Bertec Device Interface Library for Java

Developer Documentation

Version 1.82
March 2014

Bertec's authorized representative in the European Community regarding CE:

Bertec Limited
31 Merchiston Park
Edinburgh EH10 4 PW
Scotland, United Kingdom

CE

SOFTWARE LICENSE AGREEMENT

This License Agreement is between you (“Customer”) and Bertec Corporation, the author of the Bertec Device DLL software and governs your use of the of the dynamic link libraries, example source code, and documentation (all of which are referred to herein as the "Software").

PLEASE READ THIS SOFTWARE LICENSE AGREEMENT CAREFULLY BEFORE DOWNLOADING OR USING THE SOFTWARE. NO REFUNDS ARE POSSIBLE. BY DOWNLOADING OR INSTALLING THE SOFTWARE, YOU ARE CONSENTING TO BE BOUND BY THIS AGREEMENT. IF YOU DO NOT AGREE TO ALL OF THE TERMS OF THIS AGREEMENT, DO NOT DOWNLOAD OR INSTALL THE SOFTWARE.

- Bertec Corporation grants Customer a non-exclusive right to install and use the Software for the express purposes of connecting with Bertec Devices for data gathering purposes. Other uses are prohibited.
- Customer may make archival copies of the Software provided Customer affixes to such copy all copyright, confidentiality, and proprietary notices that appear on the original.
- The Customer may not resell the Software or otherwise represent themselves as the owner of said software.

The binary redistributables are royalty free to the original Licensee and can be distributed with applications, provided that proper attribution is made in the documentation and end user agreement. Binary redistributables include:

1. BertecDeviceDLL.dll
2. BertecDeviceJava.jar
3. ftd2xx.dll

Note that the FTD2XX.DLL is a USB driver provided by Future Technology Devices that enables communication with the Bertec Device.

The binary redistributables cannot be used by third parties to build applications or components.

Customer created binary redistributables from the Software source code cannot be used by anyone, including the original license holder, to create a product that competes with Bertec Corporation products. Neither the original nor altered source code may be distributed.

EXCEPT AS EXPRESSLY AUTHORIZED ABOVE, CUSTOMER SHALL NOT: COPY, IN WHOLE OR IN PART, SOFTWARE OR DOCUMENTATION; MODIFY THE SOFTWARE; REVERSE COMPILE OR REVERSE ASSEMBLE ALL OR ANY PORTION OF THE SOFTWARE; OR RENT, LEASE, DISTRIBUTE, SELL, MAKE AVAILABLE FOR DOWNLOAD, OR CREATE DERIVATIVE WORKS OF THE SOFTWARE OR SOURCE CODE.

Customer agrees that aspects of the licensed materials, including the specific design and structure of individual programs, constitute trade secrets and/or copyrighted material of Bertec Corporation. Customer agrees not to disclose, provide, or otherwise make available such trade secrets or copyrighted material in any form to any third party without the prior written consent of Bertec Corporation. Customer agrees to implement reasonable security measures to protect such trade secrets and copyrighted material. Title to Software and documentation shall remain solely with Bertec Corporation.

No Warranty

THE SOFTWARE IS BEING DELIVERED TO YOU "AS IS" AND BERTEC CORPORATION MAKES NO WARRANTY AS TO ITS USE, RELIABILITY OR PERFORMANCE. BERTEC CORPORATION DOES NOT AND CANNOT WARRANT THE PERFORMANCE OR RESULTS YOU MAY OBTAIN BY USING THE SOFTWARE. BERTEC CORPORATION MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AS TO NONINFRINGEMENT OF THIRD PARTY RIGHTS, TITLE, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. YOU ASSUME ALL RISK ASSOCIATED WITH THE QUALITY, PERFORMANCE, INSTALLATION AND USE OF THE SOFTWARE INCLUDING, BUT NOT LIMITED TO, THE RISKS OF PROGRAM ERRORS, DAMAGE TO EQUIPMENT, LOSS OF DATA OR SOFTWARE PROGRAMS, OR UNAVAILABILITY OR INTERRUPTION OF OPERATIONS. YOU ARE SOLELY RESPONSIBLE FOR DETERMINING THE APPROPRIATENESS OF USE OF THE SOFTWARE AND ASSUME ALL RISKS ASSOCIATED WITH ITS USE.

Indemnification

You agree to indemnify and hold Bertec Corporation, parents, subsidiaries, affiliates, officers and employees, harmless from any claim or demand, including reasonable attorneys' fees, made by any third party due to or arising out of your use of the Software, or the infringement by you, of any intellectual property or other right of any person or entity.

