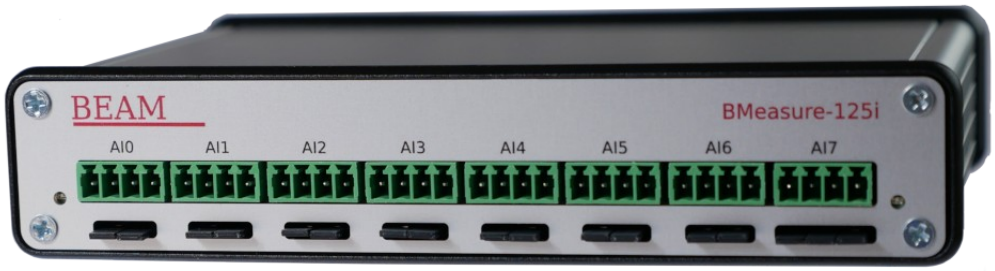
BEAM

BMeasure-lib API Manual

Product	BMeasure-125i
API Manual Version	1.3
Hardware Version	1.4.0
Software version	1.3.0
Date	2024-12-19



1. Contents

Table of Contents

2. Introduction

The Beam BMeasure-125i unit is a flexible and powerful IoT system for data capture, data logging and control in the laboratory, industrial and remote sensing arenas. It is based around an 8 channel, fully differential, synchronous sampling, 24 bit ADC that can sample at speeds up to 128 ksps. Multiple units can be connected together to provide more synchronously sampled channels.

This document describes the host API library allowing programs to be written to control the operation of a BMeasure unit and acquire the data from it. The API operates over a number of different physical interfaces including: USB 2.0, Ethernet, Wifi and RS485.

In addition to this document there is the on-line doxygen description of all of the API functions which can be accessed at: https://www.beam.ltd.uk/files/products/bmeasure-125i/doc/bmeasure-api/html/index.html and a set of C++ and Python examples.

3. Overview

The BMeasure API library, bmeasure-lib, is implemented in the C++ computer language. It has bindings layered on top of this for Python, with Matlab due to be supported soon. The API has an object orientated architecture. It has been designed as a general purpose API library for the Beam BMeasure-125i and future BMeasure products. Currently it has ports to Linux (Redhat7, Fedora29, Ubuntu) and Microsoft Windows 7, 8 and 10.

The API provides the following functionality:

- Find BMeasure units on the USB bus or local Ethernet and Wifi networks.
- Connect to one or more BMeasure units
- Fetch information and configure the BMeasure units.
- Start the BMeasure unit capturing and processing the sensor inputs.
- Capture the data from all of the analogue and digital channels from one or a combined set of BMeasure units running in sync.
- Access the data log files on the unit and download them to the host.
- Configure the AWG to produce waveforms or set voltages on the analogue output channels.
- Operate relays, read switches and other auxiliary operations.

The BMeasure API is implemented using the Beam BOAP (Beam Object Access protocol) communications system. It offers an BMeasureUnit API class to access an individual BMeasure unit in a relatively low level manner and an BMeasureUnits API class to access a set of BMeasure units synchronised together to operate as a single unit and with a queued data reception system..

The API supports threaded and non-threaded operation.

The following manual describes the API from a C++ programming perspective. The Python and other language bindings are very similar the differences being noted under the particular language bindings section.

4. Beamlib

The API uses the Beamlib C++ class library. This library provides a set of cross-platform simple classes for strings, errors, networking, threading, DSP etc. The core classes include:

BString	A simple, variable length, ASCII or UTF8 string class. It supports standard string operations such as concatenation, searching and wild-card comparisons etc.
BError	This class encapsulates and error number and an errors string. Most functions return a BError object to indicate the status of the functions operation.
BList<Type>	A simple doubly linked list of a particular object
BArray<Type>	A single dimensional array of objects laid out in consecutive memory locations.
BSocket	Network socket

Below is a set of short examples to show basic usage of the core Beamlib classes.

