Bertec Device Interface Library for .NET

Developer Documentation

Version 2.14
March 2019

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. BertecDeviceNET.dll
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 .NET 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 USB devices and implementing different protocols and calibrations for each Bertec Device Library for .NET manages all the needed interactions and provides a stream of calibrated data to your program or data analysis project to capture for storage or process in real-time. The Library also provides facilities for zeroing of the plate data (either on-demand for tare loading, or automatic for low or no loading), sample averaging, low-pass filtering, multiple device support, and data synchronization (both external and internal). Automatic detection of device disconnection and reconnection is handled by the Library with little need for your application to be directly involved. Depending on the hardware available, additional signaling both in and out with external devices can be controlled. Data results can be retrieved by using either on-demand polling or using event signals.

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

The .NET class library is also COM-accessible, allowing to be used by COM-capable programs.

Sample code is provided in the BertecExampleNET.cs file.

If you are doing development in a C or C++ environment, please refer to the BertecDevice.pdf and BertecExample.cpp files.

If you have any developmental questions on using this library or example, 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. The Center of Pressure is computed as Moments divided by Force (ex: Mx / Fz).

USING THE LIBRARY WITH YOUR PROJECT

The Bertec Device Library for .NET consists of a two DLL files - BertecDevice.DLL the provides the underlying functionality and a .NET wrapper class library that exposes the corresponding .NET functionality and class methods. You will need to add a Reference to the BertecDeviceNET class library inside of your project, and deploy both it and the BertecDevice.DLL file along with your application.

In order for the Library to method properly, the FTDI drivers will need to be installed; in particular, the FTD2XX.DLL library will need to be accessible somewhere in the path. Depending on your desired deployment method, this can either be part of your application (residing in the same directory as the BertecDevice.DLL file) or in the system folder. Current FTDI D2XX Device Driver installations can be downloaded from [http://www.ftdichip.com/Drivers/D2XX.htm].

If either the BertecDevice.DLL or FTD2XX.DLL files are not accessible, the .NET Library will throw an exception when the `BertecDevice` object is created. Your program should catch and respond these exceptions, and not attempt to use the .NET Library until they are resolved.

GATHERING DATA

Reading data from an attached device generally consists of just a few steps:

1. Create the `BertecDevice` object to init the Library and load the needed device DLL file.
2. Bind an event handler to `BertecDevice.OnData` if your application supports data callback events.
3. Call `BertecDevice.Start`.
4. Wait for connected devices to become ready by using either `BertecDevice.Status` or a bound `OnStatus` event.
5. Poll using `BertecDevice.ReadBufferedData`, or use the event hander.
6. Perform any operation your application needs, such as data collection or analysis.
7. Call `BertecDevice.Stop`
8. Dispose of the `BertecDevice` object.

Step 1: Create the `BertecDevice` object.

Creating the `BertecDevice` object will set up internal data in the Library and ready it for use; you must do this before you can use any other functionality in the Library. Note that creating multiple instances of this object is not supported and will result in unexpected behavior.

Step 2: Bind your event handler to `BertecDevice.OnData`.

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 your application uses event callbacks (the suggested method), you will need to register the event handler with the Library prior to calling `BertecDevice.Start`.

Step 3: Call `BertecDevice.Start`.

To actually detect any connected devices and begin gathering data, you must call `BertecDevice.Start` Doing so will start the device detection process and perform the required steps that each connected device needs. Data will begin to be read as soon as it becomes available. If your application has bound an event handler to `BertecDevice.OnData`, then data will be

presented to that method; otherwise your application need to repeatedly call `BertecDevice.ReadBufferedData` in order to retrieve any buffered data.

Step 4: Wait for connected devices to become ready.

By using either the `BertecDevice.Status` method or else a bound `BertecDevice.OnStatus` event handler your application can wait for the Library to report that devices have been detected and are now available. Once the Library reports a status of `DEVICES_READY` you can expect to start getting data on either the callback or the polling method.

Step 5: Handle data via an event handler or else poll using `BertecDevice.ReadBufferedData`.

If your application has bound an event handler to `BertecDevice.OnData`, it will be called with a block of data each time one becomes available. Alternatively, if your application has its own method of collecting data it can repeatedly call `BertecDevice.ReadBufferedData` to retrieve the currently buffered data one block at a time.

Step 6: Perform operations.

Take the data collected by the Library and perform some functionality with, such as capturing to a data file or presenting the values in a UI somewhere.

Step 7: Call `BertecDevice.Stop`

Once you have completed your data gathering call `BertecDevice.Stop` to end all data reading and release all USB connections. Any connected devices will be reset and will no longer send data. Currently active callbacks will remain active but will no longer receive data, and `BertecDevice.ReadBufferedData` will return an empty result. To resume data collection your application must call `BertecDevice.Start` again which will start the device detection process over again. Note that you should not use a Start/Stop cycle as a method to control data coming in; it is much better to simply call `BertecDevice.Start` and leave the Library to freely run in the background, using a flag in your data callback to determine if you should ignore or process the data.

Step 8: Dispose of the `BertecDevice` object

Once you are completely done with the Library, you will need to dispose of the `BertecDevice` object to release any connections that are still open and free all memory used by the Library, including de-registering all callbacks. Failure to do so may introduce memory or other resource leaks.