Limitation of Liability

IN NO EVENT WILL BERTEC CORPORATION BE LIABLE TO YOU FOR ANY INDIRECT, INCIDENTAL, SPECIAL, PUNITIVE, CONSEQUENTIAL, OR OTHER DAMAGES WHATSOEVER, OR ANY LOSS OF REVENUE, DATA, USE, OR PROFITS, EVEN IF BERTEC CORPORATION HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, AND REGARDLESS OF WHETHER THE CLAIM IS BASED UPON ANY CONTRACT, TORT OR OTHER LEGAL OR EQUITABLE THEORY.

This License is effective until terminated. Customer may terminate this License at any time by destroying all copies of Software including any documentation. This License will terminate immediately without notice from Bertec Corporation if Customer fails to comply with any provision of this License. Upon termination, Customer must destroy all copies of Software.

Software, including technical data, is subject to U.S. export control laws, including the U.S. Export Administration Act and its associated regulations, and may be subject to export or import regulations in other countries. Customer agrees to comply strictly with all such regulations and acknowledges that it has the responsibility to obtain licenses to export, re-export, or import Software.

This License shall be governed by and construed in accordance with the laws of the State of Ohio, United States of America, as if performed wholly within the state and without giving effect to the principles of conflict of law. If any portion hereof is found to be void or unenforceable, the remaining provisions of this License shall remain in full force and effect. This License constitutes the entire License between the parties with respect to the use of the Software.

Should you have any questions concerning this Agreement, please write to:

Bertec Corporation, 6171 Huntley Road, Suite J, Columbus, Ohio 43229

TABLE OF CONTENTS

INTRODUCTION

The Bertec Device Library for Java provides the end-user developer or data acquisition expert a common and consistent method to gather data from Bertec equipment. Instead of directly communicating with the USB devices and implementing different protocols and calibrations for each, the Bertec Device Library for Java manages all interaction with the USB devices, and provides the calibrated captured data to your program or data analysis project. The Library also provides zeroing of the plate data (either on-demand for tare loading, or automatic for low or no loading), sample averaging, and low-pass filtering.

The Library provides data results in either an event callback or a polled-data mode, depending on the needs and abilities of the data acquisition application.

The Library exports its functionality as a typical Java class library or “jar” file, which can be used by any Java-compliant development environment or data acquisition programs. As long as your development system or data acquisition software can use Java libraries, then you should have no problems with using the Library.

Sample code is provided in the BertecExampleJAVA.java file.

If you are doing development in a C or C++ environment, please refer to the BertecDevice.pdf file.

If you are doing development using .NET, please refer to the BertecDeviceNET.pdf file. This can also be used as a reference for COM-based development.

If you have any developmental questions on using this library or SDK, please contact Bertec Corporation for support.

DEFINITIONS, ACRONYMS, AND ABBREVIATIONS

Balance plate: a Bertec device that measures pressure and movement that is optimized for balance diagnostics.

Force plate: a Bertec device that measures pressure and movement.

Center of Pressure (CoP): The point on the surface of the platform through which the ground reaction force acts. It corresponds to the projection of the subject's center of gravity on the platform surface when the subject is motionless.

USING THE LIBRARY

After you have added a Reference to the BertecDeviceJAVA library, getting data from an attached device generally consists of just a few steps:

1. Create the `BertecDevice` object.
2. Either set the `BertecDevice.PollingBufferSize` value, or else bind your event handler to `BertecDevice.DataEventListner`
3. Call `BertecDevice.Start`
4. Poll using `BertecDevice.DataPoll`, or use the event hander.
5. Call `BertecDevice.Stop`
6. Dispose of the `BertecDevice` object.

Step 1: Create the `BertecDevice` object

Creating the `BertecDevice` object will set up internal data in the Library and locate attached Bertec USB devices. It is the first thing you will need to do to use the Bertec Device Library, and the object itself is how to use to communicate with the Bertec Device. Note that creating multiple instances of this object is not supported.

Step 2: Either set the `BertecDevice.PollingBufferSize` value, or else bind your event handler to `BertecDevice.DataEventListener`

Depending on how your application works, you will either want to poll for the data yourself (pulled by you) and process it, or else use the faster event functionality (pushed to you). If polling, you will need to first tell the system how much internal buffer memory it should allocate using the `PollingBufferSize` property. If using callbacks, you will need to register your event handler with the Library via `BertecDevice.addDataEventListener` .

Step 3: Call `BertecDevice.Start`

To actually gather data, you must call `BertecDevice.Start` . Doing so will start the data gathering process, and will start calling your event handler if you have registered it.