BStrings store variable length ASCII or UTF8 strings on the heap. It uses efficient shared reference counted string. Some examples:

```
BString         s1;
BString         s2;

s1 = s1 + s2;
if((s1 == s2) || (s1 > s2)){
}

s1 = s2.subString(0, 3);
s1 = stringToUppercase(s2);

cout << s1 << "\n";
printf("%s\n", s1.retStr());
```

BError class is used to return errors from functions, it encapsulates an error number and a string. It can be used in "if" statements and passed back up the calling function tree as well as through exceptions. For example:

```
BError          err;

if(err = func()){
        cerr << "Error: num: " << err.num() << " string: " << err.getString() << "\n";
        printf("Error: num: %d string: %s\n", err.num(), err.str());
}
```

The BList class offers a template list allowing different objects to be stored. Some examples of its use are show below:

```
BList<int>      l;
BIter           i;

l.append(1);
l.append(2);
for(l.start(i); !l.isEnd(i); l.next(i)){
        printf("Value: %d\n", l[i]);
}
```

The BArray class offers a template array allowing different objects to be stored in consecutive memory locations. Some examples of its use are show below:

```
BArray<BString>      a;

a.setSize(64);
a[32] = “Hello”;

for(int i = 0; i < a.size(); i++){
        printf(“Position: %d Value: %s\n”, i, a[i].retStr());
}
```

The Beamlib library is documented at:
https://portal.beam.ltd.uk/files/products/bmeasure-125i/doc/beamlibApi/html/index.html

5. BMeasure Unit Naming and Searching for Units

BMeasure units are accessed using a URL unit naming string. For the three different physical interfaces this has a format of:

- USB: “boapu://<serial number>”
- Network: “boapn://<IP address or host name>”
- RS485: “boaps://<serial device name>”

To connect to a BMeasure unit the appropriate URL string can be used or a search of the local USB devices or local network devices can be carried out. There are two functions available to search for USB and Network devices. These are:

BError BMeasureUnit::findDevicesUsb(BList<BMeasureUnitDevice>& devices);

and

BError BMeasureUnit::findDevicesNetwork(BList<BMeasureUnitDevice>& devices);

These functions return the standard BError object to indicate any errors and a list of the BMeasure devices found.

The BMeasureUnitDevice objects have two BString members: serialNumber holding the units serial number string and device holding the device access URL.

6. BMeasureUnit: Accessing a single BMeasure unit

The BMeasureUnit class provides all of the functions to control and fetch data from a single BMeasure unit in a relativelt low level and simple way. Many of its functions result in a RPC (remote procedure call) to the BMeasure unit over the physical communications interface in use. The BMeasure class it is based on provides much of this functionality.

There is a connect() function to connect to an individual BMeasure unit whereupon any of the functions can be used. In order to receive data you mush derive a new class from this and override the void sendDataServe1(const DataBlock &dataBlock); function. This function will be repeatedly called with a set of data in a DataBlock. The status field in the DataBlock will indicate the end of a measurement process and any errors that have occurred.

The BMeasureUnit class can operated in threaded or non-threaded modes. Typically it is used in a

threaded mode (first parameter to the BMeasureUnit's constructor is set to 1 or True). In the threaded mode an internal thread handles reception of packets from the BMeasure unit. This includes replies to commands and asynchronous data. The sendDataServe1() function will be called from this internal thread. In non threaded operation the RPC calls will function as normal but if data is to be acquired either the manual measure() function has to be used to acquire a single set of data samples or the classes processRx() function must be continually called to process the incoming data packets.

