either because a trigger was seen or a timeout occured but in either case data may be acquired from the device.

If **BL_STATE_ERROR** is reported *an error occured during the most recent trace*. The trace has completed but the data (if any) cannot be relied upon as reliable. Call BL_Error to learn more about the error and what you may need to do to recover.

While **BL_STATE_ACTIVE** is reported *you must not call any other function* to access the device unless you halt it first. If **BL_STATE_IDLE** is reported you can program the device as required but data is not guaranteed to be available for BL_Acquire.

BL_State *may be called at any time after the library has been initialized and device selected*. If BL_Trace is called synchronously *it reports the same information* except in this case the BL_STATE_ACTIVE state will never be reported (because it prevails only for as long as the synchronous call to BL_Trace does).

BL_Time - assign the capture duration (seconds)

Enquire the capture duration for the next trace.

```
double BL_Time(double ACaptureTime); /* C/C++ Prototype */
function BL_Time(ACaptureTime : Double = BL_ASK) : Double; { Pascal Prototype
}
```

The capture duration *is usually assigned implicitly* (as a consequence of the prevailing BL_Rate and BL_Size). It *may also be specified* explicitly but if it is, the sample rate and trace size may also be modified their values should be rechecked.

The recommended use for this function is to enquire what the prevailing duration is without changing it by specifing **BL_ASK**. Regardless of how BL_Time is called **always check the return value** because *it is the value that is actually used*.

BL_Trace - initiate capture

Initiate data capture on the device:

```
bool BL_Trace(double ATimeOut, bool ASync); /* C/C++ Prototype */
function BL_Trace(ATimeOut : Double = BL_TRACE_FORCED; ASync : Boolean = False
) : Boolean; { Pascal Prototype }
```

The process of capturing data on BitScope is referred to as a **trace**. Call **BL_Trace** to *initiate a trace* **after calling the other functions** as required to setup the trace.

BL_Trace is unique because the time it takes to execute can be indeterminate (which depends on the trigger). You must take care to call it correctly, especially if

called from your main application thread; it **may never return** if *called synchronously without a timeout*. It can be recovered in this case, *but only from a different thread*.

To avoid complications, BL_Trace offers two calling conventions:

1. **asynchronous** with or without a specified timeout
2. **synchronous** in which case a timeout is recommended.

The first argument (**ATimeOut**, in seconds) specifies the timeout used and the second (**ASync**) specifies the call is to be asynchronous. The **return value** indicates whether BL_Trace commenced successfully (in the asynchronous case) or commenced *and completed successfully* (in the synchronous case).

The simplest usage is a **synchronous call with a non-zero timeout**. In this case BL_Trace will block the caller and return when:

- the trace has successfully completed, or
- a timeout occured while waiting for a trigger, or
- an error occured while talking to the device.

If the timeout is short (on a human timescale), this type of usage is compatible with single threaded programming for an interractive program. It is simple because the return value of BL_Trace may be used directly to detemine success (i.e. BL_State is not required) and all programming is synchronous.

If the timeout required is long or you need to specify infinite timeout (i.e. the trigger timing is unknown), an asynchronous call is recommended (unless you are calling BL_Trace from a different thread). In this case BL_Trace will always *return to the caller immediately* but **the trace may not have completed**. The return value in this case will always be TRUE *unless an error occured when talking to the device*.

When BL_Trace is called asynchronously **you must poll the trace state** (BL_State) to determine when the trace has completed; it is a programming error to use any other library function (except BL_Halt) before the trace has completed.

Two special values for the timeout are available:

- **BL_TRACE_FORCED** - commence immediately regardless of the trigger
- **BL_TRACE_FOREVER** - commence and wait for the trigger, possibly forever.

Use the former if want a trace and you don’t care about the trigger condition (the trigger condition is **ignored** in this case). Use the latter if you want to wait for **as long as it takes** for a trigger. If the trigger condition is never met, the trace will never complete unless manually halted.

BL_Trigger - set up the trigger

Establish the trigger condition to apply to the selected channel.

```
bool BL_Trigger(double ALevel, int AEdge); /* C/C++ Prototype */
function BL_Trigger(ALevel : Double; AEdge : Integer) : Boolean; { Pascal Prot
otype }
```

If BL_Trace is *used with a timeout* or with **BL_TRACE_FOREVER**, a *trigger condition* is applied *as a predicate to the completion of the trace*. This means the trace will complete only when the trigger condition is satisfied.

Use **BL_Trigger** to establish a trigger condition on a selected channel. Call it multiple times, once for each channel, if a combinatorial condition is required to apply to more than one channel.

BL_Version - return the version

Enquire the device, library, bindings and other version information.

```
char * BL_Version(int ATarget); /* C/C++ Prototype */
function BL_Version(ATarget : Integer = BL_VERSION_DEVICE) : PAnsiChar; { Pasc
al Prototype }
```

When using the library is may necessary to know the device, library or other version information. This functions provides access to this information as it applies to the selected device. There are several version types that may be requested:

- **BL_VERSION_DEVICE** - device model and version identifier
- **BL_VERSION_LIBRARY** - library version and production build ID
- **BL_VERSION_BINDING** - language binding and version
- **BL_VERSION_PLATFORM** - build platform version
- **BL_VERSION_FRAMEWORK** - library framework version
- **BL_VERSION_NETWORK** - network communication version

In general you will likely be interested in the first two. The other versions are available to help diagnosis in the event that you discover problems when using a particular instance of the library or its language bindings.

License

Version

BitScope Library V2.0 Build (DK06B) - User Programming Manual (EC21A)

---

1. In this case an **Application Programming Interface (API)** is this library which includes a detailed specification for a set of callable functions (ref) ↩
2. **C** is a general-purpose programming language. It is one of the most widely used programming languages of all time (ref) ↩
3. **C++** is a statically typed, free-form, multi-paradigm, compiled, general-purpose programming language. It is regarded as an intermediate-level language, as it comprises both high-level and low-level language features. It is one of the most popular programming languages and is implemented on a wide variety of hardware and operating system platforms (ref) ↩
4. **Python** is a general-purpose, high-level dynamic programming language whose design philosophy emphasizes code readability. It supports multiple programming paradigms, including object-oriented, imperative and functional programming styles (ref) ↩
5. **Pascal** is an influential imperative and procedural programming language first published in 1970 which has since been extended with object oriented capabilies. It is a small and efficient language intended to encourage good programming practices using structured programming and data structuring (ref) ↩