Step 4: Poll using `BertecDevice.DataPoll`, or use the event hander.

If you're using the data polling method, you will need to repeatably call `BertecDevice.DataPoll` to gather the data; otherwise, the event handler will be used.

Step 5: Call `BertecDevice.Stop`

Once you have completed your data gathering call `BertecDevice.Stop` to end all of the USB data reads and end buffering of the data. The Library still remains active and data gathering can be resumed by calling `BertecDevice.Start` again.

Step 6: Dispose of the `BertecDevice` object

Once you are completely done with the system, you will need to dispose of the `BertecDevice` object to stop all USB devices and release the device driver memory. Failing to do so could result in the Bertec devices continually running, and possible memory leaks.

USING POLLED DATA TO PULL THE DATA YOURSELF

In order to use polled data, you will need to inform the Library how much internal buffer space it will need to reserve for itself. You do this using the `BertecDevice.PollingBufferSize` property, setting the time in seconds you wish it to buffer for. If you don't do this, then calling `BertecDevice.DataPoll` will fail. Here is a very simplistic example with no error handling:

```
BertecDevice.setPollingBufferSize(2.750);
BertecDevice.Start();
Int[] channelsOut=new int[1];
double[] yourDataBuffer=new double[8000];
do
{
  int samples = BertecDevice.DataPoll(channelsOut, yourDataBuffer);
  processYourData(...);
} while (samples>=0);
BertecDevice.Stop();
```

The `BertecDevice.DataPoll` function will return either the number data items placed into the buffer, up to the size of the buffer, or else an error code.

The error codes are defined in the `BertecDeviceNET.ErrorValue` enum collection. See the section on Error Codes for more information.

USING EVENTS TO GET THE DATA PUSHED TO YOU

Events are a very fast way to get data from the attached devices. These events are handled in a separate thread from your application's main thread – this must be taken into consideration when designing your application.

To use events, simply register your class's event handler with `BertecDevice.addDataEventListener`. You can register multiple events handlers, and they will each get called in turn. Here is a very simplistic example without error checking:

```
class myHandler implements BertecDeviceJava.BertecDevice.DataEventListener
{
  public void dataEventReceived(DataEvent event)
  {
    processYourData(...);
  }
};
BertecDevice.addDataEventListener(new myHandler() );
BertecDevice.Start();

      ..your main program runs...

BertecDevice.Stop();
```

ERROR CHECKING AND HANDLING

When using the data polling, you will need to check the return value from `BertecDevice.DataPoll`. If the value is greater than zero, then there is valid data in the buffer object that you passed to `BertecDevice.DataPoll`. If the return value is zero, there is no data on the wire to process – this is not considered an error unless it happens several times in a row, and then the Library determines that the device has been disconnected. A value less than zero will indicate an error condition. The error codes are defined in the `BertecDeviceNET.ErrorValue` enum collection. See the section on Error Codes for more information.

When using the event handler, errors are given using the `DataEvent.samples` parameter. If this value is less than zero, it indicates an error condition. Again, the error codes are defined in the `BertecDeviceNET.ErrorValue` enum collection. See the section on Error Codes for more information.

Status errors are also sent using the `BertecDevice.StatusEventListener` event handler. These events are sent whenever the status of the Library changes, either to an error condition or when cleared.

Generally speaking, the Bertec Device Library will automatically handle most error conditions that you would otherwise have to design for. If the user unplugs a device, and then reconnects it, the Library will handle this and restart the device. However, the system *will not* attempt to locate freshly attached devices that are plugged in while the Library is in use. If the Bertec device is turned off, or disconnected from the USB converter box, the Library will notice the lack of incoming data and attempt to restart the device once it is reconnected.

DATA PROCESSING

Since data can flow into the computer in a very rapid rate, it is critical that your program handle it as promptly as possible – buffering it in a large pre-allocated memory block is preferred. Should data not be read fast enough, it will start to be lost, and you will get notifications either through the return code from `BertecDevice.DataPoll` or the `samples` error value in the callback.

DATA FORMAT

The buffered data that is presented via callbacks or data polling is an array of double values, already calibrated for each device, grouped in sets of samples. What each value means is specific to the channel of that device. Each sample contains all channels of all attached transducers, starting from `BertecDevice.TransducersInfo[0]` and going up to the value in `BertecDevice.getTransducerCount()`.

Each sample is therefor arranged like this:

sample 0: tr0ch0,tr0ch1,tr0ch2,tr0ch3, tr1ch0,tr1ch1

sample 1: tr0ch0,tr0ch1,tr0ch2,tr0ch3, tr1ch0,tr1ch1

etc.