Two typical simple examples of its use for simple configuration follows:

```
/*****************************************************************************
 *    Example001-config.cpp
 *    T.Barnaby,  BEAM Ltd,   2018-11-28
 *****************************************************************************
 */
#include <BMeasureUnit.h>
#include <unistd.h>

using namespace BMeasureApi;

// Function to configure AWG
BError test1(){
      BError                        err;
      BList<BMeasureUnitDevice>     devices;
      BString                       device;
      BMeasureUnit                  bmeasure;
      Information                   info;
      Configuration                 config;
      AwgConfig                     awg;

      printf("Find BMeasure units\n");
      if(err = BMeasureUnit::findDevicesUsb(devices)){
            return err;
      }
      if(devices.number() == 0){
            return err.set(1, "No USB BMeasure units found\n");
      }
      device = devices[0].device;

      printf("Start Processing Task\n");
      bmeasure.start();

      printf("Connect\n");
      if(err = bmeasure.connect(device))
            return err;

      printf("Get Info\n");
      if(err = bmeasure.getInformation(info))
            return err;

      printf("NumChannels: %d\n", info.numChannels);

      //printf("Exit\n"); return err;

      printf("Get Config\n");
      if(err = bmeasure.getConfig(config))
            return err;

      // Set AWG
      printf("Set AWG\n");
```

```
	//awg.waveform = WaveformNone;
	awg.waveform = WaveformSine;
	//awg.waveform = WaveformSquare;
	//awg.waveform = WaveformTriangle;
	//awg.waveform = WaveformNoise;
	awg.frequency = 1000;
	awg.amplitude = 4;
	awg.offset = 0;
	awg.duty = 50;
	err = bmeasure.setAwgConfig(awg);

	return err;
}

int main(){
	BError	err;

	if(err = test1()){
		printf("Error: %d %s\n", err.getErrorNo(), err.str());
		return 1;
	}

	printf("Complete\n");

	return 0;
}
```

An equivalent of this in the Python language follows:

```
#!/usr/bin/python3
################################################################################
#	example002-config.py	BMeasure API example code for a configuration Client
#			T.Barnaby,  BEAM Ltd,   2018-10-20
################################################################################
#
# Example code to show operating a BMeasure instrument.
#
import sys
import time
import getopt
from threading import Thread
from bmeasure import *

# Function to read some data
def find():
	print("Find devices");
	(err, devices) = BMeasureUnit.findDevicesUsb();

	return err;

def test1():
	bmeasure = BMeasureUnit(True);

	print("Find BMeasure units");
	(err, devices) = BMeasureUnit.findDevicesUsb();
	if(err):
		return err;

	if(devices.number() == 0):
```

```
            return err.set(1, "No USB BMeasure units found\n");

      print("Found", len(devices));
      device = devices[0].device;

      print("Start Communications Task");
      bmeasure.start();

      print("Connect");
      err = bmeasure.connect(device);
      if(err):
            return err;

      print("Get Info");
      (err, info) = bmeasure.getInformation();
      if(err):
            return err;

      print("NumChannels: "), info.numChannels;

      print("Get Config");
      (err, config) = bmeasure.getConfig();
      if(err):
            return err;

      # Set AWG
      print("Set AWG");
      awg = AwgConfig();
      #awg.waveform = WaveformNone;
      awg.waveform = WaveformSine;
      #awg.waveform = WaveformSquare;
      #awg.waveform = WaveformTriangle;
      #awg.waveform = WaveformNoise;
      awg.frequency = 1000;
      awg.amplitude = 4;
      awg.offset = 0;
      awg.duty = 50;
      err = bmeasure.setAwgConfig(awg);
      if(err):
            return err;

      return err;


def main():
      if(0):
            err = find();
            if(err):
                  print("Error:"), err.getErrorNo(), err.getString();
                  return 1;

      err = test1();
      if(err):
            print("Error:"), err.getErrorNo(), err.getString();
            return 1;

      print("Complete");

      return 0;
```

```
if __name__ == "__main__":
      main();
```

There are more examples in the examples directory of the software release.

7. BMeasureUnits: Accessing multiple synchronised BMeasure units

The BMeasureUnits class offers a more sophisticated API interface that allows access to multiple synchronised BMeasure units as if they were a single unit. It also offers a data buffering system to simplify data capture. It can be used for one or more BMeasure units.

The BMeasureUnits class manages a set of BMeasure units using the unit*() functions. Typically you would use the unitAdd() or unitsFind() functions to add a unit to its list or search for units adding all found it its internal list. Once the units have been added they can be set into a particular order. The first unit, unit 0, will be used as the master unit for synchronisation purposes. Once the units have been setup the unitsConnect() function can be employed to connect to all of the units.

The BMeasureUnits class provides a reduced set of the BMeasureUnit classes RPC calls to the BMeasure units. These functions either apply to the master unit or all of the units. For example the setConfigMeasurement() sets the measurement configuration for all of the units. An individual units API can be accessed using the unit() function which returns a reference to a BMeasureUnit1 object used for communications.