USING DATA POLLING

Using data polling instead of callbacks involves repeatedly calling the `BertecDevice.ReadBufferedData` method until it returns a value indicating there is no more data left in the internal buffer. The helper property `BertecDevice.BufferedDataAvailable` can be used to pre-determine how much data is currently available in the internal buffer. Note that there may actually be more data available than what `BertecDevice.BufferedDataAvailable` reports at the time it was called; this is due to the Library continually reading the USB device and adding data to the internal buffer as it comes in.

Your application must ensure that reading the buffered data is done frequently in order to avoid any possible data loss; by default the internal buffer will contain up to 100 samples before older unread samples are discarded unless `BertecDevice.MaxBufferedDataSize` is set with a larger buffer size. Be aware that increasing the size of the internal buffer can dramatically affect the amount of memory that your application will consume, with no gains in terms of performance.

A very simplistic example of data polling with without any error handling might be similar to the following:

```
BertecDeviceNET.BertecDevice bDevice = new BertecDeviceNET.BertecDevice();
bDevice.Start();
while (bDevice.Status != BertecDeviceNET.StatusErrors.DEVICES_READY)
{
   waitingForDevices();
}
BertecDeviceNET.DataFrame[] dataFrames = new BertecDeviceNET.DataFrame[0];
while (External_Flag_To_Keep_Running_Is_True)
{
   while (bDevice.ReadBufferedData(ref dataFrames) > 0)
      processYourData(dataFrames);
}
bDevice.Stop();
```

Once the Library detects devices and performs the needed setup methods, the `BertecDevice.Status` property will return that the devices are ready. Your application would then proceed to the data collection loop with an allocated data frame buffer. The `dataFrames` array will be populated with a separate data frame for each connected device.

Inside the data collection loop the inner-most loop exhaustively reads all of the data available and then does something with it; once all data has been read the Library will return a zero value which returns code flow back to the outer loop. The outer loop simply checks to see if the keep-running flag is true, and then repeats the process as long as it is.

USING EVENTS

Events are the preferred method for reading data from the Library. All event calls are made using a separate processing thread outside of your application's own main thread, which may have implications based on your framework or UI components. Because of this you may need to design additional signaling and buffering functionality into your application to bridge these two separate processing spaces. The advantage of using events instead of data polling is that overall the data collection process is much simpler and there is a significantly lower risk of missing data due to your main application being busy with some other process. Given good design your application can also be made much more responsive to changes on the device, resulting in a more fluid experience for the user.

To use events, bind an event handler to `BertecDevice.OnData`. You can bind multiple event handlers and they will each get called in turn with a copy of the received data block. The order in these multiple callbacks are invoked should not be relied upon.

A very simplistic example of using events with without any error handling might be similar to the following:

```
class myHandlerClass
 {
   public void handleData(BertecDeviceNET.DataFrame[] dataFrames)
   {
     processYourData(dataFrames);
   }
 };

BertecDeviceNET.BertecDevice bDevice = new BertecDeviceNET.BertecDevice();
myHandlerClass handler = new myHandlerClass();
BertecDevice.OnData += handler.handleData;
bDevice.Start();

      ...your main program runs...

bDevice.Stop();
```

As apposed to the polling version, using events do not explicitly require you to check for the status of the Library or devices; your event handler will be invoked as soon as data becomes available.

ERROR CHECKING AND HANDLING

Most Library methods can return error codes and your application should check for them. See the section on Error Codes for more information on what each code means and possible resolutions. Error codes values are part of the `BertecDeviceNET.StatusErrors` enum collection.

Whenever the status of the Library changes, such as an error or loss of synchronization, any previously declared status event handlers set by `BertecDevice.OnStatus` are invoked with the new status value. The current status can always be retrieved by reading the `BertecDevice.Status` property.

DATA PROCESSING AND FORMAT

Since data can flow into the computer at a very rapid rate, it is critical that your program handle it as promptly as possible – buffering it in a pre-allocated memory block is preferred. Should data not be read fast enough it will start to be lost, with older data being overwritten as new data is collected. In this case a `StatusErrors.DATA_BUFFER_OVERFLOW` will occur.

The data block (`BertecDeviceNET.DataFrame`) presented to your application will always be the same size, regardless of how many channels the device provides or if it supports extended sync capabilities or other features. The Library has the ability to accommodate up to four devices connected at any one given time, each one having up to 32 channels of data along with time stamps and sync/aux hardware pin states.

BERTEC DEVICE LIBRARY METHODS

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

`BertecDeviceNET.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.

`BertecDeviceNET.DataFrame` : a single frame of data consisting of up to 32 device channels, along with a timestamp and aux/sync pin values.

`BertecDeviceNET.DataEventHandler` : this is the event handler delegate for getting data from the Library. Use this to put the call to your own event handler into the `BertecDevice.OnData` event.

`BertecDeviceNET.StatusEventHandler` : this is the *status change* event handler delegate. Use this to put the call to your own event handler into the `BertecDevice.OnStatus` event.

`BertecDeviceNET.DeviceSortEventHandler` : this event handler is called whenever there are more than one device connected to the computer, and can be used to provide custom sorting of attached devices (reordering which device will be presented first to the data handler or eader). Use this to put the call to your own event handler into the `BertecDevice.OnDeviceSort` event.

`BertecDeviceNET.LogEventHandler` : this event handler is called whenever something is about to be written to the device log. Your application can use this to support your own logging . Use this to put the call to your own event handler into the `BertecDevice.OnLogSort` event.

`BertecDeviceNET.StatusErrors` : enum list of error values.

`BertecDeviceNET.AutoZeroStates` : enum list of auto zeroing states.