Both the event handler and data poll return to your code the total number of channels for all transducers. In the above samples, this value would be 6.

BERTEC DEVICE LIBRARY FUNCTIONS

The Bertec Device Library is in the `BertecDeviceJava` namespace. This namespace includes the following:

`BertecDeviceJava.BertecDevice` : this is the class you will need to instantiate in order to gain access to the Bertec Devices. When done with this object, you will need to dispose of it.

`BertecDeviceJava.DataEventListener` : this is the data event handler interface for getting data from the system. Use this to put the call to your own event handler into the `BertecDevice.DataEventListener` handler.

`BertecDeviceJava.StatusEventHandler` : this is the *status change* event handler interface. Use this to put the call to your own event handler into the `BertecDevice.StatusEventListener` handler.

`BertecDeviceJava.BertecDevice.ErrorValue` : enum list of error values.

`BertecDeviceJava.BertecDevice.AutoZeroStateValue` : enum list of auto zeroing states.

`BertecDeviceJava.BertecDevice.TransducerInfo` : information about each transducer, including the channel names. Retrieved using the `getTransducers()` function.

BERTECDEVICE

`berecObject = new BertecDevice()`

Creating the `BertecDevice` object initializes the library and does initial communication with the USB devices. You must first create the object before you can use the Bertec Devices. Creating the object does *not* start the data gathering process – you must call `BertecDevice.Start()` to do so. You should only create ***one*** instance of this object. Creating multiple instances will cause errors with the USB device. When you are done with the object, destroy the object or call `dispose()` on it to release the memory.

BERTECDEVICE.DISPOSE

`void BertecDevice.dispose()`

Calling the `BertecDevice.dispose` will shut down all devices, unregister all events, and stop data acquisition. You can either do this explicitly by calling this method directly, wrapper your code block with a try-finally construct, or allow the Java garbage collection to take care of it. Failing to call this when you have completed data acquisition will leave the devices running and possibly introduce memory leaks.

BERTECDEVICE.STATUS

`int BertecDevice.getStatus()`

This function returns the current status value of the system. Use the `ErrorValue` enums to determine what the status is.

BERTECDEVICE.START

`int BertecDevice.Start()`

This function starts the data gathering process, invoking the events if they are registered, and buffering incoming data as needed. The function will return a zero value if the process is started correctly, otherwise it will return an `ErrorValue.ERROR_INVALIDHANDLE` return code.

BERTECDEVICE.STOP

`int BertecDevice.Stop()`

This function stops the data gathering process. Events will no longer be called (but remained registered), and calling the `DataPoll` function will return an error. The function will return a zero value for success; otherwise it will return an `ErrorValue.ERROR_INVALIDHANDLE` return code.

DATAEVENTLISTENER INTERFACE

BERTECDEVICE.ADDDATAEVENTLISTENER

BERECDEVICE.REMOVEDATAEVENTLISTENER

`public static interface BertecDevice.DataEventListener`

`void BertecDevice.DataEventListener.dataEventReceived(DataEvent event)`

`void addDataEventListener(BertecDevice.DataEventListener listen)`

`void removeDataEventListener(BertecDevice.DataEventListener listen)`

To use the data event functionality in the system, you will need to register your event handler with `BertecDevice.addDataEventListener(yourHandlerClass)`. The system can support multiple data event handlers. To stop using the event without stopping data acquisition, call `BertecDevice.removeDataEventListener(yourHandlerClass)`. Disposing of the `BertecDevice` object will automatically unregister all callbacks.

Your event handler function should derive from the `BertecDevice.DataEventListener` interface, and implement the `dataEventReceived(DataEvent event)` function.

Events will be fired each time there is data left in the internal buffer to be processed – each event handler will be called with the same data values. The data collection is an array of doubles, and can be iterated like any other collection. Do not delete, free, or otherwise modify this data buffer.

Since data gathering is time-critical, it is important that you process the data as fast a possible and return.

You should register your events before calling `BertecDevice.Start()` in order to insure that no data is lost.

See the section on Data Events for information about the format and type of the data block.

STATUSEVENTLISTENER INTERFACE

BERTECDEVICE.ADDSTATUSEVENTLISTENER

BERTECDEVICE.REMOVESTATUSEVENTLISTENER

`public static interface BertecDevice.StatusEventListener`

`void BertecDevice.StatusEventListener.statusEventReceived(StatusEvent event)`

`void addStatusEventListener(BertecDevice.StatusEventListener listen)`

`void removeStatusEventListener(BertecDevice.StatusEventListener listen)`

To use the status callback functionality in the system, you will need to register your callback function with `BertecDevice.addStatusEventListener(yourHandlerClass)`. The system can support multiple events handlers. To stop using the event, call `BertecDevice.removeStatusEventListener(yourHandlerClass)`. Disposing of the `BertecDevice` object will automatically unregister all callbacks.