Data is handled by an inbuilt DataBlock queue which can have up to 2 readers. A client program can use the dataWait() function to wait for data, the dataRead() function to get a pointer to the next DataBlock and the dataDone() function to tell the BMeasureUnits class that it has finished with the DataBlock.

An example of a simple data capture client follows:

```
/*************************************************************************
 *    Example010-dataClient-multi.cpp
 *    T.Barnaby,  BEAM Ltd,   2019-10-09
 *************************************************************************
 */
#include <BMeasureUnits.h>
#include <unistd.h>

using namespace BMeasureApi;

// Function to read some data
BError test1(){
      BError                        err;
      BList<BMeasureUnitDevice>     devices;
      BString                       device;
      BMeasureUnits                 bmeasure(1);
      Information                   info;
      Configuration                 config;
      MeasurementConfig             mc;
      DataBlock*                    data;
      BUInt                         c;

      printf("Start Processing Task\n");
      bmeasure.start();
```

```
printf("Find BMeasure units\n");
if(err = BMeasureUnit::findDevicesUsb(devices)){
	return err;
}
if(devices.number() == 0){
	return err.set(1, "No USB BMeasure units found\n");
}

if(err = bmeasure.unitAdd("", devices[0].device))
	return err;

printf("Connect\n");
bmeasure.unitSetEnabled(0, 1);
if(err = bmeasure.unitsConnect())
	return err;

printf("Get Info\n");
if(err = bmeasure.getInformation(info))
	return err;

printf("NumChannels: %d\n", info.numChannels);

//printf("Exit\n"); return err;

printf("Get Config\n");
if(err = bmeasure.getConfig(config))
	return err;

printf("Configure measurement\n");
mc.measureMode = MeasureModeOneShot;
mc.triggerMode = TriggerModeOff;
mc.triggerConfig = TriggerConfigNone;
mc.triggerChannel = 0;
mc.triggerLevel = 0;
mc.triggerDelay = 0;
mc.sampleRate = 8000.0;
mc.measurePeriod = 0;
mc.numSamples0 = 100;
mc.numSamples1 = 0;
if(err = bmeasure.setMeasurementConfig(mc))
	return err;

printf("Run measurement\n");
config.captureData = 1;
if(err = bmeasure.setConfig(config))
	return err;

bmeasure.dataSetNumStreams(1);

if(err = bmeasure.setMode(ModeRun))
	return err;

printf("Loop fetching data\n");
while(1){
	if(err = bmeasure.dataWait(0))
		return err;

	data = bmeasure.dataRead(0);
	printf("DataBlock: from: %d numChannels: %d numSamples: %d\n", data-
```

```
>source, data->numChannels, data->numSamples);
            bmeasure.dataDone(0);

            if(! (data->status & StatusRun))
                  break;
      }

      bmeasure.unitsDisconnect();

      return err;
}

int main(){
      BError      err;

      if(err = test1()){
            printf("Error: %d %s\n", err.getErrorNo(), err.str());
            return 1;
      }

      printf("Complete\n");

      return 0;
}
```

8. Digital IO

The BMeasure-125i has 8 digital IO pins. These pins, labels DIO0 through DIO7 can be configured as input or output lines or be used for special functions. They support 0 – 3.3 Volt output and up to 5 Volts on input.

The mode of operation of the digital IO pins can be set using the digitalMode and digitalPins parameters in the BMeasure Configuration. The digitalMode parameter can be set to one of the following:

DigitalModeInput	All 8 pins are set as digital inputs
DigitalModeOutput	All 8 pins are set as digital outputs
DigitalModeInOut	DIO0 – DIO3 are outputs and DIO4 – DIO7 are inputs
DigitalModeSyncMaster	The digital IO lines DIO5 – DIO7 are configured as outputs used for multiple BMeasure-125i synchronisation. The digital lines DIO0 – DIO4 are configured as general inputs.
DigitalModeSyncSlave	The digital IO lines DIO5 – DIO7 are configured as inputs used for multiple BMeasure-125i synchronisation. The digital lines DIO0 – DIO4 are configured as general inputs.
DigitalModeIndividual	The digital IO lines are individually configured as per the settings in the digitalPins parameter.