`BertecDeviceNET.AuxModeFlags` : enum list of possible aux pin modes.

`BertecDeviceNET.SyncModeFlags` : enum list of possible sync pin modes.

`BertecDeviceNET.ClockSourceFlags` : enum list of possible clock sources.

`BertecDeviceNET. ComputedChannelFlags`: enum list of possible computed channels.

`BertecDeviceNET. CommonDimensions`: enum list of common plate sizes as supplied by Bertec Corporation.

`BertecDeviceNET.DeviceInfo` : information about each device in the `DeviceList` collection, including the channel names.

`BertecDeviceNET.DeviceList` : a collection list of the attached devices.

BERTECDEVICE

`berecObject = new BertecDevice()`

Creating the `BertecDevice` object initializes the Library and prepares it for use but does not start the actual device interaction – `BertecDevice.Start` must be used to begin the data discovery and data collection process. 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.

If the Library is unable to locate and load the dependant DLL files (BertecDevice.DLL and FTD2XX.DLL) it will throw an exception that your application should catch and handle. The BertecDevice.DLL is part of the Bertec Device SDK and should be deployed along with the .NET library. The FTD2XX.DLL file is provided by Future Technology Devices International and is required to communicate with their USB devices. Your Windows system may already have this deployed (it is used by other USB devices), but it is suggested that you also deploy the FTDI D2XX driver installation from [http://www.ftdichip.com/Drivers/D2XX.htm] as part of your own production installation.

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 `using(bertecObject)`, or allow the .NET garbage collection to take care of it. Failing to call this may leave devices running and possibly introduce memory leaks.

BERTECDEVICE.START

`int BertecDevice.Start()`

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

BERTECDEVICE.STOP

`int BertecDevice.Stop()`

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

DATAEVENTHANDLER DELEGATE

BERTECDEVICE.ONDATA

`delegate void DataEventHandler(BertecDeviceNET.DataFrame[] dataframe)`

`event BertecDevice.OnData`

To use the data event functionality as provided by the Library, you will need to bind your event handler to `BertecDevice.OnData`. To stop using the event without stopping data acquisition, simply unbind it from `BertecDevice.OnData`. Disposing of the `BertecDevice` object will automatically unbind all event handlers. Your event handler method should match the signature of the `DataEventHandler` delegate.

Your event handler will be invoked each time there a block of data available and is called within the context of a separate thread from your main process. This *must* be taken into account your application's design; failure to do so will typically result in strange user interface behavior.

The `dataFrame` parameter is an array consisting of one or more `BertecDeviceNET.DataFrame` objects, one for each connected device (up to a maximum of four). Refer to the section on Data Frames for information about the format and type of the data block.

STATUSEVENTHANDLER DELEGATE

BERTECDEVICE.ONSTATUS

`delegate void StatusEventHandler(BertecDeviceNET.StatusErrors status)`

`event BertecDevice.OnStatus`

To use the status event functionality you will need to bind your event handler to `BertecDevice.OnStatus`. To stop using the event, unbind it from `BertecDevice.OnStatus` .Disposing of the `BertecDevice` object will automatically unregister all event handlers. Your event handler method should match the signature of the `StatusEventHandler` delegate.

Your event handler will be invoked each time there is a *change* (from A to B but never from A to A or B to B) in the status of code of the Library and will be called within the context of a separate thread from your main process. This *must* be taken into account your application's design; failure to do so will typically result in strange user interface behavior.

The `status` value passed to the callback is the same as what reading `BertecDevice.Status` would return. These are defined in the header file and are also documented further down.

It is important that you process the status change as fast a possible and return as to not possibly interrupt the data collection process.

DEVICESORTEVENTHANDLER DELEGATE

BERTECDEVICE.ONDEVICESORT

`delegate void DeviceSortEventHandler(BertecDeviceNET.DeviceList infos, ref int orderArray[])`

`event BertecDevice.OnDeviceSort`

To use the device sort order functionality you will need to bind your event handler to `BertecDevice.OnDeviceSort`. To stop using the event, unbind it from `BertecDevice.OnDeviceSort`. Disposing of the `BertecDevice` object will automatically unregister all event handlers. Your event handler method should match the signature `DeviceSortEventHandler` delegate.

Event handlers that are registered will be called each time the Library finishes discovering the list of devices attached to the computer. The `infos` object is a list of the devices discovered, which can be used to manipulate the `orderArray` to change 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 each `DeviceInfo` item in 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 new ordering of devices will be reflected in the outgoing data stream. This allows your project to always have a consistent ordering of devices that may be different from the default hardware id/system connection order.

DEVICETIMESTAMPEVENTHANDLER DELEGATE

BERTECDEVICE.ONDEVICETIMESTAMP

`delegate long DeviceTimestampEventHandler(int deviceIndex)`

`event BertecDevice.OnDeviceTimestamp`

Certain specific applications may need the ability to override the hardware timestamp that is provided by the hardware devices. This callback will be invoked each time a block of data is processed from the USB device, and will only be called when the clock source is set to CLOCK_SOURCE_INTERNAL; if the clock source is set to any one of the external modes then the timestamp callback will not be used.

To use the device sort order functionality you will need to bind your event handler to `BertecDevice.OnDeviceTimestamp`. To stop using the event, unbind it from `BertecDevice.OnDeviceTimestamp`. Disposing of the `BertecDevice` object will automatically unregister all event handlers. Your event handler method should match the signature `DeviceTimestampEventHandler` delegate.