Your event handler function should derive from the `StatusEventListener` interface, and implement the `statusEventReceived(StatusEvent event)` function.

Events will be fired each time there is a change in the status of code of the Library – they will *not* be fired unless the `StatusEvent.status` value changes from N to X. The `status` value passed to the callback is the same as in the `BertecDevice.Status` value, and is provided only as a convenience factor.

Since data gathering is time-critical, and this event is made in the context of the gathering process, it is important that you process the status change as fast a possible and return.

DEVICESORTEVENTLISTENER INTERFACE

BERTECDEVICE.ADDDEVICESORTEVENTLISTENER

BERTECDEVICE.REMOVEDEVICESORTEVENTLISTENER

public static interface BertecDevice.DeviceSortEventListener

void BertecDevice.DeviceSortEventListener.deviceSortEventReceived(DeviceSortEvent event)

void addDeviceSortEventListener(BertecDevice.DeviceSortEventListener listen)

void removeDeviceSortEventListener(BertecDevice.DeviceSortEventListener listen)

To use the plate (or device) sort order callback functionality in the system, you will need to register your callback function with `BertecDevice.addDeviceSortEventListener(yourHandlerClass)`. The system can support multiple callbacks. To stop using the event, call `BertecDevice.removeDeviceSortEventListener (yourHandlerClass)`. Disposing of the `BertecDevice` object will automatically unregister all callbacks.

Your event handler function should derive from the `DeviceSortEventListener` interface, and implement the `deviceSortEventReceived(DeviceSortEvent event)` function.

Events will be fired each time the library reloads the list of devices attached to the computer.

The `DeviceSortEvent.infos` array is a list of devices discovered, and the `DeviceSortEvent.orderArray` should be manipulated by your event handler to re-order which device is #1, which is #2, etc. By default, the USB hardware orders the devices based on internal identifiers, which may or may not be order you wish to have, and the orderArray is filled in with [0,1,2,3…] . By examining the infos array (typically the serial# of the device), and changing the index values in orderArray, you can tell the library to move a device in front of others; the ordering of devices is reflected in the outgoing data stream. This allows your project to always have a consistent ordering of devices without the overhead of dealing with the USB system ordering.

BERTECDEVICE.POLLINGBUFFERSIZE

double BertecDevice.getPollingBufferSize()

void BertecDevice.setPollingBufferSize(double time)

To use data polling instead of events, you must first set up the internal buffer and tell the system how much data you wish to buffer. The system will take the value (expressed in seconds – thus 1 second is 1.00) and create an internal buffer larger enough to at least hold that amount for all channels. If you do not do this, then the data poll will return an error. You should set this before calling `BertecDevice.Start()` and `BertecDevice.DataPoll()`. The allocated memory is released when the `BertecDevice` is disposed of.

Note that setting this again will re-allocate the internal buffer, resulting in a loss of data remaining in the internal buffer.

BERTECDEVICE.DATAPOLL

`int BertecDevice.DataPoll(int[] channelsOut, double[] dataOut)`

Instead of using the event handlers, you can use the `BertecDevice.DataPoll` function to periodically pull the data from the internal buffer set up with `BertecDevice.PollingBufferSize`. You must first have a buffer set up large enough to hold at least once sample set of data, which is the sum of all the values of `TransducersInfo.ChannelCount` in the `TransducersList` collection. This sum value is returned to you in the `channelsOut` variable that you provide to the function.

You must allocate both the `channelsOut` and `dataOut` array before calling this function. `channelsOut` should be a single int array value, like so:

```
int[] channelsOut = new int[1];
```

`dataOut` should be a buffer large enough to capture a reasonable amount of data per call; small buffers will result in more round-trip calls, while a large buffer may not get filled to the maximum (so in general, bigger is better).

```
double[] dataOut = new double[8000];
```

You must call this function often enough, to keep up with the data being gathered and buffered by the system. Failing to do so will result in an error condition and loss of data.

The function returns a positive number for how many sample “rows” are returned – each “row” is `channelsOut` wide. Thus if the function returns 20, and you have 5 channels, then there are 100 data samples in the buffer. See the section on Returned Data for information about the format and type of the data block.