When digitalMode is set to DigitalModeIndividual then the mode of each digital IO line can be set individually and the digital IO pin and be routed to different internal lines using the digitalPins array. The digitalPins array has 8 x 8 bit entries, one for each of the digital IO pins (0 – 7). The 8 bit field for each pin is split into two 4 bit parts:

Bits 0 - 3	This is used to define which internal line the external pin is connected to. Normally this has the same line number as the digital IO pin number. However if special functions are to be used, such as serial UART, I2C or SPI etc., this is used to define which internal line to use.
Bits 4 - 7	This defines the digital pin mode. This can be: DigPinModeInput = 0: The pin is an input DigPinModeOutput = 1: The pin is an output DigPinModeI2cDat = 2: The pin is a special I2C DAT line.

The internal digital IO special function lines are as follows:

Line	***Function***	***Notes***
0	spi3_clk	SPI interface clock
1	spi3_cs	SPI interface chip select (active low)
2	spi3_mosi	SPI interface master out slave in data
3	spi3_miso	SPI interface master in slave out data
4	ser2_tx	Serial UART transmit line
5	ser2_rx	Serial UART receive line
6	i2c4_clk	I2C Clock line
7	i2c4_sda	I2C Data line

The digitalPins array allows any external digital IO pin to be routed to any of the internal lines and thus special functions by setting the line numbers they are routed to. Make sure that if you change the default mapping of digital IO pins to internal line numbers that no two pins are routed to a single internal line as inputs.

For example to set pin 7 as a serial UART RX input line and all other lines as digital outputs you can configure the digitalPins array as per:

Array Index/ DIO pin	***Setting***	***Note***
0	0x10	Output
1	0x11	Output
2	0x12	Output
3	0x13	Output
4	0x14	Output
5	0x17	Output. Note that digital pin 5 is routed to internal line 7 as internal line 5 is used for serial RX

6	0x16	Output
7	0x05	Routes digital pin 7 to internal line 5 for serial RX as an input.

The API functions: setDigital() and getDigital(), can be used to set the state of output pins or read the state of input or output pins. The parameter is an 8 bit value with each bit defining the sate of one digital IO line.

When the pin to lines settings are changed from the default 1:1 settings, then the API's setDigital() and getDigital() functions perform the necessary swaps such that the bit settings match the external pin numbering. Also the data capture uses the external pin numbering. The digitalPins swap is used only to route the pins to special functions.

9. Language Bindings

9.1. C++

The bmeasure-lib API is written in C++, so this is the base API that is documented in the doxygen based on-line API documentation.

9.2. Python

The Python API is built on top of the standard BMeasure-lib 'C++' API using the SWIG API generator. Thus all of the standard BMeasure-lib C++ API documentation applies however there are some differences due to the language facility and syntax differences.

The Python language is interpreted rather than compiled and does not require strict types like 'C++'. This can help speed up the development of simple tools and programs, but it can result in less robust and less maintainable code.

To use the BMeasure-lib API library import the "bmeasure" module. The SWIG system wraps the BDS C++ objects in a Python object layer. Your can then interact with the C++ objects from Python in the same way as you would have done in C++ apart from a few differences.

1. Reference returns: 'C++' allows references/pointers to be passed as function arguments which allows functions to return values. Python does not support this. Instead Python provides the ability for functions to return multiple items on the left-hand side. When using an API call that in 'C++' returns items by reference, the Python equivalent will have these returned on the left hand side. For example:

   ```
   err = bmeasure.getStatus(NodeStatus& status);      // C++
   (err, status) = bmeasure.getStatus();              # Python
   ```

2. Testing for error returns. All BDS API calls return a BError object. This provides information as to if the function completed successfully or if there was an error. The BError object contains both an error number and an error string. 'C++' allows an "if" statement to have an assignment operator. This makes returning and checking errors quite concise. Python does not allow this and requires a separate assignment and if statement. For example:

```
if(err = bmeasure.getStatus(status)){
        return err;
}
(err, status) =  bmeasure.getStatus();
if(err):
        return err;
```

10. Physical Interfaces

A BMeasure unit has a number of possible interfaces that the bmeasure-lib will communicate over. These include the following.