The event handler will be called each time the Library reads and processes a block of data from the USB device, provided the clock source is set to CLOCK_SOURCE_INTERNAL (the default). If the clock source is set to one of the external sync pin modes then this callback will not be invoked.

The callback implementation must return a non-repeating 64 bit timestamp value, expressed in 8ths of a milliseconds (thus 1 ms equals a step of 8). This is done to match the same format as the `DataFrame.timestamp` data value.

BERTECDEVICE.STATUS

`BertecDeviceNET.StatusErrors BertecDevice.Status`

This property returns the current status value of the Library. This is the same as what any bound `OnStatus` event handler will get.

BERTECDEVICE.BUFFEREDDATAAVAILABLE

`int BertecDevice.BufferedDataAvailable`

This property returns how many blocks or "frames" of data are available to be read from the internal data buffer. It is designed to be used in conjunction with `BertecDevice.ReadBufferedData` and should *not* be used when event handlers have been set. This property will return zero if there are currently no blocks of data available to be read, or a negative value indicating an error. A positive non-zero value indicates there are at least that many data blocks available to be read.

BERTECDEVICE.READBUFFEREDDATA

`int BertecDevice.ReadBufferedData(ref BertecDeviceNET.DataFrame[] dataOut)`

Instead of using the event handlers, you can use the `BertecDevice.ReadBufferedData` function to periodically pull the data from the internal buffer maintained by the Library. This polling must be done frequently enough to avoid any possible data loss. Each call to `BertecDevice.ReadBufferedData` will take one data frame from the internal buffer, copying the values into your passed array object and returning how many frames remain in the internal buffer including the one just copied. For example, if the internal buffer contains 4 blocks of data, calling this will take the oldest data frame from the buffer and return 4; in comparison, if there is only a single block of data currently available, calling this will return a value of 1.

If there is no data remaining in the internal buffer, then this function will return zero and leave the `dataOut` untouched. This makes continually reading from the internal buffer as simple as the following example:

```
while (bDevice.ReadBufferedData(ref dataFrames) > 0)
{
   processYourData(dataFrames);
}
```

Negative results from `BertecDevice.ReadBufferedData` indicates an error of some sort, and your application should handle it appropriately.

BERTECDEVICE.CLEARBUFFEREDDATA

`int BertecDevice.ClearBufferedData ()`

Clears all data that is currently in the internal buffer. Any unread data is immediately lost, even if event handlers are being used. Returns zero for success.

BERTECDEVICE.MAXBUFFEREDDATASIZE

`int BertecDevice.MaxBufferedDataSize`

Gets or sets how many data samples that the Library will buffer before discarding old data. The default is 100 samples, which is appropriate for most systems. If you feel that your application cannot keep up or are using a very slow polling frequency, then changing this value may help but you may be better off using the callback methods. Note that large values (1000 or more) will dramatically increase memory usage and may impact program performance.

Calling this function will also discard all currently buffered data, so it suggested that your application calls this before calling `BertecDevice.Start`.

BERTECDEVICE.DEVICES

`BertecDeviceNET.DeviceList BertecDevice.Devices`

Returns a copy all of the devices (`DeviceInfo` objects) currently detected by the Library. If there is are no devices this function will return an empty list. See the `DeviceInfo` object documentation further down.

BERTECDEVICE.DEVICECOUNT

`int BertecDevice.DeviceCount`

Returns the number of supported devices connected to the computer. Only valid once `BertecDevice.Start` has been called and devices have actually been found (that is, either `BertecDevice.Status` or an `OnStatus` handler indicates a status value equal to `BertecDeviceNET.StatusErrors.DEVICES_READY`).

BERTECDEVICE.DEVICEINFO

`BertecDeviceNET.DeviceInfo BertecDevice.DeviceInfo(int index)`

Returns a copy of the information about the given device index. If there is no device at the given index then this function will return an empty result. This is a convenience method that can be used instead of the `BertecDevice.Devices` property which returns the entire list of devices.

BERTECDEVICE.DEVICESTATUS

`string BertecDevice.DeviceStatus(int index)`

This is a convenience method that will return a device's current status directly instead of reading the entire device info into a `DeviceInfo` object and accessing the `Status` property. If there is no device at the given index then this function will return a `BertecDeviceNET.StatusErrors.INDEX_OUT_OF_RANGE` value

BERTECDEVICE.DEVICECHANNELNAMES

`String[] BertecDevice.DeviceChannelNames(int index)`

This is a convenience method that will return an array of channel names for the device at the given index instead of reading the entire device info into a `DeviceInfo` object and accessing the `ChannelNames` property. If there is no device at the given index then this function will return an empty array.

BERTECDEVICE.DEVICESERIALNUMBER

`string BertecDevice.DeviceSerialNumber(int index)`

This is a convenience method that will return a device's serial number directly instead of reading the entire device info into a `DeviceInfo` object and accessing the `SerialNumber` property. If there is no device at the given index then this function will return an empty string.

BERTECDEVICE.DEVICEMODELNUMBER

`string BertecDevice.DeviceModelNumber(int index)`

This is a convenience method that will return a device's model number directly instead of reading the entire device info into a `DeviceInfo` object and accessing the `SerialNumber` property. If there is no device at the given index then this function will return an empty string.

BERTECDEVICE.AVERAGINGSIZE

`int BertecDevice.AveragingSize`

This property 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.LowpassFilterSamples`

This property 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. Setting `LowpassFilterSamples` to 1 or less will turn filtering off (the default). 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 and thus values coming from a connected device can be extremely high or low. Calling the `ZeroNow` method 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 `EnableAutozero`.