If the function returns a negative number, it will be one of the following `ErrorValue` values:

Error	Value	Means
`POLL_NOUSERBUFFER`	-1	The buffer size passed was too small to hold one sample. Increase the size of the buffer.
`POLL_NOPOLLBUFFER`	-2	There are no internal buffers allocated – you need to set `PollingBufferSize` first.
`POLL_CHECKSTATUS`	-3	A status change has occurred – either a device has been unplugged or some other error occurred. You will need to check `BertecDevice.Status`.
`POLL_OVERFLOW`	-4	The polling wasn't performed for long enough, and data has been lost.
`POLL_NODEVICES`	-5	There are apparently no devices attached. Attach a device.

POLL_NOTSTARTED	-6	You have not called `BertecDevice.Start` yet.
DATA_SYNCHRONIZING	-7	Synchronizing, data not available yet (this value is in `BertecDevice.Status` during callbacks, as the return code when polling)
SYNCHRONIZE_LOST	-8	If multiple plates are connected with the Bertec Sync option, the plates have lost sync with each other - check the sync cable.
SEQUENCE_MISSED	-9	One or more plates have missing data sequence - data may be invalid
SEQUENCE_REGAINED	-10	The plates have regained their data sequence
NO_DATA_RECEIVED	-11	No data is being received from the devices, check the cables
DEVICE_HAS_FAULTED	-12	The device has failed in some manner. Power off the device, check all connections, power back on
POLL_DEVICES_READY	-50	There are plates to be read
AUTOZEROSTATE_WORKING	-51	Currently finding the zero values
AUTOZEROSTATE_ZEROFOUND	-52	The zero leveling value was found

BERTECDEVICE.CLEARPOLLBUFFER

`int BertecDevice.ClearPollBuffer()`

This will clear all of the data that is currently in the polling buffer. This is only useful if using `DataPoll` and *not* using event handlers.

BERTECDEVICE.AVERAGINGSIZE

`int BertecDevice.getAveragingSize()`

`void BertecDevice.setAveragingSize(int size)`

This controls the number of samples from the devices, reducing the apparent data rate and result data by the value of `AveragingSize` (ex: a value of 5 will cause 5 times less data to come out). The `AveragingSize` value should be >= 2 in order for averaging to be enabled. Setting `AveragingSize` to 1 or less will turn off averaging (the default).

BERTECDEVICE.LOWPASSFILTERSAMPLES

`int BertecDevice.getLowpassFilterSamples()`

`void BertecDevice.setLowpassFilterSamples(int samples)`

This controls setting a running average of the previous set value, making the input data stream to appear smoother. The value should be >= 2 in order to turn the filter on. This does not affect the total number of samples gathered.

BERTECDEVICE.ZERONOW

`int BertecDevice.ZeroNow()`

By default, the data from the devices is not zeroed out. Calling the `ZeroNow` function with *any* load on the device will sample the data for a fixed number of seconds, and then use the loaded values as the zero baseline (this is sometimes called “tare” for simpler load plates). Calling this after calling `Start` will cause your data stream to rapidly change values as the new zero point is taken. This can be used in conjunction with the `EnableAutozero` property.

BERTECDEVICE.AUTOZEROING

`int BertecDevice.getAutozeroing()`

`void BertecDevice.setAutozeroing(int flag)`

The Library has the ability to automatically re-zero the plate devices when it detects a low- or no-load condition (less than 40 newtons for at least 3.5 seconds). Setting `AutoZeroing` to a non-zero value will cause the Library to monitor the data stream and continually reset the zero baseline values. Set this to a zero value to turn it off. This functionality will not interrupt your data stream, but you will get a sudden shift in values as the Library applies the zero baseline initially.

BERTECDEVICE.AUTOZEROSTATE

`int BertecDevice.getAutozeroState()`

This function returns the current state of the autozero system. Your program will need to poll this on occasion to find the current state – there is no event for when it changes.

The function returns one of the following `AutoZeroStateValue` values:

State	Value	Means
NOTENABLED	0	Autozeroing is currently not enabled.
WORKING	1	Autozero is currently looking for a sample to zero against.
ZEROFOUND	2	The zero level has been found. Note that once this value is returned, `AutozeroState` will always return it until you disable Autozeroing.

BERTECDEVICE.TRANSDUCERSERIALNUMBER

`String BertecDevice.getTransducerSerialNumber(int index)`

This returns a given transducer's serial number. This is a convenience property that can be used instead of accessing the `TransducerInfo` object.

BERTECDEVICE.TRANSDUCERSTATUS

`int BertecDevice.getTransducerStatus(int index)`

This returns a given transducer's current status. This is a convenience property that can be used instead of accessing the `TransducerInfo` object. This value is always up-to-date, whereas the `TransducerInfo` object status value may not be.