10.1. USB 2.0

This interface is suitable for local data acquisition and configuration uses. When connected to a USB 2.0 full speed (480 MBits/s) host it can transfer data at the full real-time capture rate of 128 ksps.

10.2. Ethernet

This interface is suitable for local and remote data acquisition and configuration uses. It can transfer data at up to 100 MBits/s and assuming the network path supports a bit rate of greater than 40 Mbits/s to the host, it can transfer data at the full real-time capture rate of 128 ksps.

10.3. Wifi

The BMeasure unit can connect via is internal Wifi interface to a local Wifi access point. The bmeasure-lib API can connect over this using the normal TCP/IP networking system. It's maximum bit rate is 20 Mbits/s but this is limited by the Wifi access point, the distance from the Wifi AP and the number of Wifi and other 2.4 GHz spectrum users. At 20 MBits/s, it can transfer data at the full real-time capture rate of 64 ksps but 32 ksps is a more likely limit.

10.4. RS485

The RS485 interface provides a local or long range communications interface. It can operate at up to 1 MBits/s over about 400m range. At 100kBits/s it can operate over a 1km range.

11. Supported Systems

11.1. Linux systems

The BMeasure software install system installs the C++ and Python bmeasure-lib API development files onto the system in question. All of the BMeasure files are located in the /usr/share/BMeasure directory. We support 64bit systems only. The Python and C++ examples are in /usr/share/BMeasure/examples.

For Python development you should install the Python3 environment using your systems software package management system. For C++ you should install the C++ development toolset.

The BMeasure package sets the PYTHONPATH environment variable to point to the bmeasure-lib Python module so that python examples can be run. Note that the system works only with Python 3.

For C++ development, the directories /usr/share/BMeasure/include/Beam and

/usr/share/BMeasure/include/BMeasure should be added to the include path of your C++ development environment and the libraries, libBeam.a and libBMeasure.a linked from the path /usr/share/BMeasure/lib64. The Makefile in the examples shows how to do this for a simple make/gcc based build.

11.2. Microsoft Windows systems

The BMeasure software install system installs the C++ and Python bmeasure-lib API development files onto the system in question. All of the BMeasure files are located in the “C:/Program Files/BMeasure” directory. We support 64bit systems only but have 32bit versions of the API libraries available. The Python and C++ examples are in “C:/Program Files/BMeasure/examples”.

For the bmeasure-lib Python host API, we support the standard Python 3.7.4 from https://www.python.org/. This can be downloaded and installed from: https://www.python.org/downloads/release/python-374/. The Windows x86-64 executable installer is the normal method.

Notes:

- Use the Custom install option and tick the option to install for all users.
- It is useful to tick the option “Add Python 3.7 to PATH”

The BMeasure package sets the PYTHONPATH environment variable to point to the bmeasure-lib Python module so that python examples can be run. Note that the system works only with Python 3.

11.2.1. Minggw-64

For C++ development, the directories “C:/Program Files/BMeasure/include/Beam” and “C:/Program Files/BMeasure/include/BMeasure” should be added to the include path of your C++ development environment and the libraries, libBeam.a and libBMeasure.a linked from the path “C:/Program Files/BMeasure/lib64”. The Makefile in the examples shows how to do this for a simple GNU make/gcc (Mingw 64bit) based build.

11.2.2. Window Visual Studio

For C++ development, the directories “C:/Program Files/BMeasure/include/Beam” and “C:/Program Files/BMeasure/include/BMeasure” should be added to the include path of your C++ development environment and the libraries, libBeam.a and libBMeasure.a linked from the path “C:/Program Files/BMeasure/lib64”.

The pthreads package should be installed. In visual studio use the menu item -

Project → Manage Nuget Packages

In browse tab search for phtread

Select install

12. More Information

For more information please refer to our WEB site at:
http://www.beam.ltd.uk/products/bmeasure-125i/index.html

The online API reference is at: http://www.beam.ltd.uk/products/bmeasure-125i/doc/bmeasure-lib-reference/html/index.html