For best results you should call this right after your application first receives a `BertecDeviceNET.StatusErrors.DEVICES_READY` value from the `BertecDevice.Status` property or your `BertecDevice.OnStatus` event handler.

BERTECDEVICE.AUTOZEROING

`int BertecDevice.AutoZeroing`

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 this property with a non-zero (true) value will cause the Library to monitor the data stream and continually reset the zero baseline values. Set this to zero (false) to turn it back 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

`BertecDeviceNET.AutoZeroStates BertecDevice.AutozeroState`

This `AutoZeroState` property returns the current state of the autozero feature. This property is rarely needed. Your program will need to poll this on occasion to find the current state – there is no event for when it changes.

The method 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 and is being applied. The Library will continually attempt to zero automatically.

BERTECDEVICE.USBTHREADPRIORITY

`int BertecDevice.UsbThreadPriority`

This property 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 UI from running). The default system scheduling of the USB reading thread should be suitable for most applications.

BERTECDEVICE.MASTERSLAVEMODE

`int BertecDevice.MasterSlaveMode`

Enables or disables the slave/master setting when more than one device connected. If the flag is set to zero (false) or if there is only one device connected, then the sync mode defaults to a value of `SYNC_NONE` (sampled) and internal hardware clock timers are reset (if present).

If this property is set to a non-zero (true) value and there is two or more devices, then the first device is set as `SYNC_OUT_MASTER` and all other devices are set as `SYNC_IN_SLAVE`, and internal hardware clock timers are reset (if present).

By default Master/Slave mode is enabled and will automatically be set up if two or more devices are present. Setting this flag to zero (false) after `BertecDevice.Init` but before calling `BertecDevice.Start` will prevent this and cause all devices to run as `SYNC_NONE`. You can change this at any point during runtime by setting this flag to the desired value.

Setting this value to a non-zero (true) value is exactly the same as calling `BertecDevice.SetSyncPinMode` with `SYNC_OUT_MASTER` for the first device and `SYNC_IN_SLAVE` for all others and resetting all clock timers to 0.

Note that for best results there should be a sync cable connected between two hardware amplifiers. For configurations without sync-capable hardware (ex: internal amplified force plates) the Library will attempt to us whatever hardware clocks are available to sync the data sources the best that it can.

BERTECDEVICE.SETSYNCPINMODE

`int BertecDevice.SetSyncPinMode(int deviceOndex, BertecDeviceNET.SyncModeFlags flags)`

Sets the SYNC pin operating mode for those hardware devices that support it; for all others it does nothing. This will override any exiting master/slave sync relationship that is currently established. Depending on the connected hardware, this may also enable driving the external sync pin as a TTL signal or read it as a continually sampled input which will be presented on the `DataFrame.syncData` value.

`BertecDeviceNET.SyncModeFlags` enums:

Enum Name	Value	Description

SYNC_NONE	0x00	The SYNC pin is an input, but its value is not interpreted in any way. This mode is also known as SYNC_IN_SAMPLED. This is the default power-up mode.
SYNC_OUT_MASTER	0x01	The SYNC pin is outputting a 1kHz square wave clock with a reference mark embedded every 2000ms.
SYNC_IN_SLAVE	0x02	The SYNC pin is inputting a 1kHz square wave clock with optional reference marks.
SYNC_OUT_PATGEN	0x04	The SYNC pin is outputting a random pattern. This is useful for debugging.
SYNC_IN_CONTINUOUS	0x05	The SYNC pin is inputting a continuous 1kHz square wave clock without reference marks.
SYNC_IN_STROBED	0x06	The SYNC pin is inputting a transient 1kHz square wave clock without reference mark.
SYNC_OUT_CONTINUOUS	0x07	The SYNC pin is outputting a continuous 1kHz square wave clock without reference marks.
SYNC_OUT_INSTANT	0x08	The SYNC pin is outputting the value most recently set via the OUTPUT command.

BERTECDEVICE.SETAUXPINMODE

`int BertecDevice.SetAuxPinMode(int deviceOndex, BertecDeviceNET.AuxModeFlags flags)`

Sets the AUX pin operating mode for those hardware devices that support it; for all others it does nothing. Depending on the connected hardware and the passed mode flags, this may enable driving the external aux pin as a TTL signal or read it as a continually sampled input which will be presented on the `DataFrame.auxData` value.

`BertecDeviceNET.AuxModeFlags` enums:

Enum Name	**Value**	**Description**
AUX_NONE_AUX_IN_ZERO	0x00	AM6500: The AUX pin is an input, but its value is not interpreted in any way. AM6800/AM6817: the input is taken from the ZERO pin, and a logic low level keeps the analog output signals zeroed. This is the default power-up mode.

AUX_IN_SAMPLED	0x01	The AUX/ZERO pin is an input, and its value is not interpreted in any way.
AUX_OUT_INSTANT	0x02	The AUX is outputting the value most recently set via the OUTPUT command.
AUX_OUT_PATGEN	0x04	The AUX pin is outputting a random pattern. This is useful for debugging.

BERTECDEVICE.SETSYNCAUXPINVALUES

`int BertecDevice.SetSyncAuxPinValues(int deviceIndex, int syncValue, int auxValue)`

Sets both the SYNC and AUX output pins to the passed values; these values only take effect if the given pin has been set to SYNC_OUT_INSTANT or AUX_OUT_INSTANT. If the pin has not been set to SYNC_OUT_INSTANT or AUX_OUT_INSTANT or the device does not support the setting these values, then the passed value(s) are ignored. Note that you must pass both values even if you intend to only set one pin or the other pin is not set to instant output mode.