BERTECDEVICE.ACQUIRERATE

`int BertecDevice.getAcquireRate()`

`void BertecDevice.setAcquireRate(int rate)`

The Library defaults to a data push rate of approximately 5Hz. On faster equipment that might be used in a near-realtime environment, you can change this up to a rate of 20Hz (`AcquireRate` equals 1).

Note that this directly effects how often the data *callback* is given data, and how much data it is given. ***It does not effect how the device is read, only how much is pushed to <u>your</u> application in the callback. This has no effect on the `DataPoll` function.*** For example, at the default `AcquireRate` of 10, the data callback is called with about 240 samples every 210ms. Changing this value to 2 causes the callback to be called with 52 samples every 47ms, and a value of 1 is equal to about 30 samples every 31ms. At the other extreme, an `AcquireRate` of 50 equates to 1025 samples at around 1016ms.

The number of samples and effective values of `AcquireRate` are dependent upon both the Bertec Device and the Windows platform you are using.

There are very few conditions where you will need to change the rate.

BERTECDEVICE.USBTHREADPRIORITY

`void BertecDevice.setUsbThreadPriority(int pri)`

This function allows your code to change the priority of the internal USB reading thread. Typically, this is not something you will need to do unless you feel that the USB interface needs more or less of the thread scheduling that Windows performs. The priority value can range from -15 (lowest possible) to 15 (highest possible – this will more than likely prevent your GUI from running). The default system scheduling of the USB reading thread should be suitable for most applications.

BERTECDEVICE.MAYBEMISSINGSYNCCABLE

`int BertecDevice.getMayBeMissingSyncCable()`

If there are multiple devices, this will attempt to detect if the sync cable is missing, disconnected, or loose. Will return a value of 1 if the cable appears to be missing, or returns 0 if the sync cable appears to be good. Only valid when used with multiple devices connected through 650x amps with a sync cable connection.

BERTECDEVICE.RESETSYNCCOUNTERS

`int BertecDevice.ResetSyncCounters()`

If there are multiple devices, this function will reset the internal counters that account for sync offset and drifts. This is an advanced function that is typically not used and does nothing if there is only a single device connected. Returns 0 for success.

BERTECDEVICE.TRANSDUCERS

`TransducerInfo[] BertecDevice.getTransducers()`

This function returns an array of the Transducers that are currently connected to the system. Each entry into the array contains information about each Transducer or Force Plate, such as channel names, serial #, and etc. See the section on the TransducerInfo class for more information.

BERTECDEVICE.TRANSDUCERINFO CLASS

The `TransducerInfo` class contains information about each Transducer or Force Plate connected to the system. You get the collection of Transducers by using the `BertecDevice.getTransducers()` function.

BERTECDEVICE.TRANSDUCERINFO.CHANNELCOUNT

`int BertecDevice.TransducerInfo.ChannelCount`

Contains how many output channels there are. For the name of each channel and the order they appear in the output stream, see the `ChannelNames` property.

BERTECDEVICE.TRANSDUCERINFO.CHANNELNAMES

`String[] BertecDevice.TransducerInfo.ChannelNames`

Contains the names of each channel that the Transducer is delivering data on. The order of the names is the order of the data in the data stream that you get via either callbacks or `DataPoll`.

BERTECDEVICE.TRANSDUCERINFO.SAMPLINGFREQ

`int BertecDevice.TransducerInfo.SamplingFreq`

The sampling frequency of the transducer, in Hertz. Typically, this is a value of 1000.

BERTECDEVICE.TRANSDUCERINFO.SERIALNUMBER

`String BertecDevice.TransducerInfo.SerialNumber`

The serial number of the device.

BERTECDEVICE.TRANSDUCERINFO.STATUS

`int BertecDevice.TransducerInfo.Status`

The last-known status of the device. This value is a “snapshot” in time, and for more up-to-date values, you should use the `BertecDevice.getTransducerStatus(int deviceNumber)` function.

BERTECDEVICE.TRANSDUCERINFO.SYNCHRONIZED

`boolean BertecDevice.TransducerInfo.Synchronized`

If set, then the device is part of a sync group. This requires the proper Bertec amplifiers connected together using a specific sync cable. See also the `TransducerInfo.SyncMaster` property, and the `BertecDevice.getSyncronized()` and `BertecDevice.getCurrentSyncDrift()` functions.