BERTECDEVICE.RESETSYNCCOUNTERS

`int BertecDevice.ResetSyncCounters()`

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

BERTECDEVICE.RESETDEVICECLOCK

`int BertecDevice.ResetDeviceClock(int deviceIndex, int64 newTimestampValue)`

This will set the given device's internal 64-bit clock timer value to the passed newTimeStampValue parameter. This is only valid for devices that support the extended Timestamp feature and will be ignored otherwise and return a UNSUPPORED_COMMAND error. The change takes place immediately, but due to signal propagation on the USB line this may take up to two samples for the new value to actually appear in the incoming data stream.

BERTECDEVICE.RESETALLDEVICECLOCKS

`int BertecDevice.ResetAllDeviceClocks(int64 newTimestampValue)`

This will set all of the attached device's internal 64-bit clock timer values to the passed newTimeStampValue parameter. This is only valid for devices that support the extended Timestamp feature and will be ignored otherwise and return a UNSUPPORED_COMMAND error. The change takes place immediately, but due to signal propagation on the USB line this may take up to two samples for the new value to actually appear in the incoming data stream.

BERTECDEVICE.RESETDEVICECLOCKATTIME

`int BertecDevice.ResetDeviceClockAtTime(int deviceIndex, int64 newTimestampValue, int64 futureConditionTime)`

This will set the given device's internal 64-bit clock timer value to the passed `newTimeStampValue` parameter once the device's internal clock timestamp value has reached or exceeded the `futureConditionTime` value. For example, if the internal clock timestamp is currently at 100 and this command is issued with a condition of 200 and a new value of 0, once the internal clock reaches 200 it will be reset back to 0. This reset is only done once; it will not reset multiple times. This is only valid for devices that support the extended Timestamp feature and will be ignored otherwise and return a UNSUPPORED_COMMAND error.

BERTECDEVICE.RESETALLDEVICECLOCKSATTIME

`int BertecDevice.ResetAllDeviceClocksAtTime(int64 newTimestampValue, int64 futureConditionTime)`

This will set all of the attached device's internal 64-bit clock timer values to the passed `newTimeStampValue` parameter once each device's internal clock timestamp value has reached or exceeded the `futureConditionTime` value (each device is triggered independently). For example, if the internal clock timestamp is currently at 100 and this command is issued with a condition of 200 and a new value of 0, once the internal clock reaches 200 it will be reset back to 0. This reset is only done once; it will not reset multiple times. This is only valid for devices that support the extended Timestamp feature and will be ignored otherwise and return a UNSUPPORED_COMMAND error.

BERTECDEVICE.SETEXTERNALCLOCKMODE

`int BertecDevice.SetExternalClockMode(int deviceIndex, BertecDeviceNET.ClockSourceFlags newMode)`

Enables the ability for the Library to clock the device's data stream against an external sync or other clock source that is tied into the physical SYNC connection on the amplifier. This allows the signal being applied to the SYNC pin to effectively override the internal 1000hz hardware clock on the device, allowing the data to be either under or over sampled as needed. Depend on the values of the newMode flags, the data may be either delay-sampled by up to 4.875ms or instead skipped/replicated as needed.

Using an external clock disables any Averaging, low-pass Filtering, and multiple plate sync abilities that were previously set up, and resets the Sync Pin Mode to a value of SYNC_NONE. After setting this mode you should also call `BertecDevice.ClearBufferedData` to discard any already collected data that you would otherwise need to process.

Flag Name	Value	Explanation

CLOCK_SOURCE_INTERNAL	0x00	This is the default state and the SDK will present data at the native device rate (1000hz). Averaging will affect this. All Sync and Aux modes are available, including multiple device sync.
CLOCK_SOURCE_EXT_RISE	0x01	This will cause data to be presented whenever the SYNC pin changes from low (0) to high (1), which can be higher (up to 4000hz) or lower (down to 1hz). Averaging is disabled, and the SYNC mode is forced to SYNC_NONE. All other Aux modes are available, but multiple device sync is disabled.
CLOCK_SOURCE_EXT_FALL	0x02	This will cause data to be presented whenever the SYNC pin changes from high (1) to low (0), which can be higher (up to 4000hz) or lower (down to 1hz). Averaging is disabled, and the SYNC mode is forced to SYNC_NONE. All other Aux modes are available, but multiple device sync is disabled.
CLOCK_SOURCE_EXT_BOTH	0x03	This will cause data to be presented whenever the SYNC pin changes from either a low-to-high or high-to-low state. Averaging is disabled, and the SYNC mode is forced to SYNC_NONE. All other Aux modes are available, but multiple device sync is disabled.
CLOCK_SOURCE_NO_INTERPOLATE	0x80	By default the ClockSource logic will attempt to perform a fractional delay on the input data. This can cause the data signal to appear to be delayed by up to 4.875ms. If such a delay would cause problems with your code path you will need to pass this bit flag along with the clock source to change from a fractional delay to a simpler skip-and-fill. Skip-and-fill will either omit or duplicate channel data depending on when the edge signal occurs in the data flow.

Note: your hardware needs will dictate if this mode of operation is suitable for your configuration. Any hardware that is to be used as an external clock signal must be capable of delivering the proper signal (voltages/pattern) to the SYNC connection, otherwise random or no samples will result in the data stream.

BERTECDEVICE.AGGREGATEDEVICEMODE

`bool BertecDevice.AggregateDeviceMode`

Enables or disables the ability to combine the output of two plates as one long virtual plate. This can be enabled or disabled at any point, and the channel values in the `DataFrame` data block will change accordingly. If this mode is turned on then you will always receive only a single `DataFrame` in your even handler or `BertecDevice.ReadBufferedData` call, even if there is more than one device connected. Note that `BertecDevice.DeviceCount` will always return the true number of devices not matter what `AggregateDeviceMode` is set to.

In order for this to work properly both devices must be of the same type, same size, and have the same data channels. You should not try to combine a balance plate with a force plate, or a sport plate with a functional model for example.

BERTECDEVICE.COMPUTEDCHANNELSFLAGS

`BertecDeviceNET.ComputedChannelFlags BertecDevice.ComputedChannelFlags`

Enables or disables the ability to compute certain channels from the force device's FZ, MX, and MY values. If the device does not have the appropriate channels, then setting this will have no effect. Note that turning on the COG and Sway Angle channels may incur a small CPU usage penalty and require setting the subject height via `BertecDevice.SubjectHeight`. The COP calculation is a simple moment over force function and has little to no additional CPU overhead. The COP also does not need the subject height set in order to be used.

This function must be called after the `BertecDevice` object is created but before `BertecDevice.Start()` is called; setting this while devices are actively delivering data will result in an error.

Setting these flags will affect both the channel names (`DeviceInfo.ChannelNames`, `BertecDevice.DeviceChannelNames`) and the actual data frame data (the channel count field in `DeviceInfo`)

Flag Name	**Value**	**Explanation**
`NO_COMPUTED_CHANNELS`	0x00	This is the default state; no additional channels will be computed.
`COMPUTE_COP_VALUES`	0x01	COP x and COP y values will be computed for any matched set of FZ, MX, and MY values. If the force plate is “split” in that it has both a Left and Right component, then additional COP values will be computed and presented.
`COMPUTE_COG_VALUES`	0x02	COG (center of gravity) x and y values will be computed for any matched set of FZ, MX, and MY values. The COG is based on the both the computed COP value and the set Subject Height value, and is calculated using an integrated Butterworth filter.
`COMPUTE_SWAY_ANGLE`	0x04	A SwayAngle channel will be computed using the COG y value against the Subject Height value. If the height has not been set or is invalid, the SwayAngle will be zero.
`COMPUTE_ALL_VALUES`	0x07	All possible computed channels will be generated.

BERTECDEVICE.SUBJECTHEIGHT

`float BertecDevice.SetSubjectHeight`

In order for the computed COG and SwayAngle channels to work properly, the height of the subject on the plate must be set to the correct height. If the subject height is set to zero or is otherwise invalid, then both the COG and SwayAngle computed values will be zero.

This is a write-only properly with the height expressed in millimeters (ex: 1.5 meters should be passed as 1500).

Unlike the computed channels, changing the subject height while data is being collected *is* supported and is expected.

BERTECDEVICE.DATARATE

`float BertecDevice.DataRate`

This is a read-only property that returns the dynamically computed value of the overall sample rates from the connected devices, in hertz. This value is updated approximately every 100 ms and will typically be around 1000hz. Note that there will be some “flutter” in this value due to the way the PC hardware operates, but the USB force plates will always deliver data at a fixed 1000hz rate.

BERTECDEVICE.REDETECTCONNECTEDDEVICES

`int BertecDevice.RedetectConnectedDevices()`

Signals the SDK that is should perform a device rescan and reinit all connected devices. This is the same as physically unplugging and then replugging all devices from the USB connection at the same time, and is provided primarily as a way to force a redetection of multiple plates that have been added after the SDK has been started.

The status code LOOKING_FOR_DEVICES will be emitted via the Status callback (if any), followed by either NO_DEVICES_FOUND if there are no devices connected, or DEVICES_READY if one or more devices have been found.

This function does not block and returns immediately.

BERTECDEVICE.SETDEVICELOGDIRECTORY

`static void BertecDevice.SetDeviceLogDirectory(string outputFolder,int maxAgeDays)`

Sets or changes the output folder for the device logs and how long existing log files should be kept. By default log files are kept in %TEMP%/bertec-device-logs and are retained for up to 7 days. Passing NULL or an empty string in the `outputFolder` parameter will default to the temp folder and 0 for `maxAgeDays` will turn off old file cleaning. This static function should be called prior to any other Library methods, including `BertecDevice.Init`; otherwise there will be log data split between two separate files as the directory changes.

BERTECDEVICE.CURRENTDEVICELOGFILENAME

`static string BertecDevice.CurrentDeviceLogFilename`

Returns a string containing the current log filename, including the path prefix. This can be used to copy or otherwise reference the output logfile that is being created by the internal diagnostics of the Library.

LOGEVENTHANDLER DELEGATE

BERTECDEVICE.ONLOGEVENT

`delegate void LogEventHandler(string logtext)`

`static event BertecDevice.OnLogEvent`

This will set an event handler that is invoked whenever a new line of text is being written to the device log file. The handler is called within the context of a separate worker thread and as such your application code should handle things appropriately. This functionality is primarily designed as an advanced method for applications to provide additional monitoring and diagnostic displays. The leading digits for the string are the millisecond timestamp when the message was generated, and will differ from when the text is actually logged and sent to your function.