BERTECDEVICE.TRANSDUCERINFO.SYNCHMASTER

`boolean BertecDevice.TransducerInfo.SynchMaster`

If set, then the device is the master control of the sync group. This requires the proper Bertec amplifiers connected together using a specific sync cable. See also the `TransducerInfo. Synchronized` property, and the `BertecDevice.getSyncronized()` and `BertecDevice.getCurrentSyncDrift()` functions.

BERTECDEVICE.TRANSDUCERINFO.ZEROLEVELNOISEVALUE

`double BertecDevice.TransducerInfo.ZeroLevelNoiseValue(int channelIndex)`

Returns the zero level noise value for a device and channel. Prior to accessing this property, either `ZeroNow()` must have been called, or `EnableAutozero` must have been set. The value returned is a computed value that can be used for advanced filtering. Valid values are always zero or positive; negative values indicate either no zeroing or some other error.

DATA EVENTS

The Device Library supports event callbacks – one for handling data collected by the system, and the other for status events. Using these callback events for data collection is the fastest way to gather data, but since these events operates in a multi-threaded environment, it can introduce some programming considerations for you. You should be aware of multithreading in your development environment before using these functions.

To use the data events, simply derive a class from `BertecDevice.DataEventListener` or `BertecDevice.StatusEventListener`, depending on if you want data from the system, or wish to handle status event changes.

BERTECDEVICE.DATAEVENTLISTENER

To use the Data Events, implement a class based on `BertecDevice.DataEventListener` with a method function called `dataEventReceived(DataEvent event)`, like so:

```
class myHandler implements BertecDeviceJava.BertecDevice.DataEventListener
 {
   public void dataEventReceived(DataEvent event)
   {
     processYourData(...);
   }

 };
```

The `dataEventRecieved` function will get a `BertecDevice.DataEvent` event object each time it is called. Any errors are given in the `DataEvent.samples` value.

BERTECDEVICE.DATAEVENT

This event object contains the force place or other device data. Each time your `dataEventReceived` implementation is called, this object will be populated with three data values:

BERTECDEVICE.DATAEVENT.CHANNELS

This is the number of channels that are present in the event. This will always be the total number of channels for all devices connected. Thus if you have a single force plate with six channels, this value will be 6; but if you have two devices, one with four channels and the other with six, then this value will be 10. You use this value to iterate through the `data` array.

BERTECDEVICE.DATAEVENT.SAMPLES

This is the number of samples of the channels available in data. If you think of the `channels` value as the number of columns, then the samples is the number of rows. The total number of values available in the `data` array will always be channels · samples.

If this value is less than zero, it indicates that there was some error while receiving the data. Please consult the list of possible error values in the `DataPoll` section.

BERTECDEVICE.DATAEVENT.DATA

This is an array of doubles that contain the received data from the force device. You can iterate through this array to capture or otherwise process the data.

EXAMPLE

```
public void dataEventReceived(DataEvent event)
{
   int index = 0;
   for (int row = 0; row < event.samples; ++row)
   {
      for (int col = 0; col < event.channels; ++col)
      {
         Do_Something(event.data[index]);
         ++index;
      }
   }
}
```

Here your ***Do_Something*** function would perhaps record the data values to a file.

BERTECDEVICE.STATUSEVENTLISTENER

To use the Status Events, implement a class based on `BertecDevice.StatusEventListener` with a method function called `statusEventReceived(StatusEvent event)`, like so:

```
class myHandler implements BertecDeviceJava.BertecDevice.StatusEventListener
 {
   public void statusEventReceived(StatusEvent event)
   {
     processTheStatus(...);
   }

 };
```

The `statusEventRecieved` function will get a `BertecDevice.StatusEvent` event object each time it is called.

BERTECDEVICE.STATUSEVENT

The `StatusEvent` object is just a container for a single data value, `status`. This `status` value is the same as what is returned by the `BertecDevice.getStatus()` function call – see the table of status codes under the `getStatus` function for what these values are.

You will only get this event when and if the status event changes.

TROUBLESHOOTING

If your application will not launch, make sure that both the BertecDeviceDLL.dll and ftd2xx.dll are in the same folder as your application, and that the BertecDeviceJAVA.jar has been registered with your development system or install deployment solution.

For any other issues, please contact Bertec Technical Support.

DOCUMENT REVISION HISTORY

Date	Revision	Description	Author
01/15/2009	1.00	Initial Revision	Todd Wilson
03/28/2012	1.80	Updated with current version	Todd Wilson
06/30/2012	1.81	Removed Sync Drift function; added Sync Cable and Sync Counter Reset functions and additional status codes.	Todd Wilson
3/24/2014	1.82	Added sort ordering, usb thread priority functions, corrected typographical errors	Todd Wilson