BERTECDEVICENET.DEVICEINFO CLASS

The `DeviceInfo`class contains information about each Bertec USB device connected to the system. You can get the collection of devices by using the `BertecDevice.Devices` property or a single device by using the `BertecDevice.DevicInfo` method.

BERTECDEVICENET.DEVICEINFO.CHANNELCOUNT

`int BertecDeviceNET.DeviceInfo.ChannelCount`

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

BERTECDEVICE.DEVICEINFO.CHANNELNAMES

`string[] BertecDeviceNET.DeviceInfo.ChannelNames`

An array consisting of the channel names that the Transducer is delivering data on. The order of the names is the order of the data in the data frame block that you get via either the `OnData` event or `ReadBufferData` method call.

BERTECDEVICE.DEVICEINFO.DIMENSIONS

`Dimensions BertecDeviceNET.DeviceInfo.Dimensions`

An object containing the width and height of the device, in millimeters. If the device does not containing self-defining information than this will return zero values.

BERTECDEVICE.DEVICEINFO.ORIGINOFFSET

`OriginOffset BertecDeviceNET.DeviceInfo.OriginOffset`

An object containing the X, Y, and Z offset of the top of the device, in millimeters. If the device does not containing self-defining information than this will return zero values.

BERTECDEVICE.DEVICEINFO.HASAUXSYNCPINS

`bool BertecDeviceNET.DeviceInfo.HasAuxSyncPins`

If this property is set to TRUE then the device has full control over the aux and sync pins and all sync and aux functions and external clock control are available.

BERTECDEVICE.DEVICEINFO.HASINTERNALCLOCK

`bool BertecDeviceNET.DeviceInfo.HasInternalClock`

If this property is set to TRUE then the device has an internal 64-bit timestamp value which can be reset (1ms accuracy).

BERTECDEVICE.DEVICEINFO.SAMPLINGFREQ

`int BertecDeviceNET.DeviceInfo.SamplingFreq`

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

BERTECDEVICE.DEVICEINFO.SERIALNUMBER

`string BertecDeviceNET.DeviceInfo.SerialNumber`

The serial number of the device.

BERTECDEVICE.DEVICEINFO.MODELNUMBER

`string BertecDeviceNET.DeviceInfo.ModelNumber`

The serial number of the device.

BERTECDEVICE.DEVICEINFO.VERSION

`unsigned int BertecDeviceNET.DeviceInfo.Version`

Version of the device. mmMM. Minor and Major are broken out from this. Ex: 0x021 is minor version 17, major version 2.

BERTECDEVICE.DEVICEINFO.STATUS

`int BertecDeviceNET.DeviceInfo.Status`

The last-known status of the device. Zero is good, negative is bad.

BERTECDEVICE.DEVICEINFO.ZEROLEVELNOISEVALUE

`float BertecDeviceNET.DeviceInfo.ZeroLevelNoiseValue(int channelIndex)`

Returns the zero level noise value for a device and channel. Prior to using this method, either `BertecDevice.ZeroNow` should be called or `BertecDevice.AutoZeroing` set to TRUE. 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.

ERROR/STATUS CODES

`BertecDeviceNET.StatusErrors` enum values:

Error	Value	Explanation
NO_BUFFERS_SET	-2	There are no internal buffers allocated. This is a critical error.
DATA_BUFFER_OVERFLOW	-4	Data polling wasn't performed for long enough, and data has been lost. Can also occur if your callback method is taking too long.
NO_DEVICES_FOUND	-5	There are apparently no devices attached. Attach a device.
DATA_READ_NOT_STARTED	-6	You have not called `bertec_Start` yet.
DATA_SYNCHRONIZING	-7	Synchronizing, data not available yet.
DATA_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.
DATA_SEQUENCE_MISSED	-9	One or more plates have missing data sequence - data may be invalid
DATA_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
LOOKING_FOR_DEVICES	-45	Scanning for any connected devices. This status will be followed by either DATA_DEVICES_READY or NO_DEVICES_FOUND.
DATA_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
ERROR_INVALIDHANDLE	-100	The `bertec_Handle` passed to a function is incorrect.
UNABLE_TO_LOCK_MUTEX	-101	Unable to properly manage a thread mutex context. This is a fatal error.
UNSUPPORED_COMMAND	-200	The current firmware and/or hardware does not support the function that was just called. The device should still function normally but the expected functionality will not be in effect.
INVALID_PARAMETER	-201	One or more of the parameters to a function call are incorrect (ex: null pointer, negative size, etc).
INDEX_OUT_OF_RANGE	-202	The device index value is negative or more than the number of devices currently attached to the system.
GENERIC_ERROR	-32767	Any error that has no predefined value.

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 BertecDeviceNET.dll 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 method; added Sync Cable and Sync Counter Reset methods and additional status codes.	Todd Wilson
3/24/2014	1.82	Added sort ordering, usb thread priority methods, corrected typographical errors	Todd Wilson
06/01/2016	2.00	Updated document to cover new aux/sync/clock model and revised interface.	Todd Wilson
6/22/2017	2.06	Updated to match current Library revision.	Todd Wilson
10/11/2017	2.07	Updated to clarify functionality.	Todd Wilson
6/19/2018	2.12	Updated to cover the computed data channels and changes to the callback functions.	Todd Wilson
1/30/2019	2.13	Added the Timestamp Callback function.	Todd Wilson
3/12/2019	2.14	Added the Redetect Connected Devices function.	Todd Wilson